UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA UDELAR

PEDECIBA MEC-UDELAR Programa de Desarrollo de las Ciencias Básicas

Coding of Multichannel Signals with Irregular Sampling Rates and Data Gaps

Pablo Cerveñansky Fierro

Tesis de Maestría presentada a la Facultad de Ingeniería de la Universidad de la República en cumplimiento parcial de los requerimientos para la obtención del título de Magister en Informática

Tutores

Dr. Álvaro Martín
Dr. Gadiel Seroussi

Tribunal

Dra. Lorena Etcheverry
Dr. Jorge Silva
Dr. Leonardo Steinfeld

Montevideo, Uruguay
Julio de 2021

Abstract

The relentless advances in mobile communications and the Internet have contributed to a rapid increase in the amount of digital data created and replicated worldwide, which is estimated to double every three years. In this context, data compression algorithms, which allow for the reduction of the number of bits needed to represent digital data, have become increasingly relevant. In this work, we focus on the compression of multichannel signals with irregular sampling rates and with data gaps. We consider state-of-the-art algorithms, which were designed to compress gapless signals with regular sampling, adapt them to operate with signals with irregular sampling rates and data gaps, and then evaluate their performance experimentally, through the compression of signals obtained from real-world datasets. Both the original and our adapted algorithms work in a near-lossless fashion, guaranteeing a bounded per-sample absolute error between the decompressed and the original signals. This includes the important *lossless compression* case, which corresponds to an error bound of zero. The algorithms compress signals by exploiting correlation between signal samples taken at close times (temporal correlation), and, in some cases, between samples from different channels (spatial correlation). For most algorithms we design and implement two variants: a masking (*M*) variant, which first encodes the position of all the gaps, and then proceeds to encode the data values separately, and a non-masking (*NM*) variant, which encodes the gaps and the data values together. For each algorithm, we compare the compression performance of both variants: our experimental results suggest that variant *M* is more robust and performs better in general. Every implemented algorithm variant depends on a window size parameter, which defines the size of the windows into which the data are partitioned for encoding. We analyze the sensitivity of variant *M* of each algorithm to this size parameter: for each dataset, we compress each data file, and compare the results obtained when using a window size optimized for said specific file, against the results obtained when using a window size optimized for the whole dataset. Our experimental results indicate that the difference in compression performance is generally rather small. The last part of our experimental analysis consists of comparing the compression performance of our adapted algorithms, with each other, and with the general-purpose lossless compression algorithm gzip. Following previous experimental results, we only consider variant *M* of each algorithm, and we always use the optimal window size for the whole dataset. Our experimental results reveal that none of the algorithm variants obtains the best compression performance in every scenario, which means that the optimal selection of a variant depends on the characteristics of the data to be compressed, and the error threshold that is allowed. In some cases, even a general-purpose compression algorithm such as gzip outperforms the specific algorithm variants. Nevertheless, we extract some general conclusions from our analysis: for large error thresholds, variant *M* of algorithm APCA achieves the best compression results, while variant *M* of algorithm PCA (and, in some cases of lossless compression, algorithm gzip) are preferred for lower threshold scenarios.

Keywords— multichannel signal compression, near-lossless compression, irregular sampling rate, data gaps.

Table of Contents

Abstract **ii**

Table of Contents **iii**

List of Figures **v**

List of Tables **viii**

1 Introduction **1**

2 Datasets **5**

2.1 Data Format 6

2.2 Dataset IRKIS 7

2.3 Dataset SST 8

2.4 Dataset ADCP 8

2.5 Dataset ElNino 9

2.6 Dataset Solar 10

2.7 Dataset Hail 11

2.8 Dataset Tornado 11

2.9 Dataset Wind 12

2.10 Summary 12

3 Algorithms and Their Variants **14**

3.1 Introduction 15

3.2 General Encoding Scheme 16

3.3 Encoding of Gaps in the Masking Variants 17

3.4 Algorithm Base 18

3.5 Algorithm PCA 19

3.5.1 Example 21

3.5.2 Non-Masking (*NM*) Variant 23

3.6 Algorithm APCA 24

3.6.1 Example 25

3.6.2 Non-Masking (*NM*) Variant 27

3.7 Encoding Scheme for Linear Model Algorithms 27

3.8 Algorithms PWLH and PWLHInt 28

3.8.1 Example 32

3.8.2 Differences Between Algorithms PWLH and PWLHInt 36

3.8.3 Non-Masking (*NM*) Variant 37

3.9 Algorithm CA 38

3.9.1 Example 41

3.9.2 Non-Masking (*NM*) Variant 45

3.10 Algorithm SF 45

3.10.1 Example 51
3.11 Algorithm FR 56
3.11.1 Example 59
3.12 Algorithm GAMPS 61
3.12.1 Example 63
3.12.2 Non-Masking (*NM*) variant 64
3.13 Implementation Details 65

4 Experimental Results 66

4.1 Experimental Setting 67
4.2 Comparison of Masking and Non-Masking Variants 69
4.3 Window Size Parameter 73
4.4 Algorithm Compression Performance 77
4.4.1 Comparison with Algorithm gzip 81
4.4.2 Timestamp Compression 83

5 Conclusions and Future Work 85

Bibliography 89

List of Figures

3.1 General encoding scheme for the evaluated algorithm variants. 16
3.2 Markov model diagram. 18
3.3 Coding routine for algorithm Base. 18
3.4 Coding routine for algorithm variant PCA$_M$. 19
3.5 Decoding routine for algorithm variant PCA$_M$. 20
3.6 Example: Signal consisting of 12 samples. 21
3.7 Example: Algorithm variant PCA$_M$ with $\epsilon = 1$ and $w = 4$. Step 1. 21
3.8 Example: Algorithm variant PCA$_M$ with $\epsilon = 1$ and $w = 4$. Step 2. 22
3.9 Example: Algorithm variant PCA$_M$ with $\epsilon = 1$ and $w = 4$. Step 3. 22
3.10 Coding routine for algorithm variant PCA$_{NM}$. 23
3.11 Coding routine for algorithm variant APCA$_M$. 24
3.12 Auxiliary routine EncodeWindow for algorithm variant APCA$_M$. 24
3.13 Decoding routine for algorithm variant APCA$_M$. 25
3.14 Example: Algorithm variant APCA$_M$ with $\epsilon = 1$ and $w = 256$. Step 1. 25
3.15 Example: Algorithm variant APCA$_M$ with $\epsilon = 1$ and $w = 256$. Step 2. 26
3.16 Example: Algorithm variant APCA$_M$ with $\epsilon = 1$ and $w = 256$. Step 3. 26
3.17 Auxiliary routine FillSegment for linear model algorithms. 27
3.18 Example of the auxiliary routine FillSegment for linear model algorithms. 28
3.19 Coding routine for algorithm variant PWLH$_M$. 29
3.20 Example: checking the valid hull condition. Variant PWLH$_M$ with $\epsilon = 1$. Step 1. 30
3.21 Example: checking the valid hull condition. Variant PWLH$_M$ with $\epsilon = 1$. Step 2. 30
3.22 Auxiliary routine EncodeWindow for algorithm variant PWLH$_M$. 30
3.23 Auxiliary routine EncodeLastWindow for algorithm variant PWLH$_M$. 31
3.24 Decoding routine for algorithm variant PWLH$_M$. 31
3.25 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 1. 32
3.26 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 2. 32
3.27 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 3. 33
3.28 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 4. 33
3.29 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 5. 34
3.30 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 6. 34
3.31 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 7. 34
3.32 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 8. 35
3.33 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 9. 35
3.34 Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 10. 36
3.35 Coding routine for algorithm variant CA$_M$. 38
3.36 Example: key steps of the coding algorithm. Variant CA$_M$ with $\epsilon = 1$. Step 1. . 39
3.37 Example: key steps of the coding algorithm. Variant CA$_M$ with $\epsilon = 1$. Step 2. . 39
3.38 Auxiliary routine EncodeArchivedPoint for algorithm variant CA$_M$. 40
3.39 Auxiliary routine EncodeWindow for algorithm variant CA$_M$. 40
3.40 Decoding routine for algorithm variant CA$_M$. 40
3.41 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 1. 41
3.42 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 2. 41

3.43 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 3. 42

3.44 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 4. 42

3.45 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 5. 43

3.46 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 6. 43

3.47 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 7. 44

3.48 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 8. 44

3.49 Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 9. 45

3.50 Coding routine for algorithm variant SF$_M$. 46

3.51 Auxiliary routine EncodePoint for algorithm variant SF$_M$. 47

3.52 Auxiliary routine EncodeWinStart for algorithm variant SF$_M$. 47

3.53 Auxiliary routine EncodeWinEndStart for algorithm variant SF$_M$. 48

3.54 Auxiliary routine EncodeWinEnd for algorithm variant SF$_M$. 48

3.55 Example: key steps of the coding algorithm. Variant SF$_M$ with $\epsilon = 1$. Step 1. 48

3.56 Example: key steps of the coding algorithm. Variant SF$_M$ with $\epsilon = 1$. Step 2. 48

3.57 Decoding routine for algorithm variant SF$_M$. 50

3.58 Auxiliary routine DecodePoint for algorithm variant SF$_M$. 50

3.59 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 1. 51

3.60 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 2. 51

3.61 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 3. 52

3.62 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 4. 52

3.63 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 5. 53

3.64 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 6. 53

3.65 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 7. 54

3.66 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 8. 54

3.67 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 9. 55

3.68 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 10. 55

3.69 Example: Algorithm variant SF$_M$ with $\epsilon = 1$ and $w = 256$. Step 11. 56

3.70 Coding routine for algorithm variant FR$_M$. 57

3.71 Auxiliary routine GetDisPointsMDP for algorithm variant FR$_M$. 57

3.72 Decoding routine for algorithm variant FR$_M$. 58

3.73 Example: Algorithm variant FR$_M$ with $\epsilon = 1$ and $w = 256$. Step 1. 59

3.74 Example: Algorithm variant FR$_M$ with $\epsilon = 1$ and $w = 256$. Step 2. 60

3.75 Example: Algorithm variant FR$_M$ with $\epsilon = 1$ and $w = 256$. Step 3. 60

3.76 Example: Algorithm variant FR$_M$ with $\epsilon = 1$ and $w = 256$. Step 4. 61

3.77 Coding routine for algorithm variant GAMPS$_M$. 62

3.78 Example: Three signals consisting of 12 samples each. 63

3.79 Example: Algorithm variant GAMPS$_M$ with $\epsilon = 0$ and $w = 256$. Step 1. 64

4.1 CR and RD plots for variants a_M and a_{NM}, for each algorithm $a \in A_M$, for the data type "SST" of the dataset SST. In the RD plot for algorithm PCA we highlight with a red circle the marker for the maximum value (50.78%) obtained for all the tested CAIs. 70

4.2 CR and RD plots for variants a_M and a_{NM}, for each algorithm $a \in A_M$, for the data type "Longitude" of the dataset Tornado. In the RD plot for algorithm APCA we highlight with a blue circle the marker for the minimum value (-0.29%) obtained for all the tested CAIs. 71

4.3 Plots of w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the error parameter e, obtained for the data type "VWC" of the file "irkis-1202.csv" of the dataset IRKIS. 74

4.4 Plots of w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the error parameter e, obtained for the data type "VWC" of the file "irkis-1203.csv" of the dataset IRKIS. In the RD plot for variant PCA$_M$ we highlight with a red circle the marker for the maximum value (10.6%) obtained for all the tested CAIs. 75

4.5 CR and window size parameter plots for every evaluated algorithm, for the data type “Sea Temp.” of the dataset ElNino. For each error parameter $e \in E$, we use blue circles to highlight the markers for the minimum CR value and the best window size (in the respective plots corresponding to the best coding variant). . 78

List of Tables

2.1 Example of a dataset CSV file with the format we defined. 6

2.2 Number of rows, columns, entries and gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (dimensionless), for each file of the dataset IRKIS. The gaps are ignored when calculating the median, mean and standard deviation. 8

2.3 Number of rows, columns, entries and gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in °C), for each file of the dataset SST. The gaps are ignored when calculating the median, mean and standard deviation. 8

2.4 Number of rows, columns, entries and gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in m/s), for each file of the dataset ADCP. The gaps are ignored when calculating the median, mean and standard deviation. 9

2.5 Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in their respective units of measurement), for each data type of the dataset ElNino. The gaps are ignored when calculating the median, mean and standard deviation. 9

2.6 Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in W/m^2), for each data type of the dataset Solar, for year 2011. The gaps are ignored when calculating the median, mean and standard deviation. 10

2.7 Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in W/m^2), for each data type of the dataset Solar, for year 2012. Notice that there are no gaps in the data. 10

2.8 Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in W/m^2), for each data type of the dataset Solar, for year 2013. The gaps are ignored when calculating the median, mean and standard deviation. 10

2.9 Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in W/m^2), for each data type of the dataset Solar, for year 2014. The gaps are ignored when calculating the median, mean and standard deviation. 10

2.10 Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in their respective units of measurement), for each data type of the dataset Hail. Notice that there are no gaps in this dataset. 11

2.11 Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in coordinate degrees), for each data type of the dataset Tornado. Notice that there are no gaps in this dataset. 11

2.12 Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in their respective units of measurement), for each data type of the dataset Wind. Notice that there are no gaps in this dataset. 12

2.13 Datasets overview. 12

2.14 Data types overview. 13

3.1 Characteristics of the evaluated algorithm variants. 16

4.1 Range of values for the RD between the masking and non-masking variants of each algorithm (last column); we highlight the maximum (red) and minimum (blue) values taken by the RD. The results are aggregated by dataset. The second column indicates the characteristic of each dataset, in terms of the number of gaps. The third column shows the number of cases in which the masking variant outperforms the non-masking variant of a coding algorithm, and its percentage among the total pairs of CAIs compared for a dataset. 72

4.2 RD between the OWS and LOWS variants of each CAI. The results are aggregated by algorithm and the range to which the RD belongs. 76

4.3 Compression performance of the best evaluated coding algorithm, for various error values on each data type of each dataset. Each row contains information relative to certain data type. For each error parameter value, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the best window size for the best algorithm (the one that achieves the minimum CR), which is identified by a certain cell color described in the legend above the table. 79

4.4 maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^*$ and error parameter $e \in E$. For each e, the cell corresponding to the minmaxRD(a) value is highlighted. 80

4.5 Compression performance of the best evaluated coding algorithm, for various error values on each data type of each dataset, including the results obtained by gzip. Each row contains information relative to certain data type. Each row contains information relative to certain data type. For each error parameter value, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the best window size for the best algorithm (the one that achieves the minimum CR), which is identified by a certain cell color described in the legend above the table. Algorithm gzip doesn't have a window size parameter, so the cell is left blank in these cases. 82

4.6 maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^* \cup \{\text{gzip}\}$ and error parameter $e \in E$. For each e, the cell corresponding to the minmaxRD(a) value is highlighted. 83

4.7 Lossless compression performance of algorithm variants PCA$_{NM}$ and APCA$_{NM}$, for the timestamp column of each dataset. For each variant, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the best window size (the window size that achieves the minimum CR). For each dataset, the cells corresponding to the algorithm that obtains the best result are colored, and the last column shows the RD between the corresponding CAIs, as defined in (4.5). 84

Chapter 1

Introduction

In the last 20 years, we have witnessed a staggering development of mobile communications and the Internet. Both factors have contributed to an accelerated expansion of the digital universe. Researchers estimate that the amount of digital data, created and replicated worldwide, more than doubles every three years: it went from 4.4 ZB[1] in 2013, to 33 ZB in 2018, and it is expected to reach 175 ZB by the year 2025 [1, 2]. In this context, research on data compression has become more relevant than ever.

Data compression techniques allow for the reduction of the number of bits needed to represent digital data. There are two types of compression algorithms: *lossless* and *lossy*. Lossless compression algorithms allow the original data to be perfectly reconstructed from the compressed data. On the other hand, lossy compression algorithms only allow to reconstruct an approximation of the original data, though they usually obtain better compression rates (i.e. the compressed data is represented using even a smaller number of bits).

Data compression is ubiquitous, with applications in industry as well as various branches of scientific research. It is worth mentioning a few examples, just to paint a picture of its broad scope:

- All the multimedia data that is sent, from the servers of both music (e.g. Spotify, Apple Music) and video (e.g. Youtube, Netflix) streaming services, to an end user device, is compressed to optimize the bandwidth usage [3, 4]. Something similar occurs with digital television, where signals are encoded in the source, and decoded in the receiver [5].
- Voice and video calls made on *VoIP (Voice over IP)* software, such as Zoom [6] or Skype [7], always require the implementation of compression techniques on every end user device, without which it would not be possible to maintain a real-time conversation with proper sound and video quality.
- Digital cameras use different compression algorithms to reduce the size of the image files, which allows to decrease the storage and transmission costs. In general, the end user is able to select between a lossy compression algorithm (JPEG being the most popular one), and a lossless one (TIFF is one of the industry standards, but leading brands have their own algorithms) [8, 9].

[1] A *zettabyte (ZB)* is a unit of measurement of the size of digital information on a computer or other electronic device, which is equivalent to 10^{12} GB.

- There is a great variety of general-purpose compression tools, which allow to losslessly compress any type of file. Among the most popular ones are gzip [10] and WinRAR [11], which also allow to encrypt and split the compressed files.
- In the medicine field, some tests (e.g. EEG, ECG) require for a patient to wear a monitoring device that measures clinical diagnosis data and wirelessly broadcasts it to a remote storage device, for an extended period of time. In such cases, compression algorithms are used to compress the transmitted data, which reduces the amount of energy consumed by the monitoring device, thus extending battery life [12, 13].
- NASA's space missions involve transmitting information back to Earth from space probes in far away places, which requires a great amount of energy. In past missions to Mars and Pluto, using onboard compression algorithms allowed to save resources, such as energy, space, and money, which is crucial for making space exploration viable [14, 15].
- A *Wireless Sensor Network (WSN)* consists of spatially distributed sensors that communicate wirelessly to collect data about the surrounding environment. They are used in a wide variety of environmental, health, industrial, and military applications [16, 17]. Since the sensors are often placed in remote locations, it is important that they use energy-efficient compression algorithms and communication protocols, so that their power consumption is optimized.

Even though some kind of data compression technique is involved in each one of the examples presented above, which specific compression algorithm is used in each case depends on a variety of factors [18], such as the characteristics of the data, the desired compression rate, whether the data is going to be archived or transmitted[2], and whether the energy resources are critical or not[3].

In this work, we focus on the compression of multichannel signals with irregular sampling rates and with data gaps. There is a wide range of research on compression algorithms for multichannel signals with *regular sampling rates* (see, for example, [19, 20] for images, [21, 22] for audio, [23, 24] for video, and [25, 26] for biomedical signals). However, real-world datasets sometimes consist of multichannel signals with *irregular sampling rates.* This occurs frequently in datasets gathered by WSNs[4], since different groups of sensors may be out of sync, and some might even malfunction. It is also common that errors arise when acquiring, transmitting or storing the data, causing a *data gap.* Nevertheless, state-of-the-art algorithms designed for sensor data compression reported in the literature [27, 28] assume, in general, that the signals have regular sampling rates and that there are no gaps in the data. Thus, in this thesis we design and implement a number of variants of state-of-the-art algorithms, adapt them so they are able to encode multichannel signals with irregular sampling rates and data gaps, and then evaluate their compression performance experimentally.

To assess the performance of our implemented algorithm variants, we consider eight different real-world datasets: dataset IRKIS [29] is gathered by multiple soil moisture measuring sensors installed along the Dischma valley, in the municipality of Davos, Switzerland; datasets ElNino [30], SST and ADCP [31] consist of various oceanographic and surface meteorological readings, obtained by sensors placed in buoys and moorings in the Pacific Ocean; dataset Solar [32] consists of solar radiation measurements in the city of Miami, Florida, US; and datasets Hail, Tornado

[2] In general, lossless compression is recommended for archival purposes, while lossy compression is suggested to optimize the bandwidth usage when transmitting data.

[3] In general, compression algorithms executed in a battery-run device tend to have low computational complexity, so that a small amount of energy is consumed during the compression process.

[4] In the context of WSNs, we consider that each sensor, which records data corresponding to a single signal, represents a different channel.

and Wind [33] consist of data related to thunderstorms and tornadoes along US territory. Each experimental dataset consists of multichannel signals with one or both of the characteristics we are interested in, namely, irregular sampling rate and data gaps. The number of gaps, as well as other characteristics (e.g. whether they are smooth or rough signals, the number of outliers, periodicity), varies among different signals, which allows us to analyze the performance of the evaluated algorithms under different circumstances. The datasets come from multiple sources [29–33], each using a different data representation format. Thus, as a first step, we transformed the data into a uniform format, which we define ourselves, and can be easily adapted to represent different kinds of datasets. The details of this data format, as well as the description of the experimental datasets, laying out the source, characteristics and relevant statistics of every signal involved, are presented in **Chapter 2**.

Both the original and our implemented variants are *near-lossless* compression algorithms. This type of algorithms guarantee a bounded per-sample absolute error between the decompressed and the original signals. The error threshold can be specified via a parameter, denoted ϵ. When ϵ is equal to zero, the compression is lossless, i.e., the decompressed and the original signals are identical. Additionally, the original algorithms and our proposed variants follow a model-based compression approach that compresses signals by exploiting correlation between signal samples taken at close times (*temporal correlation*) and, in some cases, between samples from different channels (*spatial correlation*). In addition to efficient compression performance, they offer some data processing features, like inferring uncertain sensor readings, detecting outliers, indexing, etc. [27]. The model-based techniques are classified into different categories, depending on the type of model: *constant models* approximate signals by piecewise constant functions, *linear models* use linear functions, and *correlation models* simultaneously encode multiple signals exploiting temporal and spatial correlation. There also exist *nonlinear models*, which approximate signals by nonlinear functions, but known algorithms that follow this technique do not support near-lossless compression, and generally do not yield the best compression results on WSN data [27, 28]. In total, we implement variants for eight different compression algorithms: *Piecewise Constant Approximation (PCA)* [34] and *Adaptive Piecewise Constant Approximation (APCA)* [35], which are constant model algorithms; *PieceWise Linear Histogram (PWLH)* [36], *PWLHInt* (see Subsection 3.8.2), *Critical Aperture (CA)* [37], *Slide Filter (SF)* [38], and *Fractal Resampling (FR)* [39], which are linear model algorithms; and *Grouping and AMPlitude Scaling (GAMPS)* [40], which is a correlation model algorithm. The variants implemented for our evaluation are presented in **Chapter 3**, including a description of their parameters, details of their coding and decoding routines, and examples that show the encoding process step by step.

For most algorithms we design and implement two variants, *masking (M)* and *non-masking (NM)*, which differ in the encoding of the gaps in the data. Variant M of an algorithm first encodes the position of all the gaps, and then proceeds to encode the data values separately, while variant NM encodes the gaps and the data values together. The proposed strategy to compress the gaps in variant M, which is presented in Section 3.3, uses arithmetic coding [41, 42] combined with a Krichevsky-Trofimov probability assignment [43] over a Markov model. We point out that, with both variants, the gaps in a decompressed signal match the gaps in the original signal exactly, regardless of the value of the error threshold parameter (ϵ). Efficient compression of gap information, especially in variant M, is an original contribution of this thesis.

In **Chapter 4** we present and analyze a series of experimental results, with the aim of evaluating and comparing the various tested algorithm variants in practical scenarios. All of our experiments involve the compression of data obtained from the real-world datasets presented in Chapter 2. We assess the compression performance of an algorithm variant through the *compression ratio (CR)*, which is calculated by the formula

$$\mathrm{CR} = \frac{\text{size of compressed data}}{\text{size of original data}}. \tag{1.1}$$

Thus, smaller values of CR correspond to a better performance. To compare the compression performance between two algorithm variants a_1 and a_2, we consider the *relative difference (RD)* metric, which is calculated by the formula

$$\mathrm{RD}(a_1, a_2) = 100 \times \frac{\text{size of data compressed with } a_2 - \text{size of data compressed with } a_1}{\text{size of data compressed with } a_2}. \quad (1.2)$$

Therefore, a_1 has a better performance than a_2 if $\mathrm{RD}(a_1, a_2)$ is positive[5].

In Section 4.2, for each algorithm a that supports variants M and NM[6], we compare the respective compression performance of both, a_M and a_{NM}. The results show that on datasets with few or no gaps the performance of both variants is roughly the same, i.e. $\mathrm{RD}(a_M, a_{NM})$ is always close to zero, ranging between -0.29 and 1.76%. On the other hand, on datasets with many gaps variant M always performs better, in some cases with a significant difference, with $\mathrm{RD}(a_M, a_{NM})$ ranging between 2.44 and 50.78%. These experimental results suggest that variant M is more robust and performs better in general.

Every original algorithm (and its respective variants) depends on a window size parameter, denoted w, which defines the size of the windows into which the data are partitioned for encoding. In algorithm PCA, it defines a *fixed window size*, while in the rest of the algorithms it defines a *maximum window size*. In Section 4.3 we analyze the sensitivity of variant M of each algorithm[7] to this parameter. For each dataset, we compress each data file, and compare the results obtained when using a window size optimized for said specific file, against the results obtained when using a window size optimized for the whole dataset. The results indicate that the difference in compression performance is generally rather small, with the RD being less than or equal to 2% in 97.8% of the experimental cases. Therefore, we could fix the window size parameter in advance, for example by optimizing over a training set, without significantly compromising the overall performance of the compression algorithm variant. This is relevant, since obtaining the optimal window size for a specific data file is, in general, computationally expensive.

The last part of our experimental analysis, which is presented in Section 4.4, consists of comparing the compression performance of our adapted algorithm variants, with each other, and with the general-purpose lossless compression algorithm gzip [10]. Taking into account the discussion in the previous two paragraphs, for this analysis we only consider variant M of each algorithm, and in every case we use the optimal window size for each whole dataset. Our experimental results indicate that none of the algorithm variants obtains the best compression performance in every scenario. This means that the optimal selection of a variant depends on the characteristics of the data to be compressed, and the error threshold (ϵ) that is allowed. In some circumstances, even a general-purpose compression algorithm such as gzip outperforms the specific variants. Nevertheless, we extract some general conclusions from our analysis. For large error thresholds, variant APCA_M achieves the best compression results, obtaining the minimum CR in 76 out of 84 experimental cases (90.5%). On the other hand, algorithm gzip and variant PCA_M are preferred for lower thresholds scenarios, since they obtain the minimum CR in 18 and 17 out of 42 experimental cases (42.9% and 40.5%), respectively.

Finally, in **Chapter 5** we sum up the conclusions of our thesis, and we propose some ideas to carry out as future work.

[5] More formal definitions of the compression performance metrics CR and RD will be given in Section 4.1.

[6] Algorithms SF and FR only support variant M, while the rest of the algorithms support variants M and NM.

[7] Due to the results obtained in Section 4.2, our experiments in the subsequent sections in Chapter 4 focus on studying the compression performance of variant M of the algorithms.

Chapter 2

Datasets

In this chapter we introduce the datasets used in our experiments. Every dataset consists of signals with one or both of the characteristics we are interested in, namely, irregular sampling rate and data gaps. In Chapter 4, we present our experimental results, which make use of these datasets to analyze the compression performance of our implemented algorithm variants, which are presented in Chapter 3.

The datasets come from multiple sources [29–33], each using a different data representation format. We transformed the data into a consistent, uniform format, which can be easily adapted to represent different kinds of datasets. In **Section 2.1** we describe this format, including an example file that illustrates how the data is represented. In **sections 2.2** to **2.9** we present each of the eight real-world datasets, namely, IRKIS [29], SST, ADCP [31], ElNino [30], Solar [32], Hail, Tornado, and Wind [33], laying out the source, characteristics and relevant statistics of every signal involved. Finally, in **Section 2.10** we summarize the information presented in this chapter, specifying the number of files, the number of gaps, and the data types of each dataset, as well as the measuring unit, scale, and range of each data type.

2.1 Data Format

The data from each dataset [29–33] was transformed into a consistent format, which can be adapted to represent different types of datasets. In Table 2.1 we present an example of a *comma-separated values (CSV)* file with this format. Although a CSV file is simply a delimited text file that uses commas to separate values, we display its contents in the form of a table for legibility. The first three rows have general data, namely a dataset name, in this example ElNino, the total number of data rows, and the timestamp value for the first sample. The remaining rows are divided into two sections: *metadata* and *data*. The data section consists of a row with the labels of each data column, followed by the data rows representing the actual signal data, while the metadata section has additional information indicating the measuring unit, scale, and range, of the data displayed in the respective data columns.

DATASET:	ElNino				
DATA ROWS:	7				
FIRST TIMESTAMP:	1980-03-07 00:00:00				
METADATA:					
COLUMNS	UNIT	SCALE	MINIMUM	MAXIMUM	
Time Delta	hours	1	0	131071	
Lat	coord. degrees	100	-1000	1000	
Long	coord. degrees	100	-18000	18000	
Mer. Wind	m/s	10	-150	150	
Air Temp.	°C	100	0	4000	
Sea Temp.	°C	100	0	4000	
DATA:					
Time Delta	Lat	Long	Mer. Wind	Air Temp.	Sea Temp.
0	-2	-10946	7	2614	2624
24	-2	-10946	11	N	N
24	-2	-10946	22	N	N
48	-1	-10946	19	N	2431
24	-2	-10946	15	2557	2319
48	-2	-10946	3	2472	2364
24	-2	-10946	-1	N	2434

Table 2.1: Example of a dataset CSV file with the format we defined.

The values in the first data column, which we label “Time Delta”, represent the timestamps associated with the data from the rest of the columns. We often refer to this column as the *timestamp column*. It is always the first column, and it is present in every CSV file. In practice, this timestamp may represent the time at which the data was read, transmitted, stored, etc. Each timestamp is represented as an integer value, which is an increment with respect to the previous timestamp, measured in the unit specified in the metadata section. For the first timestamp, the increment is taken with respect to the “First Timestamp” entry in the third row, and its value is always zero. Thus, in the example presented in Table 2.1, the first four timestamps are interpreted as “1980-03-07 00:00:00”, “1980-03-08 00:00:00”, “1980-03-09 00:00:00”, and “1980-03-11 00:00:00”. Notice that the difference between subsequent timestamps is not constant, varying between 24 and 48 hours, which means that the signals in this dataset have an irregular sampling rate (recall that this is one of the data characteristics we are interested in).

Besides the timestamp column, the CSV presented in Table 2.1 consists of five additional data columns, each representing a different data type. The first two columns display the latitude and longitude coordinates, respectively, of a buoy floating in the ocean, while the last three columns display readings of various physical magnitudes (i.e. wind velocity, air and sea temperatures), made by sensors on the buoy. Additional information on dataset ElNino [30] is presented in Section 2.5. To keep the data representation consistent among different datasets, the readings are always transformed into the integer domain. In general, the information shown in the metadata section is determined by the range and accuracy of the sampling instrument used for acquiring and storing the data, and in some cases by the nature of the data (e.g. the range of the latitude and longitude coordinates depends on the area in which the data are recorded). For example, in the dataset file presented in Table 2.1, the sampling instrument used for measuring the air and sea temperatures has a precision of two significant digits, and its range is between 0 and 40°C. Thus, the three sample values shown in the air temperature column, correspond to 26.14, 25.57, and 24.72 °C, respectively (these values are obtained after dividing each displayed integer value by the corresponding scale).

In every dataset, the timestamp column consists of integer values, but, in general, the rest of the columns admit both integer values and the character “N”. An integer value represents an actual data sample, while character “N” represents a gap in the data. In practice, this data gap may occur when there’s an error acquiring, transmitting or storing the data. In the example in Table 2.1, there are some gaps in the last two data columns (recall that this is another of the data characteristics we are interested in).

2.2 Dataset IRKIS

Dataset IRKIS [29] consists of soil moisture measurements from stations installed along the Dischma valley, in the municipality of Davos, Switzerland, in the period between October 2010 and October 2013. This information is particularly useful to researchers who study the role of soil moisture in relation to the snowpack runoff and catchment discharge in high alpine terrain [44].

The dataset corresponds to seven stations, labeled 1202, 1203, 1204, 1205, 222, 333, and SLF2, respectively, which can be located in the map presented in [29]. For each station, the dataset contains measures of soil moisture data at different depths below the surface, namely 10, 30, 50, 80 and 120 cm. For each depth, there are moisture measures coming from two sensors, labeled A and B in the dataset files. Thus, for each station, there are 10 different data samples stored for each timestamp. The sensors measure the *Volumetric Water Content (VWC)*, which is equal to the ratio of water volume to soil volume. Therefore, higher VWC values indicate a more moist soil. The scale for the VWC columns is 10^3.

In Table 2.2 we present some statistics of dataset IRKIS. The data from each station is stored in a separate CSV file, and every row in the table contains statistics for one such file. The first three columns show the total number of rows, columns and entries (i.e. number of rows times number of columns) in the dataset, respectively. The fourth column specifies the number of gaps, and the percentage of gaps relative to the total number of entries. The last five columns show the minimum, maximum, median, mean, and standard deviation, of the sample values (dimensionless).

Station	Rows	Cols	Entries	Gaps (%)	Min	Max	Mdn	Mean	SD
1202	26,305	10	263,050	125,190 (47.6)	240	541	428	417.1	48.8
1203	26,305	10	263,050	200,051 (76.1)	165	385	249	257.7	40.1
1204	26,305	10	263,050	178,657 (67.9)	218	464	298	313.3	70.0
1205	26,305	10	263,050	224,287 (85.3)	272	600	315	342.1	63.9
222	26,304	10	263,040	56,007 (21.3)	128	562	426	384.2	119.7
333	26,088	10	260,880	37,520 (14.4)	38	451	327	291.5	81.1
SLF2	26,305	10	263,050	43,049 (16.4)	215	580	352	360.9	65.1

Table 2.2: Number of rows, columns, entries and gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (dimensionless), for each file of the dataset IRKIS. The gaps are ignored when calculating the median, mean and standard deviation.

2.3 Dataset SST

Dataset SST [31] consists of *Sea Surface Temperature (SST)* measurements from buoys floating in the Pacific Ocean. This dataset is collected by the Tropical Atmosphere Ocean (TAO) project, which was established in 1985 to study annual climate variations near the equator [45].

The dataset consists of readings from 55 buoys, which can be located in the map presented in [31]. For each timestamp, the dataset contains the SST measurement taken in each buoy. Thus, a total of 55 data samples are stored for each timestamp. The temperature is measured in degrees Celsius (°C), and its scale is 10^3.

In Table 2.3 we present some statistics of dataset SST. The data for each month is stored in a separate CSV file, and every row in the table contains statistics for one such file. The first three columns show the total number of rows, columns and entries (i.e. number of rows times number of columns) in the dataset, respectively. The fourth column specifies the number of gaps, and the percentage of gaps relative to the total number of entries. The last five columns show the minimum, maximum, median, mean, and standard deviation, of the sample values (in °C).

Month	Rows	Cols	Entries	Gaps (%)	Min	Max	Mdn	Mean	SD
01-2017	4,392	55	241,560	92,866 (38.4)	3	32,367	27,221	27,326.9	2,406.0
02-2017	3,984	55	219,120	84,292 (38.5)	3	31,277	27,477	27,677.3	1,953.1
03-2017	4,362	55	239,910	99,541 (41.5)	3	31,809	28,070	28,050.9	1,760.4

Table 2.3: Number of rows, columns, entries and gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in °C), for each file of the dataset SST. The gaps are ignored when calculating the median, mean and standard deviation.

2.4 Dataset ADCP

Dataset ADCP [31] consists of water current velocity measurements from moorings in the Pacific Ocean. This dataset is collected by the Tropical Atmosphere Ocean (TAO) project, which was established in 1985 to study annual climate variations near the equator [45].

The dataset consists of readings from 3 moorings, which can be located in the map presented in [31]. In each mooring, readings are made by a total of 63 *acoustic Doppler current profilers (ADCPs)*, placed at different depths below the ocean. Each ADCP measures the *eastward (UCUR)*, *northward (VCUR)*, and *upward (WCUR)* components of the water current velocity. Therefore, there are 567 ($3 \times 63 \times 3$) data samples stored for each timestamp. The velocity is measured in m/s, and its scale is 10^3.

In Table 2.4 we present some statistics of dataset ADCP. The data for each month is stored in a separate CSV file, and every row in the table contains statistics for one such file. The first three columns show the total number of rows, columns and entries (i.e. number of rows times number of columns) in the dataset, respectively. The fourth column specifies the number of gaps, and the percentage of gaps relative to the total number of entries. The last five columns show the minimum, maximum, median, mean, and standard deviation, of the sample values (in m/s).

Month	**Rows**	**Cols**	**Entries**	**Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
01-2015	744	567	421,848	134,544 (31.9)	-805	1,394	6	113.7	273.1
02-2015	672	567	381,024	125,151 (32.8)	-761	1,822	7	154.5	318.3
03-2015	744	567	421,848	133,110 (31.6)	-870	2,094	13	185.5	367.4

Table 2.4: Number of rows, columns, entries and gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in m/s), for each file of the dataset ADCP. The gaps are ignored when calculating the median, mean and standard deviation.

2.5 Dataset ElNino

Dataset ElNino [30] consists of various oceanographic and surface meteorological measurements from buoys floating in the Pacific Ocean between 1980 and 1998. This dataset is collected by the Tropical Atmosphere Ocean (TAO) project, which was established in 1985 to study annual climate variations near the equator [45].

The dataset consists of readings from 78 buoys, which move around in the Pacific Ocean, near the equator. For each timestamp, the position of each buoy is specified, in terms of its latitude and longitude (in coordinate degrees), and in each buoy the following measurements are taken: speed of the zonal and meridional winds (m/s), relative humidity (%), and air temperature and sea temperature (°C). Therefore, a total of 546 (78 × 7) data samples are stored for each timestamp. The scale for the latitude, longitude, and air and sea temperatures columns is 10^2, while the scale for the speed of the zonal and meridional winds, and relative humidity columns is 10.

In Table 2.5 we present some statistics of dataset ElNino. The dataset consists of a single file, and each row in the table contains statistics for a different data type. The first column specifies the number of gaps, and the percentage of gaps over the total number of entries. The file has 6,371 rows, so there is a total of 496,938 ($78 \times 6,371$) entries of each data type. The rest of the columns show the minimum, maximum, median, mean, and standard deviation, of the sample values (in their respective units of measurement).

Data Type	**Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
Lat (coord. degrees)	318,858 (64.2)	-881	905	1	47.3	458.2
Long (coord. degrees)	318,858 (64.2)	-18,000	17,108	-11,126	-5,402.5	13,536.4
Zon. Wind (m/s)	344,021 (69.2)	-124	143	-40	-33.0	33.7
Mer. Wind (m/s)	344,020 (69.2)	-116	130	3	2.5	30.0
Humidity (%)	384,619 (77.4)	454	999	812	812.4	53.1
Air Temp. (°C)	337,095 (67.8)	1,705	3,166	2,734	2,688.8	181.6
Sea Temp. (°C)	335,865 (67.6)	1,735	3,126	2,829	2,771.5	205.7

Table 2.5: Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in their respective units of measurement), for each data type of the dataset ElNino. The gaps are ignored when calculating the median, mean and standard deviation.

2.6 Dataset Solar

Dataset Solar [32] consists of solar radiation measurements from sensors in the city of Miami, Florida, US. This dataset is collected by SolarAnywhere, an online tool used for monitoring solar radiation around the world.

The dataset consists of readings from 12 sensors, where each sensor measures three different types of solar irradiance, namely, *Global Horizontal Irradiance (GHI)*, *Direct Normal Irradiance (DNI)*, and *Diffuse Horizontal Irradiance (DHI)*. Thus, a total of 36 data samples are stored for each timestamp. The solar irradiance is measured in W/m^2, and its scale is 1.

In tables 2.6, 2.7, 2.8, and 2.9, we present some statistics of dataset Solar for years 2011, 2012, 2013, and 2014, respectively. The data for each year is stored in a different CSV file. Each file has 8,759 rows, with a total of 105,108 ($12 \times 8,759$) entries of each data type. Each row in the tables contains statistics for a different data type. The first column specifies the number of gaps, and the percentage of gaps over the total number of entries. The rest of the columns show the minimum, maximum, median, mean, and standard deviation, of the sample values (in W/m^2).

Data Type	**Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
GHI	23 (0.02)	0	1,005	11	187.6	274.7
DNI	23 (0.02)	0	959	3	183.5	278.2
DHI	23 (0.02)	0	763	8	70.5	96.8

Table 2.6: Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in W/m^2), for each data type of the dataset Solar, for year 2011. The gaps are ignored when calculating the median, mean and standard deviation.

Data Type	**Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
GHI	0 (0)	0	1,004	9	186.0	269.7
DNI	0 (0)	0	958	3	178.3	272.8
DHI	0 (0)	0	485	7	72.7	100.4

Table 2.7: Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in W/m^2), for each data type of the dataset Solar, for year 2012. Notice that there are no gaps in the data.

Data Type	**Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
GHI	156 (0.15)	0	1,004	9	183.7	267.8
DNI	156 (0.15)	0	954	3	168.9	261.8
DHI	156 (0.15)	0	483	7	75.8	103.2

Table 2.8: Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in W/m^2), for each data type of the dataset Solar, for year 2013. The gaps are ignored when calculating the median, mean and standard deviation.

Data Type	**Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
GHI	204 (0.19)	0	1,006	9	189.5	274.8
DNI	204 (0.19)	0	959	4	181.0	274.6
DHI	204 (0.19)	0	473	7	73.1	100.1

Table 2.9: Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in W/m^2), for each data type of the dataset Solar, for year 2014. The gaps are ignored when calculating the median, mean and standard deviation.

2.7 Dataset Hail

Dataset Hail [33] consists of hail size measurements from several stations around the US, taken between 2015 and 2017. This dataset is collected by the Storm Prediction Center (SPC), a US government agency established in 1995, whose goal is to forecast the risk of severe thunderstorms and tornadoes along US territory.

The dataset consists of hail diameter readings from several stations distributed around the US. For each timestamp, besides the hail diameter measurement, the position of the station where the measurement is taken is specified, in terms of its latitude and longitude (in coordinate degrees). The scale of the latitude and longitude is 100. The hail diameter is measured in 1/100 of an inch, and its scale is 1.

In Table 2.10 we present some statistics of dataset Hail. The dataset consists of a single file, and each row in the table contains statistics for a different data type. The first column specifies the number of gaps, and the percentage of gaps over the total number of entries. The file has 17,059 rows, so there is the same number of entries of each data type. The rest of the columns show the minimum, maximum, median, mean, and standard deviation, of the sample values (in their respective units of measurement).

Data Type	Gaps (%)	Min	Max	Mdn	Mean	SD
Lat (coord. degrees)	0 (0)	2,570	4,932	3,845	3,854.7	475.7
Long (coord. degrees)	0 (0)	-12,442	-6,783	-9,676	-9,487.4	841.4
Size (1/100 inch)	0 (0)	100	600	100	136.2	51.8

Table 2.10: Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in their respective units of measurement), for each data type of the dataset Hail. Notice that there are no gaps in this dataset.

2.8 Dataset Tornado

Dataset Tornado [33] consists of records with the location of tornadoes around the US, from between 2015 and 2017. For each timestamp, a location hit by a tornado is specified via its latitude and longitude (in coordinate degrees). This dataset is collected by the Storm Prediction Center (SPC), a US government agency established in 1995, whose goal is to forecast the risk of severe thunderstorms and tornadoes along US territory. The scale of the latitude and longitude is 100.

In Table 2.11 we present some statistics of dataset Tornado. The dataset consists of a single file, and each row in the table contains statistics for a different data type. The first column specifies the number of gaps, and the percentage of gaps over the total number of entries. The file has 3,841 rows, so there is the same number of entries of each data type. The rest of the columns show the minimum, maximum, median, mean, and standard deviation, of the sample values (in coordinate degrees).

Data Type	Gaps (%)	Min	Max	Mdn	Mean	SD
Lat	0 (0)	2,456	4,891	3,688	3,693.9	491.1
Long	0 (0)	-12,397	-6,822	-9,313	-9,260.0	787.9

Table 2.11: Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in coordinate degrees), for each data type of the dataset Tornado. Notice that there are no gaps in this dataset.

2.9 Dataset Wind

Dataset Wind [33] consists of wind velocity measurements from several stations around the US, taken between 2015 and 2017. This dataset is collected by the Storm Prediction Center (SPC), a US government agency established in 1995, whose goal is to forecast the risk of severe thunderstorms and tornadoes along US territory.

The dataset consists of wind velocity readings from stations distributed around the US. For each timestamp, besides the wind velocity measurement, the position of the station where the measurement is taken is specified, in terms of its latitude and longitude (in coordinate degrees). The scale of the latitude and longitude is 100. The wind velocity is measured in mph, and its scale is 10.

In Table 2.12 we present some statistics of dataset Wind. The dataset consists of a single file, and each row in the table contains statistics for a different data type. The first column specifies the number of gaps, and the percentage of gaps over the total number of entries. The file has 40,260 rows, so there is the same number of entries of each data type. The rest of the columns show the minimum, maximum, median, mean, and standard deviation, of the sample values (in their respective units of measurement).

Data Type	**Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
Lat (coord. degrees)	0 (0)	0	4,899	3,776	3,778.4	460.8
Long (coord. degrees)	0 (0)	-12,432	-8	-8,732	-8,870.1	968.4
Speed (mph)	0 (0)	0	2,359	1,810	1,350.1	875.2

Table 2.12: Number of gaps (total and percentual), and minimum, maximum, median, and standard deviation, of the sample values (in their respective units of measurement), for each data type of the dataset Wind. Notice that there are no gaps in this dataset.

2.10 Summary

In Table 2.13 we summarize information of the eight datasets presented in the previous sections of this chapter. The second column indicates, qualitatively, the characteristic of each dataset, in terms of the number of gaps. The third column shows the number of CSV files corresponding to each dataset. The last two columns show the number of data types in each dataset, and their names, respectively. In many files there are multiple data columns with the same data type.

Dataset	**Characteristic**	**#Files**	**#Types**	**Data Types**
IRKIS [29]	Many gaps	7	1	VWC
SST [31]	Many gaps	3	1	SST
ADCP [31]	Many gaps	3	1	Velocity
ElNino [30]	Many gaps	1	7	Lat, Long, Zon. Wind, Mer. Wind, Humidity, Air Temp., Sea Temp.
Solar [32]	Few gaps	4	3	GHI, DNI, DHI
Hail [33]	No gaps	1	3	Lat, Long, Size
Tornado [33]	No gaps	1	2	Lat, Long
Wind [33]	No gaps	1	3	Lat, Long, Speed

Table 2.13: Datasets overview.

In Table 2.14 we show the measuring unit, scale, and range of each data type. Recall that this information is included in the CSV file for each dataset. We also show, in the last column, the value of the column-dependent parameter B_c, obtained for the data columns of each data type. It represents the number of bits required to represent either an integer sample value or a data gap indicator in the respective data column. We will formally define and discuss B_c in Section 3.2. We point out that data types with the same name can have different measuring unit, scale, and/or range, among different datasets.

Dataset	**Data Type**	**Unit**	**Scale**	**Minimum**	**Maximum**	B_c **(bits)**
IRKIS	Time Delta	minutes	1	0	131,071	17
	VWC	dimensionless	1,000	0	600	10
SST	Time Delta	seconds	1	0	131,071	17
	SST	°C	1,000	0	40,000	16
ADCP	Time Delta	minutes	1	0	131,071	17
	Velocity	m/s	1,000	-1,100	2,700	12
ElNino	Time Delta	hours	1	0	131,071	17
	Lat	coord. degrees	100	-1,000	1,000	11
	Long	coord. degrees	100	-18,000	18,000	16
	Zon. Wind	m/s	10	-150	150	9
	Mer. Wind	m/s	10	-150	150	9
	Humidity	%	10	0	1,000	10
	Air Temp.	°C	100	0	4,000	12
	Sea Temp.	°C	100	0	4,000	12
Solar	Time Delta	minutes	1	0	131,071	17
	GHI	W/m^2	1	0	1,020	10
	DNI	W/m^2	1	0	970	10
	DHI	W/m^2	1	0	800	10
Hail	Time Delta	minutes	1	0	131,071	17
	Lat	coord. degrees	100	2,500	5,000	12
	Long	coord. degrees	100	-12,500	-6,700	13
	Size	1/100 inch	1	0	700	10
Tornado	Time Delta	minutes	1	0	131,071	17
	Lat	coord. degrees	100	2,400	5,000	12
	Long	coord. degrees	100	-12,500	-6,800	13
Wind	Time Delta	minutes	1	0	131,071	17
	Lat	coord. degrees	100	0	5,000	13
	Long	coord. degrees	100	-12,500	0	14
	Speed	mph	10	0	2,400	12

Table 2.14: Data types overview.

Chapter 3

Algorithms and Their Variants

In this chapter we present the algorithm variants that we have implemented and evaluated. In **Section 3.1** we give an overview of various state-of-the-art compression algorithms, including their parameters and our masking/non-masking variants. In **Section 3.2** we describe the general encoding scheme used for every algorithm variant, while in **Section 3.3** we propose a general strategy to compress gaps in the masking variants, using arithmetic coding combined with a Krichevsky–Trofimov probability assignment over a Markov model. In **Section 3.4** we present algorithm Base, which is a trivial algorithm that serves as a base ground for the compression performance comparison developed in Chapter 4. In **sections 3.5** to **3.12** we present the variants implemented for eight different algorithms, showing their coding and decoding routines, and describing an example that shows the encoding process step by step. Finally, in **Section 3.13** we present additional implementation details.

3.1 Introduction

We focus on compression algorithms for multichannel signals with irregular sampling rates and with data gaps. Signals with these characteristics are often observed in datasets gathered by WSNs, where different groups of sensors may be out of sync, some might malfunction, and errors may arise when acquiring, transmitting or storing the data. For example, this occurs in the experimental datasets presented in Chapter 2. However, state-of-the-art algorithms designed for sensor data compression reported in the literature [27, 28] assume, in general, that the signals have regular sampling rate and that there are no gaps in the data. Therefore, in this chapter we introduce a number of variants of state-of-the-art algorithms that we design and implement, which are adapted so they are able to encode multichannel signals with irregular sampling rates and data gaps. In Chapter 4 we evaluate their compression performance experimentally.

The original state-of-the-art algorithms considered, as well as our adapted variants, are *lossy* algorithms that support near-lossless compression, which is defined as follows.

Definition 3.1.1. *Near-lossless* compression guarantees a bounded per-sample absolute error between the decompressed and the original signals. This user-defined error threshold is specified through parameter ϵ.

When ϵ is zero, the compression is *lossless*: the decompressed and the original signals are identical. For most algorithms we implement two variants: a *masking (M)* variant, which first encodes the position of all the data gaps, and then proceeds to encode the data values separately, and a *non-masking (NM)* variant, which encodes the gaps and the data values together. In both variants, regardless of the value of parameter ϵ, the gaps in a decompressed signal match the gaps in the original signal exactly. In Section 4.2, we compare the compression performance of both variants, *M* and *NM*, for every evaluated algorithm that supports both.

Except from algorithm Base (a trivial lossless algorithm described in Section 3.4), all our algorithm variants parse the input data into windows, which are encoded in sequence (and, in most cases, independently). The size of these windows depends on a parameter denoted w. In both variants of algorithm PCA (presented in Section 3.5), parameter w defines a *fixed window size*, while for the rest of the algorithm variants it defines a *maximum window size*. More details are presented with the specific description of each algorithm, later in this chapter. In Section 4.3 we analyze the sensitivity of the algorithm variants to parameter w, in terms of their compression performance.

The original algorithms and our proposed variants follow a model-based compression approach: they compress signals by exploiting *temporal correlation* (i.e. correlation between signal samples taken at close times), and, in some cases, *spatial correlation* (i.e. correlation between samples from different channels). They offer an efficient compression performance, as well as some data processing features, such as inferring uncertain sensor readings, detecting outliers, indexing, etc. [27]. The model-based techniques are classified into different categories, depending on the type of model: *constant models* approximate signals by piecewise constant functions, *linear models* use linear functions, and *correlation models* simultaneously encode multiple signals exploiting temporal and spatial correlation. There also exist *nonlinear model* algorithms, such as *Chebyshev Transform* [46], *Discrete Cosine Transform (DCT)* and *Discrete Wavelet Transform (DWT)* [47], which approximate signals by nonlinear functions. Even though nonlinear model algorithms support lossless compression, they are mostly used for lossy compression (e.g. JPEG [48], which is based on DCT, is the most popular lossy image compression standard). However, they do not support near-lossless compression, and generally do not yield the best compression results on WSN data [27, 28]. In this chapter, we propose variants for eight different compression algorithms: PCA [34] and APCA [35] (constant model algorithms); PWLH [36], PWLHInt (see

Subsection 3.8.2), CA [37], SF [38], and FR [39] (linear model algorithms); and GAMPS [40] (correlation model algorithm). In Section 4.4 we compare the compression performance of our adapted algorithm variants, with each other, and with the general-purpose lossless compression algorithm gzip [10].

In Table 3.1 we outline some characteristics of the evaluated algorithm variants. For each algorithm, the second and third columns indicate whether it supports lossless and near-lossless compression, respectively, the fourth column shows its compression model, the fifth and sixth columns indicate if the masking (*M*) and non-masking (*NM*) variants apply, respectively, and the last column specifies if the algorithm depends on a window size parameter (w). Algorithm Base is a trivial lossless algorithm that is used as a base ground for comparing the performance of the remaining algorithms, all of which support both lossless and near-lossless encoding.

Algorithm	**Lossless**	**Near-lossless**	**Compression Model**	***M***	***NM***	w
Base	x		constant		x	
PCA [34]	x	x	constant	x	x	x
APCA [35]	x	x	constant	x	x	x
PWLH [36] and PWLHInt	x	x	linear	x	x	x
CA [37]	x	x	linear	x	x	x
SF [38]	x	x	linear	x		x
FR [39]	x	x	linear	x		x
GAMPS [40]	x	x	correlation	x	x	x

Table 3.1: Characteristics of the evaluated algorithm variants.

3.2 General Encoding Scheme

Figure 3.1 shows a general encoding scheme used for the evaluated algorithm variants.

```
input  : in: CSV data file to be encoded
         v: variant (M or NM)
         $\epsilon$: maximum error threshold
         w: window size
output: out: binary file with the encoding of in
Create output file out
Encode an algorithm identification key, variant key v, and parameter w (if applicable)
Encode the header of the input file
Encode the timestamp column using algorithm variant $\text{APCA}_{NM}$
if $v == M$ then
    Encode gap locations in each signal column of the input file (independently) into out
end
if we are using a constant or linear model algorithm then
    Encode each signal column of the input file (independently) into out, using the
      coding routine for variant v of a specific algorithm (i.e. Base, PCA, APCA, PWLH,
      PWLHInt, CA, SF, FR)
else if we are using a correlation model algorithm then
    Encode all the signal columns of the input file into out, using the coding routine for
      variant v of a specific algorithm (i.e. GAMPS)
end
return out
```

Figure 3.1: General encoding scheme for the evaluated algorithm variants.

Its inputs are a CSV data file in the format presented in Section 2.1, a key (v) that describes the algorithm variant (either M or NM), and the maximum error threshold (ϵ) and window size (w) parameters. The output is a binary file, which represents the input file encoded with a compression algorithm using the specified variant and parameters. Constant and linear model algorithms only exploit the temporal correlation in the data, thus they iterate through the data columns and encode each independently (Line 9). On the other hand, correlation models also exploit the spatial correlation, so the data columns are jointly encoded (Line 11). The decoding scheme is symmetric.

The timestamp column, which is comprised of integers, is the first column in every CSV data file, and it is also the first column to be encoded (Line 4). It is always encoded using algorithm variant APCA$_{NM}$. We delve further into the topic of timestamp compression in Subsection 4.4.2.

Next, we define a column-dependent parameter, B_c, which is used by the evaluated algorithms when encoding the timestamp, as well as the rest of the data columns.

Definition 3.2.1. The number of bits B_c required to encode a specific value in the i-th data column of a dataset file f is given by

$$B_c(f,i) = \lceil \log_2 \big(\max(f,i) - \min(f,i) + N_c(f,i) \big) \rceil, \tag{3.1}$$

where $\max(f,i)$ and $\min(f,i)$ are the maximum and minimum values allowed, respectively, for the i-th data column of f, and $N_c(z,d) \in \{0,1\}$ is a constant that accounts for an extra symbol needed to encode a gap, whose value is 1 if the data column admits gaps, and 0 otherwise.

Recall, from Section 2.1, that the timestamp column consists of integer values, but, in general, the rest of the columns admit both integer values and the character “N”, which represents a gap in the data. Thus, N_c is 0 for the timestamp column, and 1 for the rest of the data columns in each dataset. For the data columns in our experimental datasets, the minimum and maximum values obtained for parameter B_c are 9, for the zonal and meridional winds columns in dataset ElNino, and 17, for the timestamp columns in every dataset (recall this information from Table 2.14).

We point out that the encoder is able to calculate B_c for each data column in each dataset file: recall, from Section 2.1, that the maximum and minimum values allowed for each column are specified in the header of the dataset CSV file, making this information known to the encoder. Since the header of the file is encoded (Line 3 in Figure 3.1), the decoder can also calculate B_c in each case. In the sequel, in all the algorithm descriptions, we will assume that the value of B_c is known for the data column of interest in the algorithm.

In Figure 3.1, notice that when variant M of an algorithm is executed, the positions of the gaps in every data column are encoded, independently for each column, in Line 6; the details are explained in Section 3.3.

3.3 Encoding of Gaps in the Masking Variants

We recall that the masking variant of an algorithm starts by losslessly encoding the position of all the gaps in the data, independently for each data column (Line 6 in Figure 3.1). We describe the position of the gaps in a column by encoding a sequence of binary symbols, $x_1 ... x_n$, each symbol x_i indicating the presence (0) or absence (1) of a sample in the i-th timestamp of the column, in chronological order. To this end we use an arithmetic coder (AC) [41, 42]. Given a sequence of probability assignments, $p(\cdot|x_1 ... x_{i-1})$, for the symbol in position i given the past symbols $x_1 ... x_{i-1}$, $1 \leq i \leq n$, an AC generates a lossless encoding bit stream for $x_1 ... x_n$, of length $-\log P(x_1 ... x_n) + O(1)$, where $P(x_1 ... x_n) = \prod_{i=1}^{n} p(x_i|x_1 ... x_{i-1})$. This code length is optimal for this probability assignment, up to an additive constant [49].

For a sequence x of independent and identically distributed random binary symbols (with unknown probability distribution), the Krichevsky–Trofimov probability assignment [43], which we define next, yields an asymptotically optimal code length for the (unknown) probability distribution, in the sense that the worst case redundancy for such code is asymptotically minimized [50].

Definition 3.3.1. Given a string x over an alphabet $A = \{0, 1\}$, the *Krichevsky–Trofimov (KT) probability assignment* assigns the following probabilities for each symbol position $i, 1 \leq i \leq n$

$$p(0|x_1...x_{i-1}) = \frac{n_0 + 1/2}{i}, \qquad p(1|x_1...x_{i-1}) = \frac{n_1 + 1/2}{i}, \tag{3.2}$$

where n_0 and n_1 denote the number of occurrences of 0 and 1 in $x_1...x_{i-1}$, respectively.

Analyzing the experimental datasets presented in Chapter 2, we notice that, in general, the data gaps occur in bursts. Thus, it makes sense to consider a simple binary Markov model, such as the one defined next, sometimes referred to as the Gilbert-Elliot model [51, 52], which efficiently captures the burstiness of data gap occurrences.

The first-order Markov model has two states, S_0 and S_1, and we say that x_i occurs in state S_b iff the previous symbol, x_{i-1}, equals b. We arbitrarily let S_1 be the initial state (i.e. the state in which x_1 occurs). In Figure 3.2 we present a diagram for this Markov model. A KT probability assignment for a first-order Markov model is obtained by applying (3.2) separately for the subsequence of symbols that occur in states S_0 and S_1. This is implemented by maintaining two pairs of symbol occurrence counters, n_0, n_1, one pair for each state.

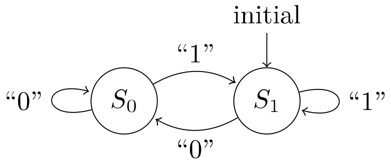

Figure 3.2: Markov model diagram.

3.4 Algorithm Base

Algorithm *Base* is a trivial lossless coding algorithm that yields a standard representation of the uncompressed data, which serves as a base ground for comparing the performance of the rest of the algorithms. It encodes every sample independently and using the same number of bits.

Algorithm Base follows the general schema presented in Figure 3.1, with a specific coding routine shown in Figure 3.3. This routine iterates through every entry of a column of a CSV data file. Since algorithm Base only supports variant *NM*, these entries can be either the character “N”, which represents a gap in the data, or an integer value representing an actual data sample. Every column entry is encoded independently using B_c bits. A special integer, *NO_DATA*, is reserved for encoding a gap. The decoding routine is symmetric to the coding routine.

```
input  : column: column of the CSV data file to be encoded
         out: binary file encoded with algorithm Base
foreach entry $y$ in column do
    if $y$ == “N” then
        value = NO_DATA
    else
        value = $y$
    end
    Encode value using $B_c$ bits
end
```

Figure 3.3: Coding routine for algorithm Base.

3.5 Algorithm PCA

Algorithm *Piecewise Constant Approximation (PCA)* [34] is a constant model algorithm that supports lossless and near-lossless compression. As its name suggests, it approximates a signal with a piecewise constant function. For PCA we define both variants, *M* and *NM*.

In Figure 3.4 we show the coding routine for variant *M*, in which all the column entries are integer values (the gaps are encoded separately). The column entries are parsed into consecutive non-overlapping windows of size w (Line 1), and each of these windows is encoded independently (lines 3-13).

```
input  : column: column of the CSV data file to be encoded
         out: binary file encoded with algorithm variant PCA$_M$
         $\epsilon$: maximum error threshold
         $w$: fixed window size
 1 Parse column into consecutive non-overlapping windows of size $w$, except possibly for
     the last window that may consist of fewer samples
 2 foreach window in the parsing, win, do
 3     Let $m$ and $M$ be the minimum and maximum sample values in win, respectively
 4     if $M - m \leq 2\epsilon$ then
 5         Output bit 0 to out
 6         mid_range = $\lfloor (m + M)/2 \rfloor$
 7         Encode mid_range using $B_c$ bits
 8     else
 9         Output bit 1 to out
10         foreach sample in win, value, do
11             Encode value using $B_c$ bits
12         end
13     end
14 end
```

FIGURE 3.4: Coding routine for algorithm variant PCA$_M$.

A window in the parsing, *win*, can be encoded in two different ways. If the difference between its maximum and minimum values is less than or equal to 2ϵ (i.e. the condition in Line 4 is satisfied), then bit 0 is output, and the value of *mid_range* for *win* is encoded (lines 5-7). On the other hand, if the condition in Line 4 evaluates to false, then bit 1 is output, and each of the values in *win* is encoded separately (lines 9-12). We point out that, in the former case, the absolute error between the encoded and the original values in *win* is guaranteed to be less than or equal to ϵ, as proven by the following Lemma.

Lemma 3.1. Let $m, M \in \mathbb{Z}$, $\epsilon \in \mathbb{N}$, such that $M - m \leq 2\epsilon$, let $mid\_range = \lfloor (m + M)/2 \rfloor$, and let $y \in \mathbb{R}$, such that $m \leq y \leq M$. Then $|y - mid\_range| \leq \epsilon$.

Proof. Assume there exists a $y' \in \mathbb{R}$, $m \leq y' \leq M$, such that $|y' - mid\_range| > \epsilon$.

We have $|y' - mid\_range| > \epsilon \implies 2|y' - mid\_range| > 2\epsilon \implies |2y' - (m + M)| > 2\epsilon$.

If $2y' \geq m + M$, then $|2y' - (m + M)| > 2\epsilon \implies 2y' - (m + M) > 2\epsilon$.
Since $M \geq y'$, then $2y' - (m + M) > 2\epsilon \implies 2M - (m + M) = M - m > 2\epsilon$, which contradicts the assumptions of the Lemma.

Otherwise, if $2y' < m + M$, then $|2y' - (m + M)| > 2\epsilon \implies 2y' - (m + M) < -2\epsilon$.
Since $m \leq y'$, then $2y' - (m + M) < -2\epsilon \implies 2m - (m + M) = m - M < -2\epsilon \implies M - m > 2\epsilon$, which, again, contradicts the assumptions of the Lemma. □

Observe that a window that satisfies the condition in Line 4 is encoded with fewer bits than a window that does not. Exactly $1 + B_c$ and $1 + w * B_c$ bits are required for encoding the window in each case, respectively.

The decoding routine for variant M is shown in Figure 3.5. It consists of a loop that repeats until every entry in the column has been decoded, which occurs when condition in Line 2 becomes false. Recall that the coding algorithm encodes the number of rows, as part of the header of the input file (Line 3 in Figure 3.1), so this information is known by the decoding routine (input *col_size*). Each iteration of the loop starts with the reading of a single bit from the input binary file (Line 4). If this bit is 0, then the mid-range of an encoded window is decoded, and it is written *size* times to the decoded CSV data file (lines 6-7). On the other hand, if the read bit is 1, then the following process is repeated a total of *size* times: a value is decoded and written to the decoded CSV data file (lines 9-12).

```
input  : in: binary file encoded with algorithm variant $\text{PCA}_M$
         out: decoded column of CSV data file
         w: fixed window size
         col_size: number of entries in the column
 1 $n = 0$
 2 while $n < col\_size$ do
 3     $size = \min\{w, col\_size - n\}$
 4     Decode bit from in
 5     if $bit == 0$ then
 6         Decode mid_range using $B_c$ bits
 7         Output size copies of mid_range to out
 8     else
 9         repeat size times
10             Decode value using $B_c$ bits
11             Output value to out
12         end
13     end
14     $n \mathrel{+}= size$
15 end
```

Figure 3.5: Decoding routine for algorithm variant PCA_M.

3.5.1 Example

Next we present an example of the encoding of the 12 samples illustrated in Figure 3.6. Recall that the specific timestamp values are irrelevant for algorithm PCA. For this example we let the error threshold parameter (ϵ) be equal to 1, and the fixed window size (w) equal to 4.

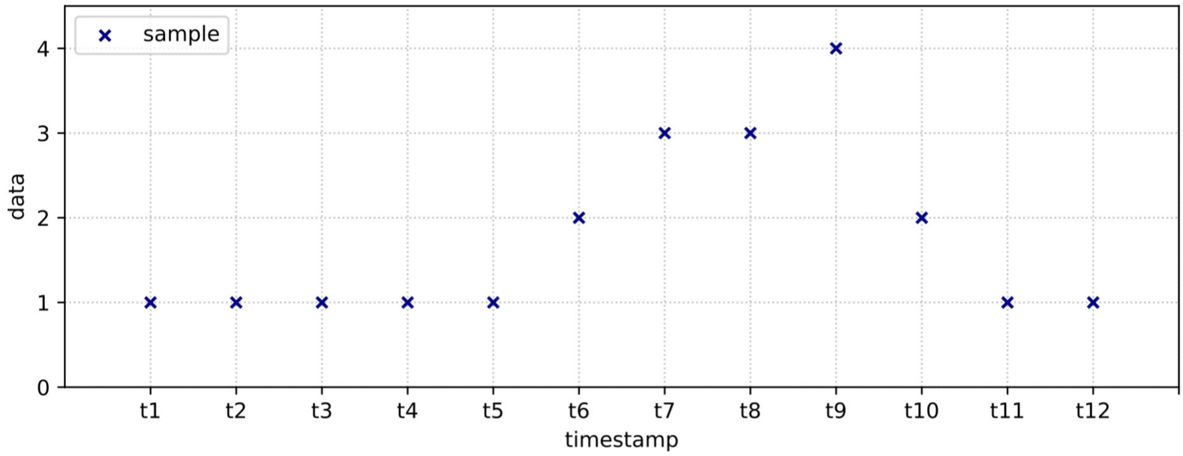

FIGURE 3.6: Example: Signal consisting of 12 samples.

Since there are 12 samples to encode and $w = 4$, three windows are encoded independently, each consisting of exactly four samples. The first window includes the first four samples, which are all equal to 1, so in this case the condition in Line 4 of the coding routine is satisfied. Therefore, lines 5-7 are executed, which encode a single value, *mid_range*, as a representation of the four samples in the window. For this first window, *mid_range* equals 1. Figure 3.7 shows this step in the graph. Notice that, since all the values in the window are equal, the condition in Line 4 would be satisfied regardless of the value of parameter ϵ.

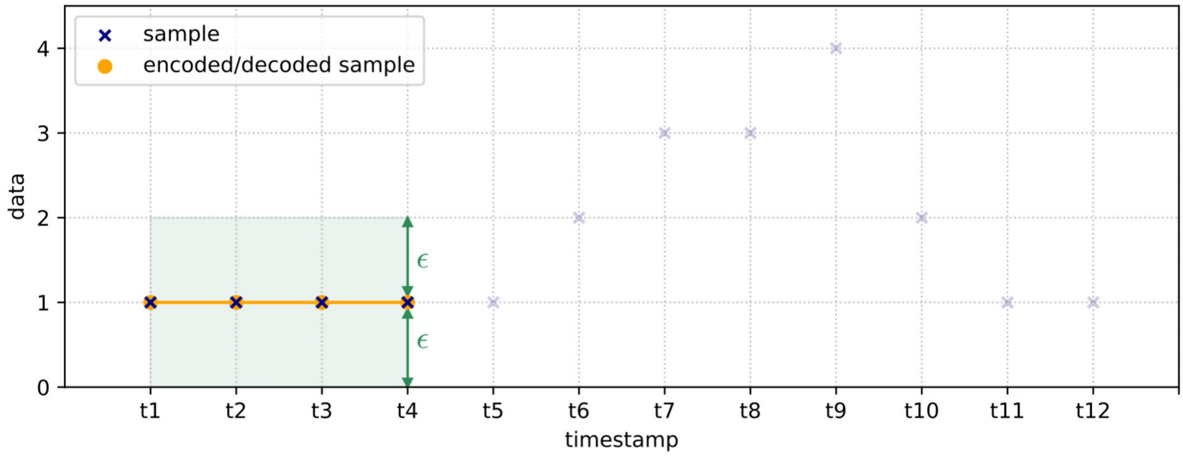

FIGURE 3.7: Example: Algorithm variant PCA$_M$ with $\epsilon = 1$ and $w = 4$. Step 1.

The second window is comprised of the next four samples, i.e. $[1, 2, 3, 3]$. Again, the condition in Line 4 is satisfied, because we have $|3 - 1| \leq 2 * 1$, but in this case *mid_range* equals 2, so these four samples are described through the encoding of a single value, 2. This step is shown in Figure 3.8.

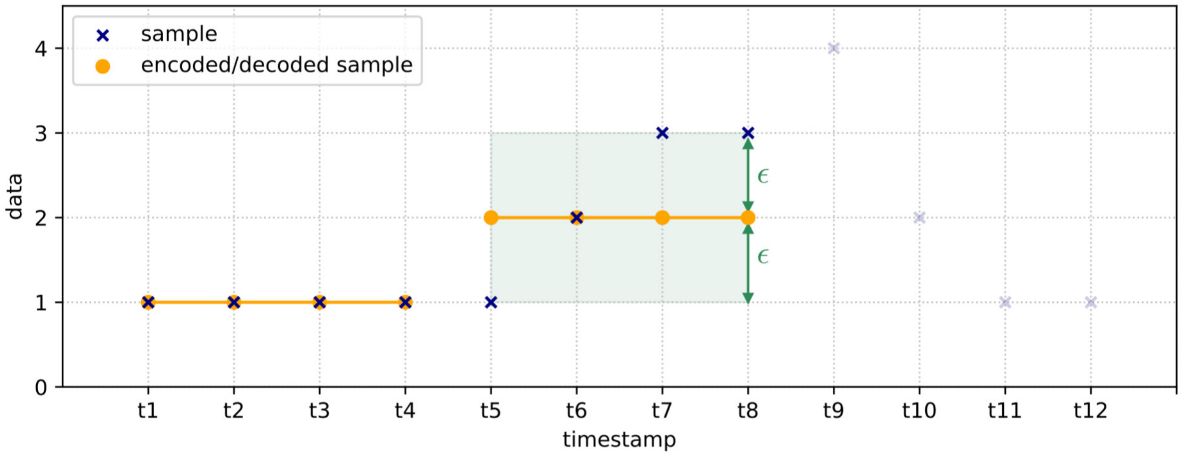

FIGURE 3.8: Example: Algorithm variant PCA$_M$ with $\epsilon = 1$ and $w = 4$. Step 2.

The third and last window consists of the last four samples, i.e. $[4, 2, 1, 1]$. In this case, the condition in Line 4 evaluates to false, so lines 9-12 are executed, which encode each sample value separately. This last step is shown in Figure 3.9.

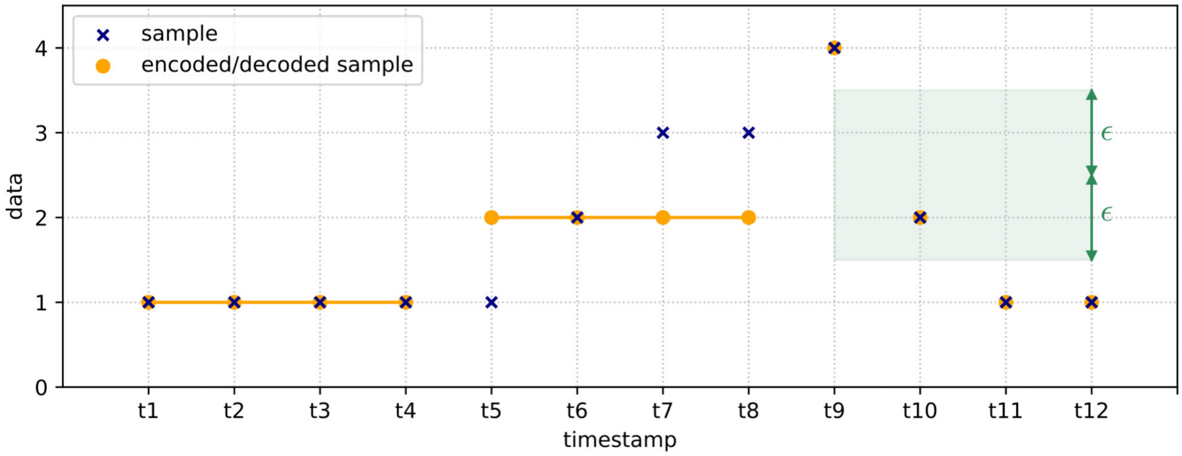

FIGURE 3.9: Example: Algorithm variant PCA$_M$ with $\epsilon = 1$ and $w = 4$. Step 3.

This simple example fairly represents every scenario that may arise during the encoding process. Since the threshold condition holds for the first two windows, both are encoded with exactly the same number of bits, i.e. $1 + B_c$. On the other hand, since the threshold condition does not hold for the last window, it is encoded with $1 + w * B_c$ bits. Thus, windows consisting of adjacent samples are encoded using fewer bits. This example illustrates why algorithm PCA is expected to achieve better compression performances on smooth (i.e. slowly varying) rather than rough (i.e. with abrupt changes) signals. Overall, $2 * (1 + B_c) + 1 + w * B_c = 3 + 6 * B_c$ bits are required for encoding the 12 samples in this example.

3.5.2 Non-Masking (*NM*) Variant

In Figure 3.10 we show the coding routine for variant *NM* of algorithm PCA. In this case, the column entries may be not only integers representing sample values, but also the character "N", which represents a gap in the data. As in variant *M*, after parsing the column entries into consecutive non-overlapping windows of size w (Line 1), each of these windows is encoded independently (lines 3-23). However, since not every entry in a window is guaranteed to be an integer, we consider additional scenarios when encoding a window.

```
input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm variant PCA$_{NM}$
        $\epsilon$: maximum error threshold
        w: fixed window size
 1 Parse column into consecutive non-overlapping windows of size w, except possibly for
     the last window that may consist of fewer samples
 2 foreach window in the parsing, win, do
 3     if every entry in win is equal to "N" then
 4         Output bit 0 to out
 5         Encode NO_DATA using $B_c$ bits
 6     else
 7         Let m and M be the minimum and maximum sample values in win, respectively
 8         if every entry in win is different from "N" and $M - m \leq 2\epsilon$ then
 9             Output bit 0 to out
10             $mid\_range = \lfloor (m + M)/2 \rfloor$
11             Encode mid_range using $B_c$ bits
12         else
13             Output bit 1 to out
14             foreach entry in win, y, do
15                 if y == "N" then
16                     value = NO_DATA
17                 else
18                     value = y
19                 end
20                 Encode value using $B_c$ bits
21             end
22         end
23     end
24 end
```

FIGURE 3.10: Coding routine for algorithm variant PCA$_{NM}$.

A window *win* can be encoded in three different ways. If every entry in *win* represents a gap in the data (i.e. the condition in Line 3 is satisfied), then bit 0 is output, and the special integer *NO_DATA* is encoded (lines 4-5). If every entry in *win* represents a sample value, and the difference between its maximum and minimum values is less than or equal to 2ϵ (i.e. the condition in Line 8 is satisfied), then bit 0 is output, and the value of *mid_range* for *win* is encoded (lines 9-11). In this case, due to Lemma 3.1, the absolute error between the encoded and the original values in *win* is guaranteed to not be greater than ϵ. In every other case, bit 1 is output, and each of the entries in *win* is encoded separately (lines 13-21), using *NO_DATA* for encoding a gap. Notice that in the first two cases the window is encoded with the same number of bits, i.e. $1 + B_c$, while in the last case the window is encoded with $1 + w * B_c$ bits.

The decoding routine for variant *NM* is quite similar to the decoding routine for variant *M*, presented in Figure 3.5, the only difference being that, in lines 6-7 and 10-11, when *NO_DATA* is decoded, a character "N" is written to the decoded CSV data file.

3.6 Algorithm APCA

Algorithm *Adaptive Piecewise Constant Approximation (APCA)* [35] is a constant model algorithm that supports lossless and near-lossless compression. As its name suggests, it works similarly to algorithm PCA, i.e. it approximates a signal with a piecewise constant function, with the difference that, in APCA, the size of the windows into which the data are parsed for encoding is not fixed, but variable. APCA supports both variants, *M* and *NM*.

In Figure 3.11 we show the coding routine for variant *M*, in which all the column entries are integer values (the gaps are encoded separately). It consists of a loop that iterates over all column entries. The algorithm maintains a window of consecutive samples, *win*, which is initially empty (Line 1). In each iteration, the addition of an entry to the window is considered (lines 3-8). If the new entry makes the difference between the maximum and minimum values greater than 2ϵ, or the window size greater than w, then the window is encoded, and emptied (lines 5-6). In any case, the current entry is added to *win* (Line 8), and is eventually encoded. In particular, if the loop ends and *win* is not empty, it is encoded in Line 11. We point out that Lemma 3.1 guarantees that the absolute error between the encoded and the original values in a window is less than or equal to ϵ.

```
input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm variant $APCA_M$
        $\epsilon$: maximum error threshold
        w: maximum window size
 1 Create an empty window, win
 2 foreach entry y in column do
 3     Let m and M be the minimum and maximum sample values in the extended window
        [win, y], resp.
 4     if $M - m > 2\epsilon$ or $|win| == w$ then
 5         EncodeWindow(win, out, w) // routine shown in Figure 3.12
 6         Set win to an empty window
 7     end
 8     Append y to win
 9 end
10 if win is not empty then
11     EncodeWindow(win, out, w) // routine shown in Figure 3.12
12 end
```

Figure 3.11: Coding routine for algorithm variant APCA_M.

The auxiliary routine used for encoding a window (in lines 5 and 11), EncodeWindow, is shown in Figure 3.12. Observe that every window is encoded with the same number of bits, i.e. $\lceil \log_2 w \rceil + B_c$, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and B_c bits are used for encoding its mid-range.

```
input : win: window to encode
        out: binary file encoded with algorithm variant $APCA_M$
        w: maximum window size
Encode $|win|$ using $\lceil \log_2 w \rceil$ bits
Let m and M be the minimum and maximum sample values in win, respectively
$mid\_range = \lfloor (m + M)/2 \rfloor$
Encode mid_range using $B_c$ bits
```

Figure 3.12: Auxiliary routine EncodeWindow for algorithm variant APCA_M.

The decoding routine for variant M is shown in Figure 3.13. It consists of a loop that keeps running until every entry in the column has been decoded, which occurs when the condition in Line 2 becomes false. The decoding loop is fairly simple. First, both the window size, *size*, and its mid-range value are decoded (lines 3-4). Then, the mid-range value is written *size* times to the decoded CSV data file (Line 5).

```
input  : $in$: binary file encoded with algorithm variant APCA$_M$
         $out$: decoded column of CSV data file
         $w$: maximum window size
         $col\_size$: number of entries in the column
1 $n = 0$
2 while $n < col\_size$ do
3     Decode $size$ using $\lceil \log_2 w \rceil$ bits
4     Decode $mid\_range$ using $B_c$ bits
5     Output $size$ copies of $mid\_range$ to $out$
6     $n \mathrel{+}= size$
7 end
```

FIGURE 3.13: Decoding routine for algorithm variant APCA$_M$.

3.6.1 Example

Next we present an example of the encoding of the same 12 samples of the previous example, presented in Figure 3.6, and also shown in Figure 3.14 below. Recall that the specific timestamp values are irrelevant for algorithm APCA. For this example we let the error threshold parameter (ϵ) be equal to 1, and the maximum window size (w) equal to 256.

The condition in Line 4 of the coding routine evaluates to false in the first eight iterations, so these samples, $[1, 1, 1, 1, 1, 2, 3, 3]$, are added to the first window. The sample processed in the 9th iteration is 4, whose addition to the window would violate the error threshold constraint, because we have $|4 - 1| > 2 * 1$. Therefore, the window is encoded, which requires $\lceil \log_2 w \rceil = \log_2 256 = 8$ bits for encoding its size (i.e. 8), and B_c bits for encoding its mid-range (i.e. 2), and the sample value 4 is added to a new empty window. This step is shown in Figure 3.14.

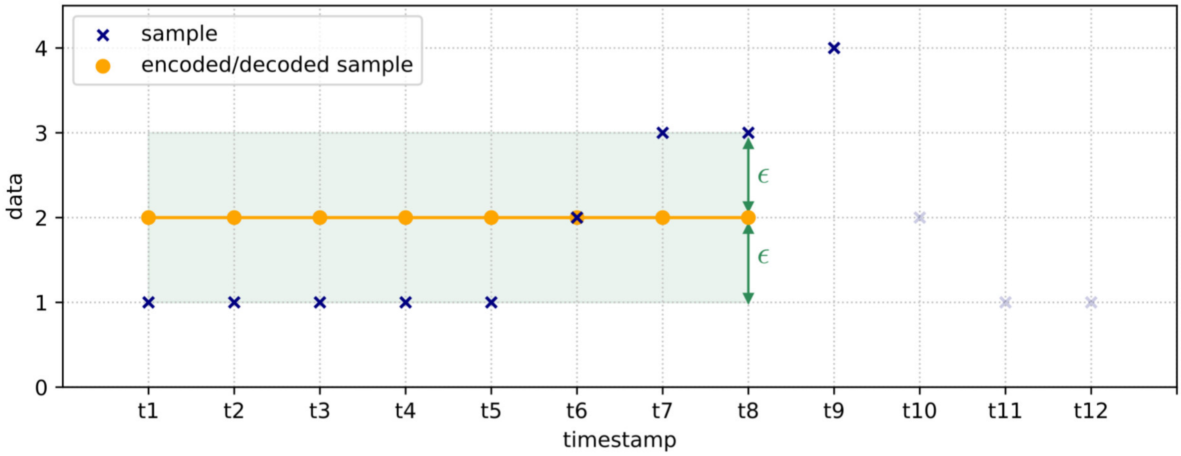

FIGURE 3.14: Example: Algorithm variant APCA$_M$ with $\epsilon = 1$ and $w = 256$. Step 1.

For the second window, the condition in Line 4 evaluates to false in the 10th iteration. However, the error threshold constraint is violated in the 11th iteration, for the sample value 1. The second window, $[4, 2]$, is encoded with size 2 and mid-range 3, and the sample value 1 is added to a new empty window. This step is shown in Figure 3.15.

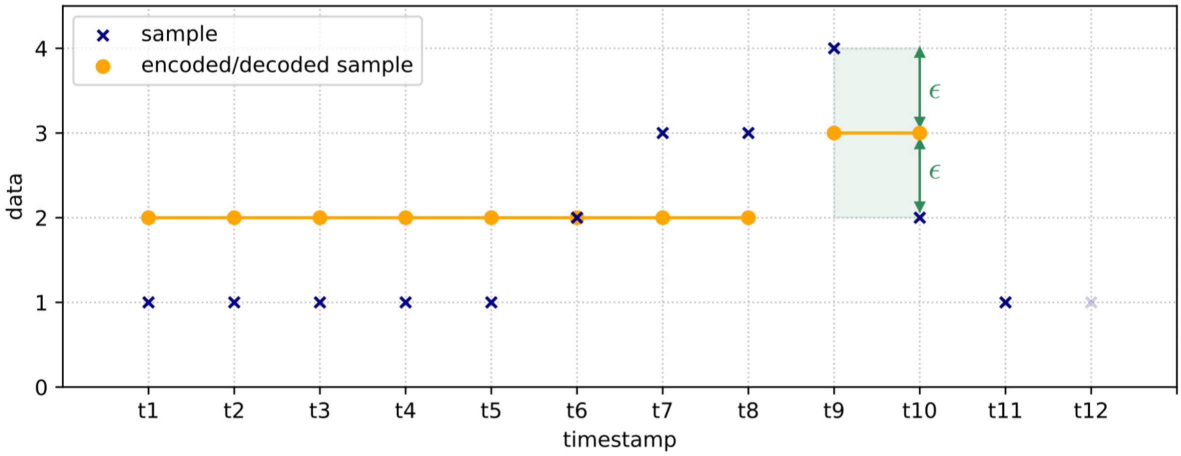

FIGURE 3.15: Example: Algorithm variant APCA$_M$ with $\epsilon = 1$ and $w = 256$. Step 2.

For the third window, the condition in Line 4 evaluates to false in the 12th and last iteration. This window, which is equal to $[1, 1]$, is encoded in Line 11, after executing the last iteration. In this case, the window size is 2 and its mid-range is 1. This last step is shown in Figure 3.16.

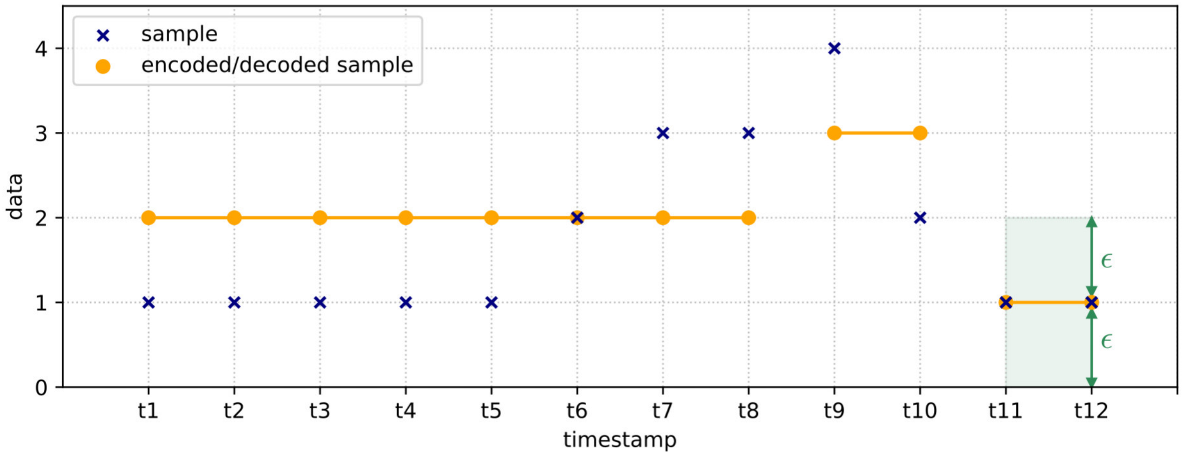

FIGURE 3.16: Example: Algorithm variant APCA$_M$ with $\epsilon = 1$ and $w = 256$. Step 3.

We point out that each of the three windows in this example is encoded using exactly the same number of bits, i.e. $\lceil \log_2 w \rceil + B_c$. However, the first window consists of eight samples, while the last two consist of only two samples. Therefore, the first window achieves a better compression ratio (more samples encoded per bit). This example illustrates why, similarly to PCA, algorithm APCA is expected to achieve better compression performances on smooth rather than rough signals. Overall, $3 * (\lceil \log_2 w \rceil + B_c) = 24 + 3 * B_c$ bits are required for encoding the 12 samples in this example.

3.6.2 Non-Masking (*NM*) Variant

The coding and decoding routines for variant *NM* of algorithm APCA are similar to their variant *M* counterparts, the difference being that the former routines are able to handle both sample values and gaps. Recall that in the coding routine for variant *M*, a window is encoded when adding a new entry makes it violate the error threshold constraint or the window size restriction (Line 4 in Figure 3.11). In the coding routine for variant *NM*, a window must also be encoded if the incoming entry is character “N” (gap in the data) and the other entries in the window are integers (sample values), or vice versa. A window that consists of gaps is encoded with the same number of bits as a window that consists of integers, i.e. $\lceil \log_2 w \rceil + B_c$, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and B_c bits are used for encoding the special integer *NO_DATA*.

3.7 Encoding Scheme for Linear Model Algorithms

In the following four sections we present the evaluated linear model algorithms, i.e. PWLH, PWLHInt, CA, SF and FR. As we recall from Section 3.1, this type of algorithms approximate signals using linear functions. Even though the encoding scheme varies among algorithms, it always requires encoding a sequence of line segments in the two-dimensional Euclidean space. Each line segment represents a sequence of consecutive samples, with x and y-axis corresponding to timestamps and sample values, respectively. A *sample point* is a pair of integers (x, y), where y is a sample value (i.e. a column entry other than “N”, which represents a data gap) with an associated timestamp x.

A linear model encoding algorithm operates by successively encoding endpoints of segments, which span samples whose vertical distance to the segment along the y-axis does not exceeds the prescribed error threshold parameter (ϵ). The decoder, in turn, sequentially decodes the endpoints of each segment and linearly interpolates the intermediate samples, up to integer constraints. In Figure 3.17 we present the auxiliary routine FillSegment, which performs this interpolation; it is invoked by the decoding routine for every linear model algorithm variant. Its inputs are the timestamp column (recall that this column is encoded first and, thus, it is available to the decoder when decoding sample columns), and a pair of timestamps and sample values, which correspond to the coordinates of a segment endpoints (Line 1). The output is a list consisting of the (integer) sample values that are decoded from said segment.

```
input  : t_col: timestamp column
         $t_o$: timestamp of the first endpoint of the segment
         $t_f$: timestamp of the last endpoint of the segment
         $s_o$: sample value of the first endpoint of the segment
         $s_f$: sample value of the last endpoint of the segment
output: decoded_samples: list with the sample values decoded from the segment
1 Let segment be the line segment whose endpoints coordinates are ($t_o$,$s_o$) and ($t_f$,$s_f$)
2 Create an empty list, decoded_samples
3 foreach timestamp $t_i$ in t_col, such that $t_o \leq t_i \leq t_f$, do
4     Let $s_i$ be the sample value obtained by substituting $t_i$ for the x-coordinate in the
        segment equation, and rounding the result to the nearest integer
5     Append $s_i$ to decoded_samples
6 end
7 return decoded_samples
```

Figure 3.17: Auxiliary routine FillSegment for linear model algorithms.

For most algorithms, both the x-coordinates and the y-coordinates of the endpoints are encoded as integers, the exceptions being algorithm PWLH, presented in Section 3.8, which encodes the y-coordinates as floats[1], and algorithm SF, presented in Section 3.10, which encodes the coordinates of both axes as floats.

Next, we present an example that illustrates the working of the auxiliary routine FillSegment. The inputs are represented in Figure 3.18: the coordinates of the encoded segment endpoints are $(t_5, 1)$ and $(t_9, 3.5)$, while the timestamp column is equal to $[t_1, ..., t_N]$, where $N \geq 9$. The segment defined in Line 1 of the routine is colored red. After creating the empty list of decoded samples (Line 2), a loop iterates over every timestamp t_i, $t_5 \leq t_i \leq t_9$, in the column, it decodes the corresponding sample value s_i (Line 4), and adds it to the list (Line 5). Given timestamp t_i, sample value s_i is obtained by taking the equation of the segment and substituting t_i for the x-coordinate, and then rounding the result to the nearest integer. In Figure 3.18, the decoded sample values, s_i, $5 \leq i \leq 9$, are colored in orange. They are equal to $[1, 2, 2, 3, 4]$, which is the list output by the routine FillSegment in this example.

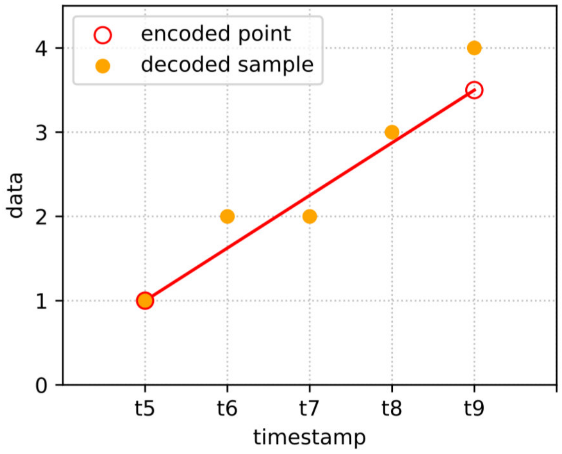

FIGURE 3.18: Example of the auxiliary routine FillSegment for linear model algorithms.

3.8 Algorithms PWLH and PWLHInt

Algorithm *PieceWise Linear Histogram (PWLH)* [36] is a linear model algorithm that supports lossless and near-lossless compression. For PWLH we define both variants, *M* and *NM*. We also define algorithm *PWLHInt* by introducing minor design changes to algorithm PWLH. The description in the current section applies to both PWLH and PWLHInt, except for the specific differences that are pointed out in Subsection 3.8.2.

PWLH is a linear model algorithm, so its encoding process involves the encoding of a sequence of line segments. In Figure 3.19 we present the coding routine for variant *M*. It consists of a loop that iterates over all column entries, which are always integer values (the gaps are encoded separately). The algorithm maintains a window of consecutive sample points, *win*, which is initially empty (Line 1). In each iteration, the addition of a new incoming point, E, to the window is considered (lines 3-16). Notice that E, defined in lines 3-4, is a sample point (recall definition in Section 3.7): its y-coordinate is an integer sample value, and its x-coordinate is the timestamp (obtained from the timestamp column) for said sample. We point out that in algorithm PWLH, as well as in every evaluated linear model algorithm, windows consist of sample points, while in the constant and correlation model algorithms, windows consist of integer values corresponding to column entries.

[1] In Subsection 3.8.2 we define algorithm PWLHInt, which is an adaptation of algorithm PWLH that encodes the y-coordinates as integers.

```
input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm variant PWLH$_M$
        $\epsilon$: maximum error threshold
        w: maximum window size
        t_col: timestamp column
1  Create an empty window, win
2  foreach entry y in column do
3      Obtain timestamp for y, $t_y$, from t_col
4      Let E be the point with coordinates ($t_y$, y)
5      if y is the first entry in column then
6          Append E to win
7          continue
8      end
9      Let S be the set of points in the extended window [win, E]
10     Let H be the convex hull of S
11     Let valid_hull be true iff H satisfies the valid hull condition, defined in 3.8.1, for $\epsilon$
12     if not valid_hull or |win| == w then
13         EncodeWindow(win, out, w, t_col) // routine shown in Figure 3.22
14         Set win to an empty window
15     end
16     Append E to win
17 end
18 if win is not empty then
19     EncodeLastWindow(win, out, w, t_col) // routine shown in Figure 3.23
20 end
```

FIGURE 3.19: Coding routine for algorithm variant PWLH$_M$.

The incoming points obtained in the subsequent iterations are parsed into consecutive windows of variable size (up to a maximum size w), such that the set of points in a certain window satisfy the *valid hull condition*, defined in the next paragraph. If the set of points in a window satisfies the valid hull condition, the absolute error between the encoded and the original samples is guaranteed to be less than or equal to ϵ. To compute the convex hull we apply Graham's Scan algorithm [53]. Since the points are already sorted by their respective x-coordinates, the sorting step in the algorithm is eliminated, and the time complexity to build and update the convex hull is $O(n)$ instead of $O(n \log n)$ [27].

Definition 3.8.1. Let S be a set of points, with $|S| > 1$, and let H be the convex hull of S. H satisfies the *valid hull condition* for a maximum error threshold ϵ, iff there exists an edge line in the boundary of H, for which the maximum Euclidean distance from any of the points in H to said edge line is less than or equal to 2ϵ.

In figures 3.20 and 3.21 we present an example that illustrates how the valid hull condition is checked in the coding routine of Figure 3.19. In Figure 3.20, set S, which is defined in Line 9, contains the three points in *win*, plus the incoming point, *E4*. In this case, the convex hull of S, H, satisfies the valid hull condition for $\epsilon = 1$, since the maximum distance from any of its points to the upper edge line of its boundary is approximately 1.1, which is less than $2\epsilon = 2$. Therefore, the condition in Line 12 evaluates to false (we assume that $|win| < w$), and *E4* is added to *win* (Line 16). In the next step, presented in Figure 3.21, set S contains the four points in *win*, plus *E5*. In this case, H does not satisfy the valid hull condition, since for each of the three edge lines in its boundary, there exists a point in H such that its distance to the edge line is greater than 2ϵ. Therefore, the condition in Line 12 is satisfied, so *win* is encoded (Line 13), emptied (Line 14) and added *E5* (Line 16).

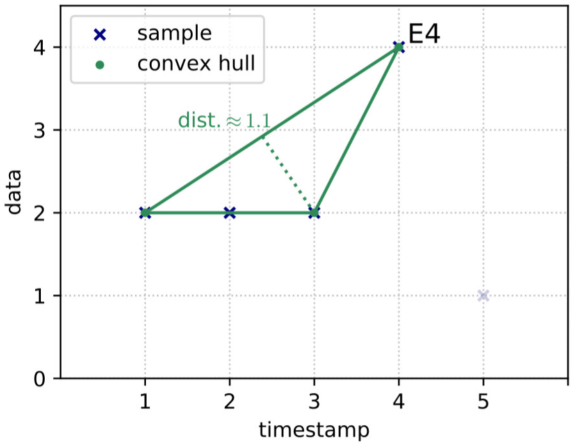

FIGURE 3.20: Example: checking the valid hull condition. Variant PWLH$_M$ with $\epsilon = 1$. Step 1.

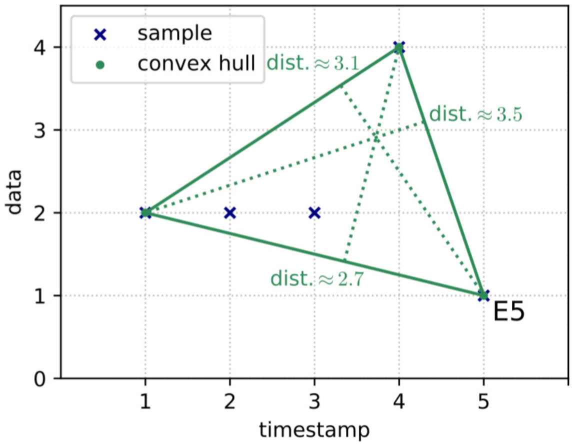

FIGURE 3.21: Example: checking the valid hull condition. Variant PWLH$_M$ with $\epsilon = 1$. Step 2.

A window is encoded via the auxiliary routine EncodeWindow, shown in Figure 3.22. This method first encodes the window size using $\lceil \log_2 w \rceil$ bits (Line 1), then selects the segment that minimizes the *mean square error (MSE)* for the points in the window, taken among all the segments for which the x-coordinates of the first and last endpoints match the x-coordinates of the first and last points in the window, respectively (lines 2-3). Notice that the y-coordinates of the two endpoints of this segment are float values, which may not match the value of any column entry, and might even be out of the range specified in the dataset CSV file for the corresponding data type. Finally, in lines 5-6, both y-coordinates are encoded as floats, using 32 bits. Recall that the coding algorithm encodes the timestamp column, with the respective x-coordinates, first. Thus, the decoding routine is able to decode both coordinates of each endpoint.

```
input  : win: window to encode
         out: binary file encoded with algorithm variant PWLH_M
         w: maximum window size
         t_col: timestamp column
1 Encode |win| using $\lceil \log_2 w \rceil$ bits
2 Let LS be the set of line segments such that $s \in LS$ iff the x-coordinates of the first and
    last endpoints of s match the x-coordinates of the first and last points in win, resp.
3 Let $segment \in LS$ be the line segment that minimizes the MSE for the points in win
    (among the segments included in LS)
4 Let $s_o$ and $s_f$ be the y-coordinates of the endpoints of segment
5 Encode $s_o$ as a float, using 32 bits
6 Encode $s_f$ as a float, using 32 bits
```

FIGURE 3.22: Auxiliary routine EncodeWindow for algorithm variant PWLH$_M$.

We point out that calculating the segment that minimizes the MSE for a set of points is computationally more expensive than checking the valid hull condition [27]. This is the reason why the valid hull condition is checked in every iteration, when deciding whether or not a point should be added to the window, while the segment that minimizes the MSE is only computed for the points in a complete window.

In the coding routine presented in Figure 3.19, if *win* is not empty after executing the last iteration of the loop, it is encoded in Line 19, via the auxiliary routine EncodeLastWindow, shown in Figure 3.23. If *win* consists of a single point, its size is encoded using $\lceil \log_2 w \rceil$ bits, and the y-coordinate of the point is encoded using B_c bits. On the other hand, if *win* consists of multiple points, it is encoded in the same way as the previous windows, i.e. via the auxiliary routine EncodeWindow, shown in Figure 3.22.

```
input : win: window to encode
        out: binary file encoded with algorithm variant $PWLH_M$
        w: maximum window size
        t_col: timestamp column
if $|win| == 1$ then
    Encode 1 using $\lceil \log_2 w \rceil$ bits
    Encode the y-coordinate of the single point in win using $B_c$ bits
else
    EncodeWindow(win, out, w, t_col) // routine shown in Figure 3.22
end
```

FIGURE 3.23: Auxiliary routine EncodeLastWindow for algorithm variant PWLH_M.

The decoding routine for variant M is shown in Figure 3.24. It consists of a loop that repeats until every entry in the column has been decoded, which occurs when condition in Line 2 becomes false. Recall that the coding algorithm encodes the timestamp column first (Line 6 in Figure 3.1), so the timestamps are known to the decoding routine (input *t_col*). Each iteration of the loop starts with the decoding of the window size (Line 3). If the window size is greater than 1, then the points in the window are modeled by a line segment. In this case, the decoding routine decodes the float representation of the y-coordinates of the segment endpoints (lines 5-6), obtains the timestamps corresponding to their x-coordinates (Line 7), and calls the auxiliary routine FillSegment with those inputs (Line 8). As we recall from Figure 3.17, routine FillSegment returns a list consisting of the sample values that are decoded from the segment, which are then written in the decoded CSV data file (lines 9-11). Notice that, even though inputs s_o and s_f are floats, FillSegment returns a list of integers. When the window size is equal to 1, a value is decoded (using B_c bits) and written to the decoded file (lines 13-14).

```
input : in: binary file encoded with algorithm variant $PWLH_M$
        out: decoded column of CSV data file
        w: maximum window size
        col_size: number of entries in the column
        t_col: timestamp column
 1 $n = 0$
 2 while $n < col\_size$ do
 3     Decode size using $\lceil \log_2 w \rceil$ bits
 4     if $size > 1$ then
 5         Decode $s_o$ as a float, using 32 bits
 6         Decode $s_f$ as a float, using 32 bits
 7         Obtain timestamps for the endpoints of the segment, $t_o$ and $t_f$, from t_col
 8         samples = FillSegment(t_col, $t_o$, $t_f$, $s_o$, $s_f$) // routine in Figure 3.17
 9         foreach sample in samples, value, do
10             Output value to out
11         end
12     else
13         Decode value using $B_c$ bits
14         Output value to out
15     end
16     $n$ += size
17 end
```

FIGURE 3.24: Decoding routine for algorithm variant PWLH_M.

3.8.1 Example

Next we present an example of the encoding of the same 12 samples of previous examples, presented in Figure 3.6, and also shown in Figure 3.25 below. For this example we let the interval between consecutive timestamps be equal to 60, the error threshold parameter (ϵ) equal to 1, and the maximum window size (w) equal to 256.

Since there are only 12 samples to encode, no window in this example can reach the maximum size (256). Therefore, a window is only encoded when the convex hull, defined in Line 10 of the coding routine, violates the valid hull condition defined in Line 11. The first iteration is the only one in which the condition in Line 5 is satisfied, so the algorithm just adds the first incoming point, *E1*, to the window (Line 6). Figure 3.25 shows this step in the graph. Observe that, besides the sample points, the convex hull for the current window is also shown.

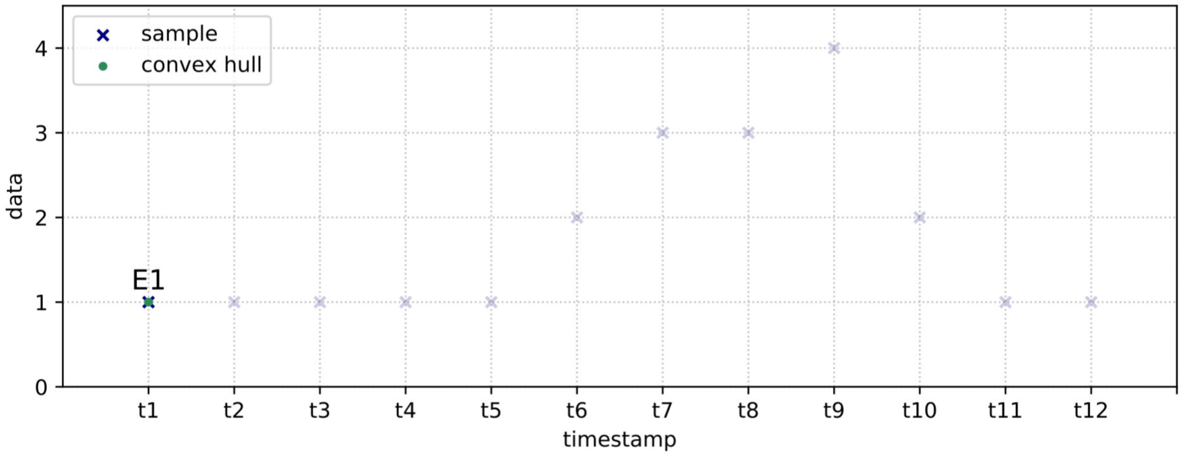

FIGURE 3.25: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 1.

In the second iteration, the boundary of the convex hull that includes incoming point *E2* consists of a single line segment, where the maximum distance from either point to the line is zero, so the valid hull condition is satisfied, and *E2* is added to the window. This step is shown in Figure 3.26.

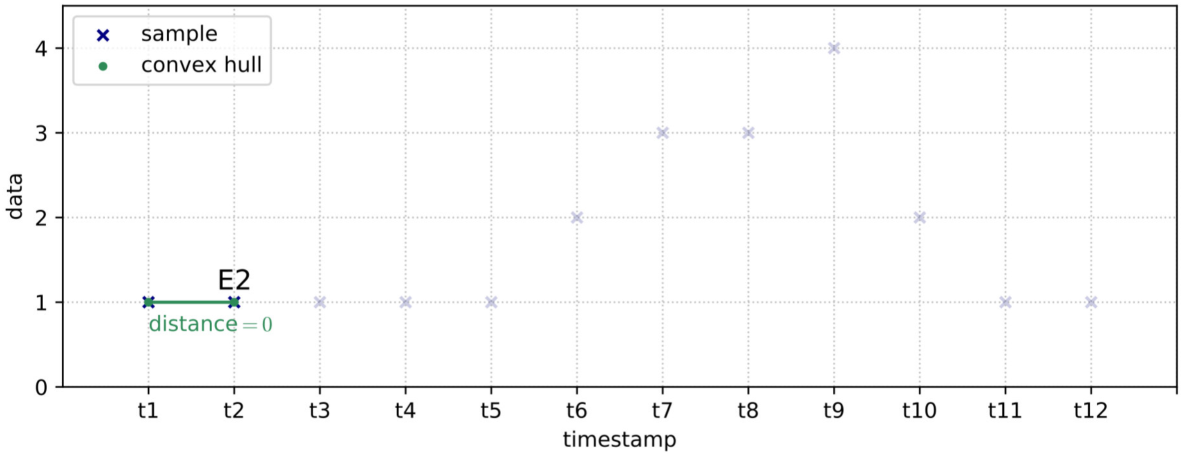

FIGURE 3.26: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 2.

The following three sample values are also equal to 1. The convex hull that includes the respective incoming points still consists of a single line segment, so these points are also added to the window. This step is shown in Figure 3.27.

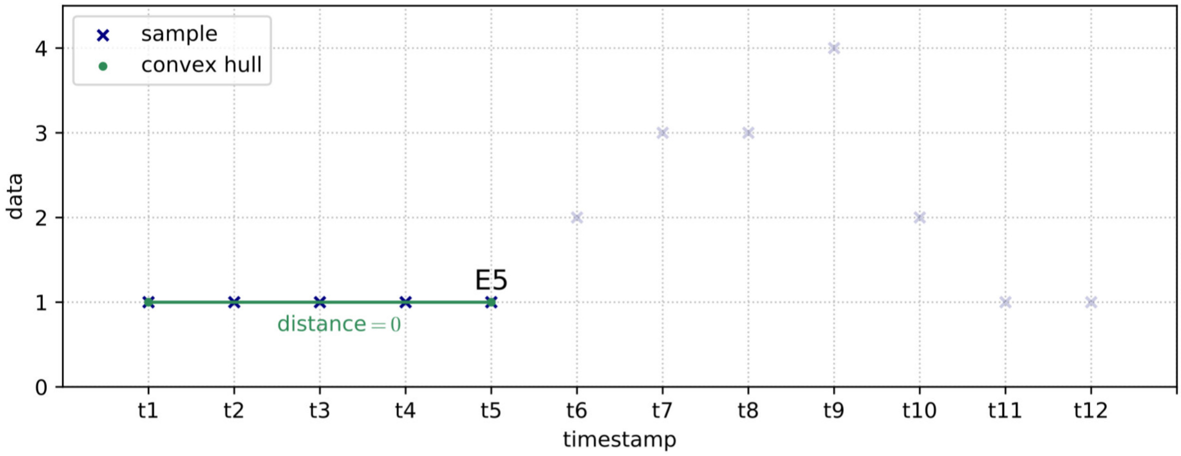

FIGURE 3.27: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 3.

In the 6th iteration, sample value 2 is considered. The updated convex hull, whose boundary now consists of three edges, is shown in Figure 3.28. In this case, the maximum distance between the upper edge in its boundary and any of its points is approximately 0.8, which is less than $2\epsilon = 2$. Therefore, the valid hull condition is still satisfied, and *E6* is added to the window.

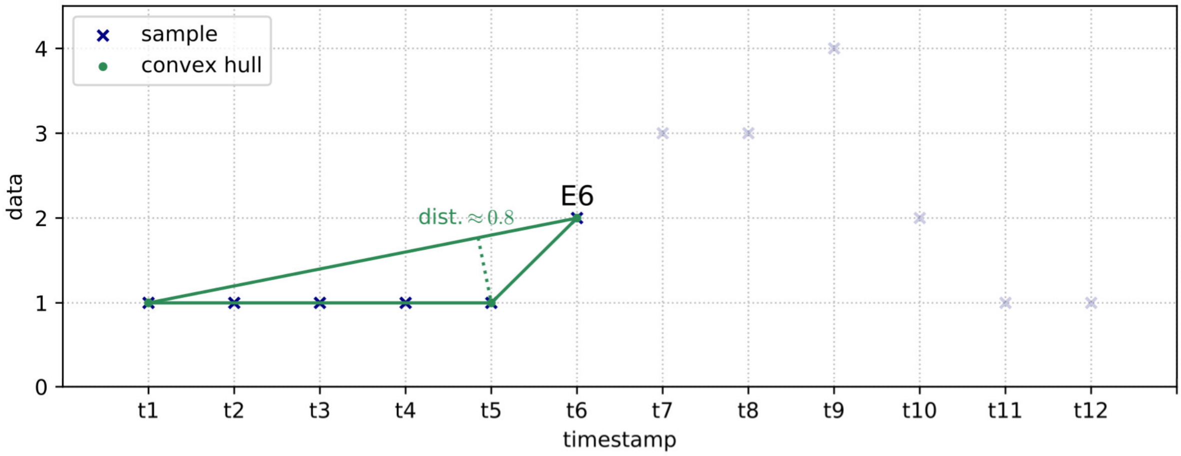

FIGURE 3.28: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 4.

The following three iterations are similar to the previous one. In every case, the convex hull is updated, and, even though the maximum distance between the upper edge in its boundary and any of its points increases, it is never greater than 2ϵ, so the three incoming points are added to the window. These steps are shown in figures 3.29, 3.30 and 3.31.

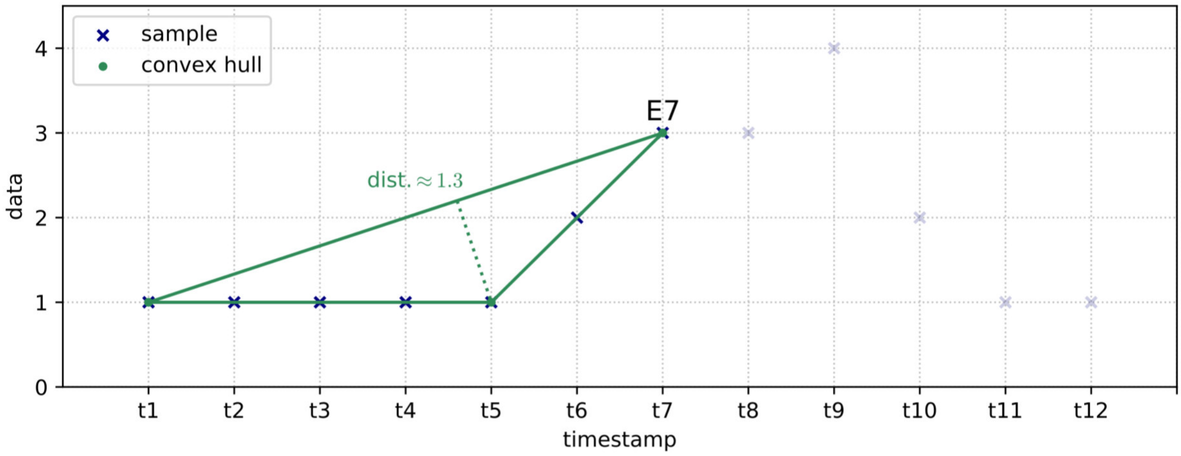

FIGURE 3.29: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 5.

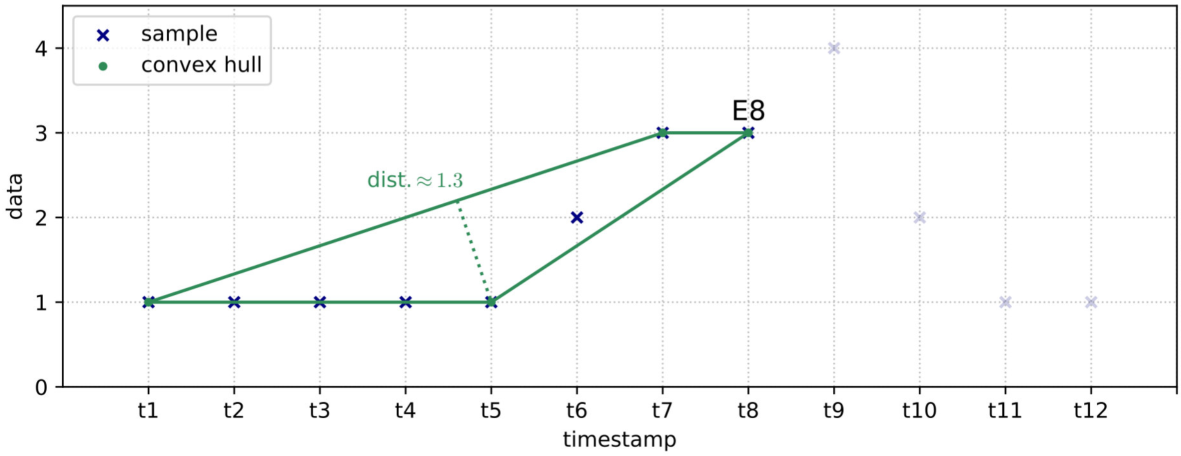

FIGURE 3.30: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 6.

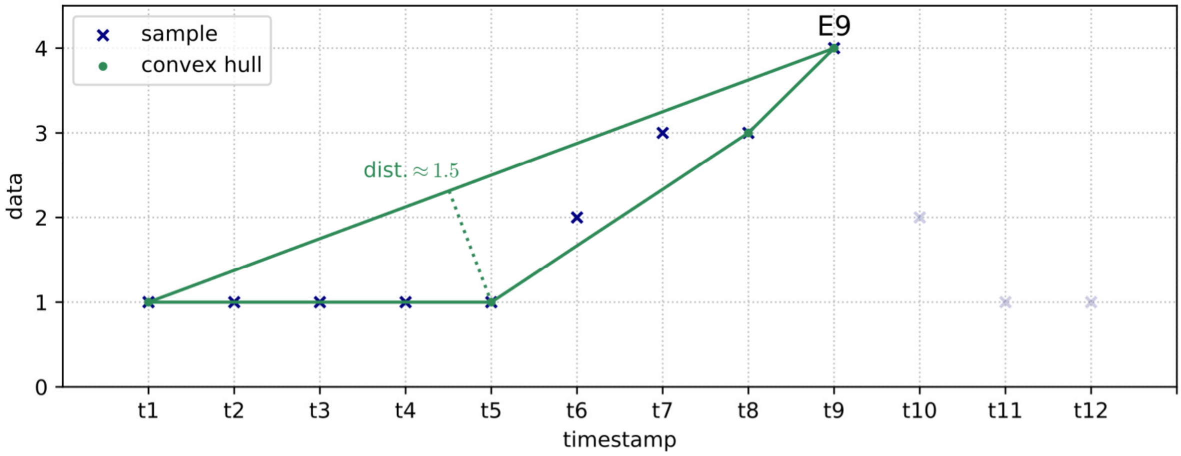

FIGURE 3.31: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 7.

Eventually, in the 10th iteration, sample value 2 is considered. In the updated convex hull, which is shown in Figure 3.32, for the first time the valid hull condition is not satisfied, i.e. *valid_hull* in Line 11 becomes false. Observe that, for each of the four edges in the boundary of the convex hull, there exists a point in the hull such that its distance to the edge line is greater than 2ϵ.

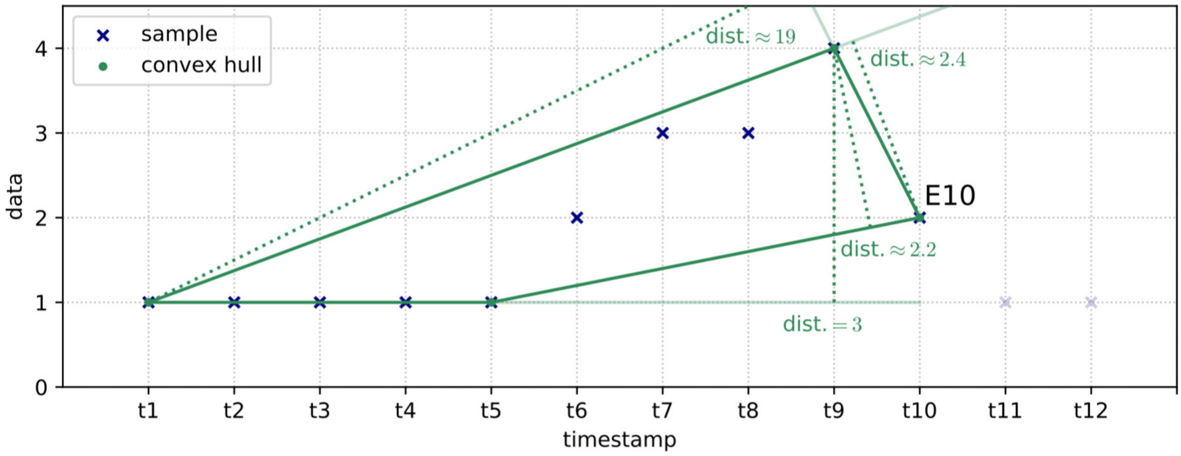

FIGURE 3.32: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 8.

Since the condition in Line 12 becomes true, the window is encoded via the auxiliary routine EncodeWindow (Line 13), and the window is emptied and added *E10* (lines 14 and 16). Routine EncodeWindow models the points in the window through the line segment that minimizes the MSE for said points (recall Figure 3.22). This segment and its two encoded endpoints are shown in Figure 3.33. Encoding the window requires $\lceil \log_2 w \rceil = \log_2 256 = 8$ bits for encoding its size (i.e. 9), and 32 bits for encoding each of the float values corresponding to the y-coordinates of the segment endpoints.

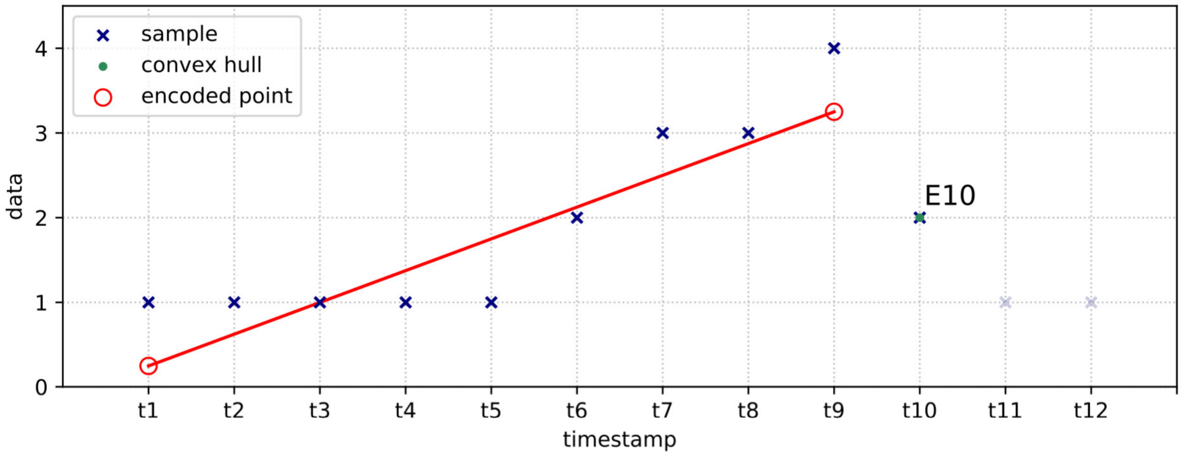

FIGURE 3.33: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 9.

In the last two iterations of the coding routine, which correspond to the last two samples, the valid hull condition is not violated. Therefore, after executing the last iteration, *win* includes three points, which are encoded via the auxiliary routine EncodeLastWindow (Line 19), also using $\lceil \log_2 w \rceil + 2 * 32$ bits. The associated segment, with its two encoded endpoints, is shown in Figure 3.34. In this figure, we also display the values of the decoded samples, which are the values that the decoding routine, shown in Figure 3.24, writes to the decoded CSV data file. Overall, $2*(\lceil \log_2 w \rceil + 2*32) = 144$ bits are required for encoding the 12 samples in this example.

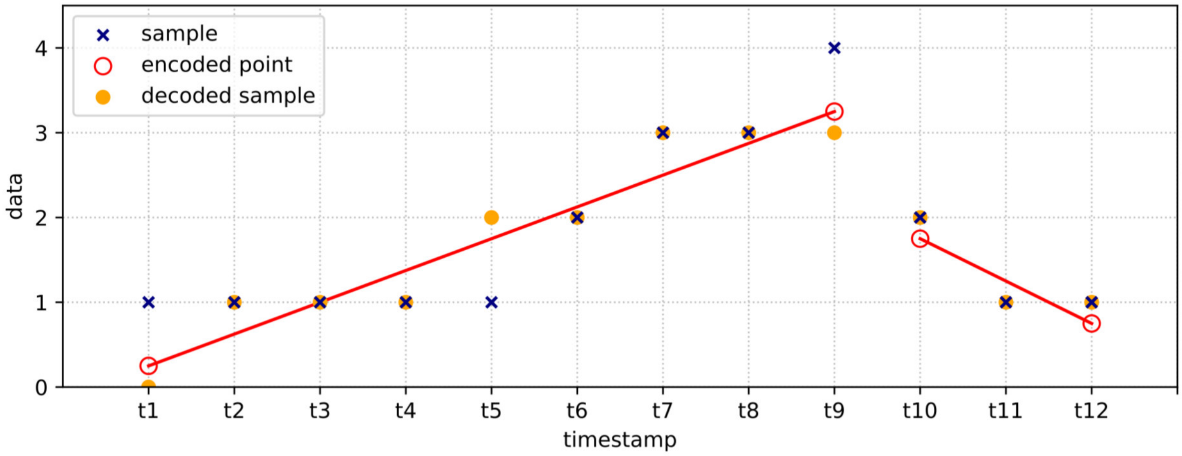

FIGURE 3.34: Example: Algorithm variant PWLH$_M$ with $\epsilon = 1$ and $w = 256$. Step 10.

3.8.2 Differences Between Algorithms PWLH and PWLHInt

Recall, from Section 2.1, that, in the signals we are interested in compressing, the data samples are always represented as integers. However, algorithm PWLH [36] encodes the y-coordinates of both endpoints of a line segment as floats, using 32 bits (lines 4-5 in Figure 3.22). When $B_c < 32$ (recall, from Section 3.2, that this is the case in our experimental datasets, where the maximum B_c is 17), the compression performance of the algorithm can be improved if we transform these y-coordinates into the (integer) domain for the associated data column, since this change allows us to encode a y-coordinate using B_c bits. That is precisely the idea behind the design of algorithm PWLHInt.

Algorithm PWLHInt is defined by applying three changes to algorithm PWLH. First, we change lines 5-6 in the auxiliary routine EncodeWindow, shown in Figure 3.22, so that both y-coordinates are rounded to the nearest integer, and then encoded using B_c bits. Lines 5-6 are modified accordingly in the decoding routine, shown in Figure 3.24, this being the only change in the decoding routine. Secondly, we add a new constraint to the condition in Line 12 of the coding routine, shown in Figure 3.19, to make sure that the (rounded) y-coordinates of both endpoints of the approximation segment belong to the range defined for the data column being encoded (in this case, the approximation segment obtained in EncodeWindow must be computed before checking the condition). These first two changes guarantee that B_c bits are enough to encode the y-coordinate of an endpoint. Lastly, rounding a coordinate value to the nearest integer, before encoding it, could represent a deviation of as much as 0.5 from its original value, which may cause a decoding error greater than the threshold ϵ. Therefore, the coding routine of algorithm PWLHInt operates with an adjusted maximum error threshold, $\epsilon' = \epsilon - 0.5$, to make sure that the per-sample absolute error between the decoded and the original signals is less than or equal to ϵ.

Notice that changes made to transform algorithm PWLH into PWLHInt result in a trade-off between factors that affect the compression performance in opposite ways. Rounding and reducing the range of the y-coordinates of the segment endpoints, as well as adjusting the threshold, are factors that are likely to worsen the performance, since, in general, they lead to algorithm PWLHInt encoding more segments than algorithm PWLH. On the other hand, encoding the y-coordinates using B_c instead of 32 bits, is expected to improve the performance of PWLHInt when $B_c < 32$, which is the case in our experimental datasets. In Section 4.4 we evaluate the coding algorithms, and the experimental results suggest that the compression performance of algorithm PWLHInt is superior to that of the original algorithm PWLH [36]. Therefore, the effect of the factors that improve the compression performance of the algorithm outweighs that of those which worsen it, making the trade-off justified by our empirical results

3.8.3 Non-Masking (*NM*) Variant

The coding and decoding routines for variant *NM* of algorithms PWLH and PWLHInt are similar to their respective variant *M* counterparts, the difference being that the former routines are able to handle both sample values and gaps. In the coding routine for variant *NM*, a window may consist either of sample points or gaps, but it cannot include both. Therefore, a new constraint is added to the condition in Line 12 of the coding routine, shown in Figure 3.19, so that a window is also encoded if the new entry is character "N" (gap in the data) and the other window entries are sample points, or vice versa. To encode a window that consists of gaps, algorithm PWLH uses $\lceil \log_2 w \rceil$ bits for encoding its size, and 32 bits for encoding the special float *NO_DATA_FLOAT*, while algorithm PWLHInt also uses $\lceil \log_2 w \rceil$ bits for encoding its size, but it uses B_c bits for encoding the special integer *NO_DATA*.

3.9 Algorithm CA

Algorithm *Critical Aperture (CA)* [37] is a linear model algorithm that supports lossless and near-lossless compression. We implement both variants, *M* and *NM*.

Since CA is a linear model algorithm, its encoding process involves the encoding of a sequence of line segments. In Figure 3.35 we show the coding routine for variant *M*, in which all the column entries are integer values (the gaps are encoded separately). It consists of a loop that iterates over all column entries, parsing them into an alternating sequence of a single sample point followed by a window of variable size (up to a maximum size w), such that all the sample points in the same window lie within vertical distance ϵ from the segment whose endpoints correspond to the single sample point and the last point of the window. The routine operates by successively encoding a single sample point (using an auxiliary routine EncodeArchivedPoint, shown in Figure 3.38), and then growing a window of subsequent sample points, adding one point at a time until it is complete, and finally encoding the size and the last point of the window (using an auxiliary routine EncodeWindow, shown in Figure 3.39).

```
input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm variant $CA_M$
        $\epsilon$: maximum error threshold
        $w$: maximum window size
        t_col: timestamp column
Create an empty window, win
Let SMin and SMax be objects that will represent straight line rays
foreach entry $y$ in column do
    Obtain timestamp for $y$, $t_y$, from t_col
    Let $E$ be the point with coordinates ($t_y$, $y$)
    if $y$ is the first entry in column then
        Make $A = E$, then EncodeArchivedPoint($A$, out) // routine in Figure 3.38
    else if win is empty (i.e. $A$ was defined in the previous iteration) then
        Make $S = E$, then append $S$ to win
        Let SMin and SMax be the rays with initial point $A$, that pass through points
          $(S.x, S.y - \epsilon)$ and $(S.x, S.y + \epsilon)$, respectively
    else
        Let $(A,E)$ be the segment with initial point $A$ that passes through point $E$
        if not slope(SMin) $\leq$ slope(($A$,$E$)) $\leq$ slope(SMax) or $|win|$ == $w$ then
            EncodeWindow(win, out, $w$) // routine shown in Figure 3.39
            Set win to an empty window
            Make $A = E$, then EncodeArchivedPoint($A$, out) // routine in Figure 3.38
        else
            Make $S = E$, then append $S$ to win
            Let SMinOld and SMaxOld be rays equal to SMin and SMax, respectively
            Let SMin and SMax be the rays with initial point $A$, that pass through
              points $(S.x, S.y - \epsilon)$ and $(S.x, S.y + \epsilon)$, respectively
            Let $SMin = \max\{SMinOld, SMin\}$ and $SMax = \min\{SMaxOld, SMax\}$
        end
    end
end
if win is not empty then
    EncodeWindow(win, out, $w$) // routine shown in Figure 3.39
end
```

Figure 3.35: Coding routine for algorithm variant CA_M.

The algorithm determines when to stop growing the window, i.e., when the current window *win* is complete, based upon three sample points: *incoming (E)* is the point corresponding to the column entry for the current iteration; *archived (A)* is the point most recently encoded, and it corresponds to a single sample that precedes a window; *snapshot (S)* is the point most recently added to *win*. The algorithm maintains a cone, determined by two rays departing from A, *SMin* and *SMax*, such that E is added to *win* as long as it lies within the cone. This cone, in turn, is defined so that if E is indeed added to *win* (thus becoming point S), all the points in *win* are at a vertical distance at most ϵ from the line segment (A, S). Since the encoding process involves the encoding of a sequence of these segments (A, S), the absolute error between the encoded and the original samples is guaranteed to be at most ϵ.

In figures 3.36 and 3.37 we present an example that illustrates the key steps of the coding algorithm. In Figure 3.36, the three points, E, A, and S, as well as the two rays departing from A, *SMin* and *SMax*, are shown. A is defined (and encoded) in the first iteration (Line 7), while S and the two rays are defined in the second iteration (lines 9-10). E is the incoming point in the third iteration, where line segment (A, E) is defined in Line 12. The slope of this segment is between the slopes of *SMin* and *SMax*, i.e. E lies within the cone determined by the two rays, so the condition in Line 13 evaluates to false (we assume that $|win| < w$), i.e. *win* is not complete. Thus, lines 18-21 are executed: S is made equal to E, and it is added to *win* (Line 18), and *SMin* and *SMax* are updated (lines 19-21). The updated information is shown in Figure 3.37.

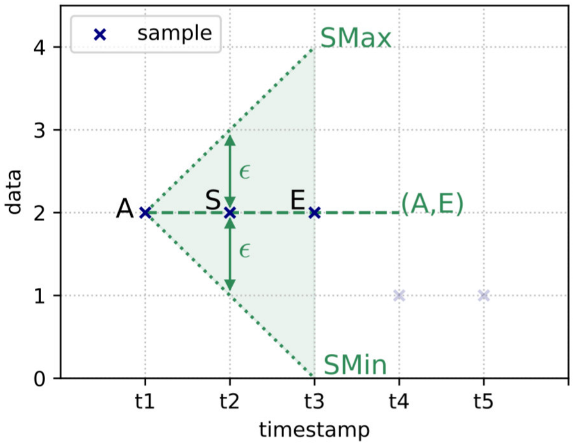

FIGURE 3.36: Example: key steps of the coding algorithm. Variant CA$_M$ with $\epsilon = 1$. Step 1.

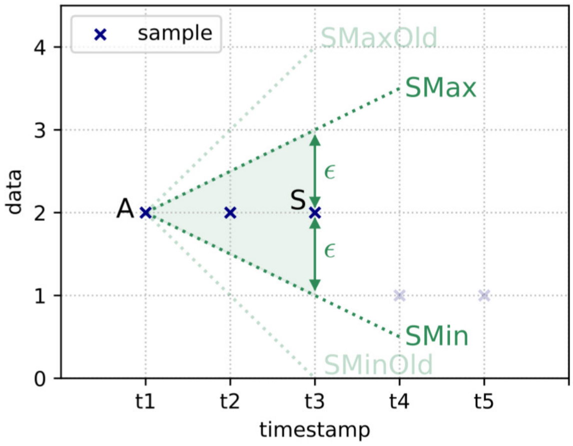

FIGURE 3.37: Example: key steps of the coding algorithm. Variant CA$_M$ with $\epsilon = 1$. Step 2.

In the successive iterations, the angle of the cone keeps decreasing, until eventually either point E lies outside of the cone, or *win* reaches its maximum size. When this occurs, *win* is complete, and the condition in Line 13 becomes true, so the size and the last point of *win*, i.e. point S, are encoded (Line 14), *win* is emptied (Line 15), and the process starts again, by defining and encoding a new point A (Line 16). We point out that an iteration in which point A is updated is always followed by an iteration in which *SMin* and *SMax* are updated. Also, notice that point A, and rays *SMin* and *SMax*, are always initialized in the first two iterations, respectively, so the cone is properly defined every time the condition in Line 13 is checked.

The coding routine for variant M invokes two auxiliary routines, EncodeArchivedPoint and EncodeWindow, which are presented in figures 3.38 and 3.39, respectively. Notice that, since the x-coordinates for both point A and the last point of *win* are already encoded in the timestamp column (recall Line 4 in Figure 3.1), it suffices to encode their respective y-coordinates. In routine EncodeWindow, the size of *win* must be encoded so that the decoder is able to figure out which timestamp (x-coordinate) corresponds to its last point.

```
input  : $A$: archived point to encode
         $out$: binary file encoded with algorithm variant CA$_M$
Encode the y-coordinate of point $A$ using $B_c$ bits
```

FIGURE 3.38: Auxiliary routine EncodeArchivedPoint for algorithm variant CA$_M$.

```
input  : $win$: window to encode
         $out$: binary file encoded with algorithm variant CA$_M$
         $w$: maximum window size
Encode the y-coordinate of the last point in $win$ using $B_c$ bits
Encode $|win|$ using $\lceil \log_2 w \rceil$ bits
```

FIGURE 3.39: Auxiliary routine EncodeWindow for algorithm variant CA$_M$.

The decoding routine for variant M is shown in Figure 3.40. It consists of a loop that keeps running until every entry in the column has been decoded, which occurs when condition in Line 2 becomes false. In each iteration, a point is obtained, decoding its y-coordinate, *value*, from the binary file, and reading its x-coordinate from the timestamp column (lines 3-4). If *decode_point_A* is true, *value* is the result of an encoding produced by routine EncodeArchivedPoint, so it corresponds to the y-coordinate of an archived point (A). In this case, point A is saved and *value* is written to the decoded CSV data file (lines 6-7). On the other hand, if *decode_point_A* is false, *value* comes from an encoding produced by routine EncodeWindow, so it corresponds to the y-coordinate of a snapshot point (S), whose x-coordinate is obtained from the timestamp column given the decoded window *size* (lines 11-12). In this case, the points in the window are modeled by a line segment, whose endpoints are A and S, so the auxiliary routine FillSegment is called with their respective coordinates as inputs (Line 13). Routine FillSegment (recall Figure 3.17) returns a list consisting of the (integer) sample values that are decoded from the segment, which are then written in the decoded CSV data file (lines 14-16).

```
input  : $in$: binary file encoded with algorithm variant CA$_M$
         $out$: decoded column of CSV data file
         $w$: maximum window size
         $col\_size$: number of entries in the column
         $t\_col$: timestamp column
 1 Let $n = 0$, and $decode\_point\_A$ = true
 2 while $n < col\_size$ do
 3     Decode $value$ using $B_c$ bits
 4     Obtain timestamp for $value$, $t_{value}$, from $t\_col$
 5     if $decode\_point\_A$ then
 6         Let $A$ be the point with coordinates ($t_{value}$, $value$)
 7         Output $value$ to $out$
 8         Let $size = 1$
 9         Let $decode\_point\_A$ = false
10     else
11         Decode $size$ using $\lceil \log_2 w \rceil$ bits
12         Let $S$ be the point with coordinates ($t_{value}$, $value$)
13         $samples$ = FillSegment($t\_col$, $A.x$, $S.x$, $A.y$, $S.y$) // routine in Figure 3.17
14         foreach sample in $samples$, $value$, do
15             Output $value$ to $out$
16         end
17         Let $decode\_point\_A$ = true
18     end
19     $n$ += $size$
20 end
```

FIGURE 3.40: Decoding routine for algorithm variant CA$_M$.

3.9.1 Example

Next we present an example of the encoding of the same 12 samples of previous examples, presented in Figure 3.6, and also shown in Figure 3.41 below. For this example we let the interval between consecutive timestamps be equal to 60, the error threshold parameter (ϵ) equal to 1, and the maximum window size (w) equal to 256.

This example involves encoding 12 samples, so no window can reach the maximum size (256). Therefore, an incoming point (E) is added to the current window, *win*, as long as it lies within the cone determined by the two rays departing from the archived point (A).

In the first iteration, the condition in Line 6 is satisfied, so A is made equal to *E1*, and it is encoded via the auxiliary routine EncodeArchivedPoint, using B_c bits for encoding its y-coordinate (i.e. 1). In the second iteration, *win* is empty, so the condition in Line 8 is satisfied. Therefore, the snapshot point (S) is made equal to *E2*, it is added to *win*, and two rays, *SMin* and *SMax*, are defined. Figure 3.41 shows both saved points, A and S, as well as both rays, *SMin* and *SMax*, after the second iteration is completed.

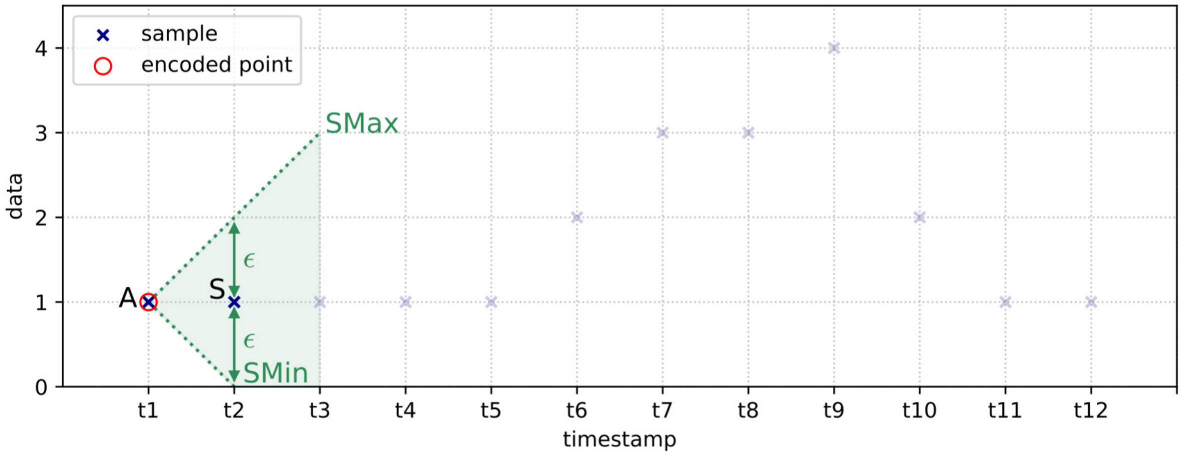

FIGURE 3.41: Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 1.

In the third iteration, another sample value equal to 1 is processed, but in this case *win* is not empty. In Figure 3.42, point *E3*, as well as segment (A,*E3*), defined in Line 11, is shown.

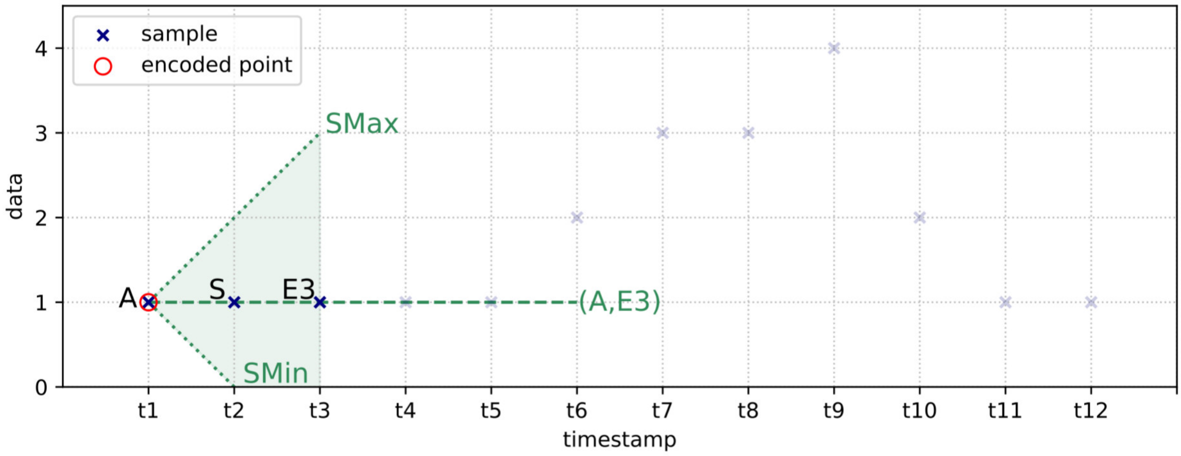

FIGURE 3.42: Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 2.

Since *E3* lies within the cone determined by the two rays departing from *A*, the condition in Line 12 evaluates to false. Therefore, *S* is made equal to *E3*, it is added to *win*, and *SMin* and *SMax* are updated (lines 18-21). Figure 3.43 shows the information after the third iteration is completed.

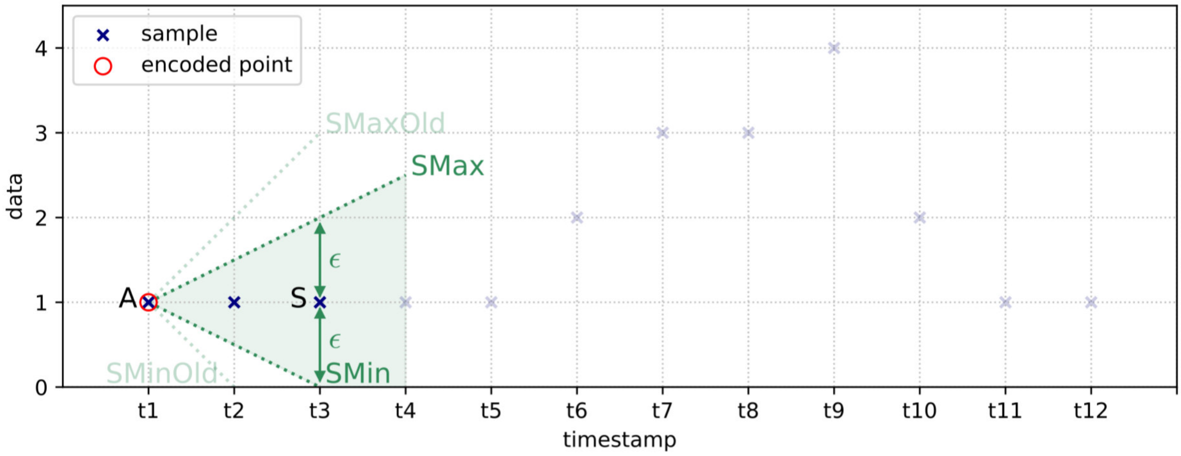

FIGURE 3.43: Example: Algorithm variant CA_M with $\epsilon = 1$ and $w = 256$. Step 3.

The following two iterations are similar to the previous one. In each iteration, the respective incoming point is added to the window, and both rays, *SMin* and *SMax*, are updated, decreasing the angle of the cone. Figure 3.44 shows the information after the 5th iteration is completed.

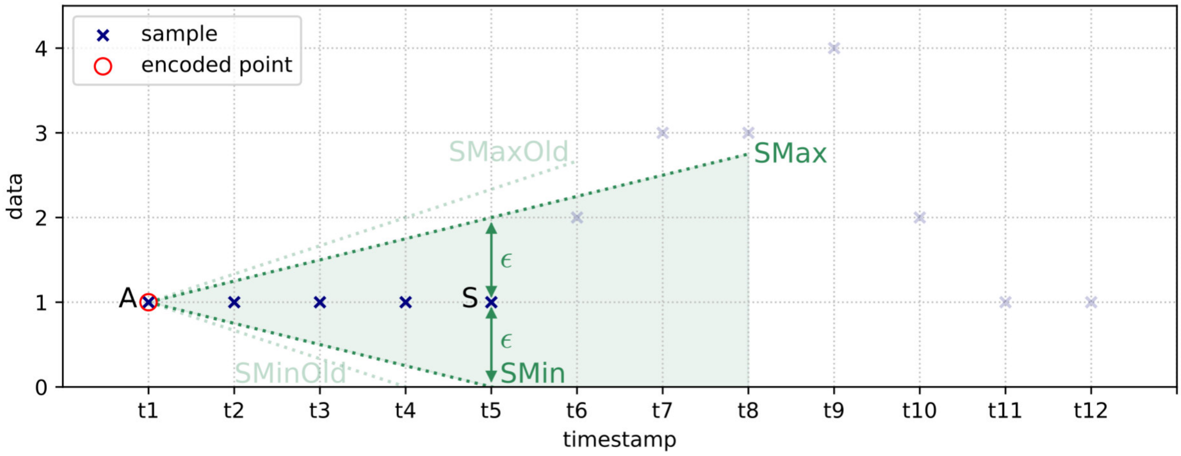

FIGURE 3.44: Example: Algorithm variant CA_M with $\epsilon = 1$ and $w = 256$. Step 4.

In the 6th iteration, sample value 2 is processed. In Figure 3.45, point *E6* and segment (*A*,*E6*) are shown.

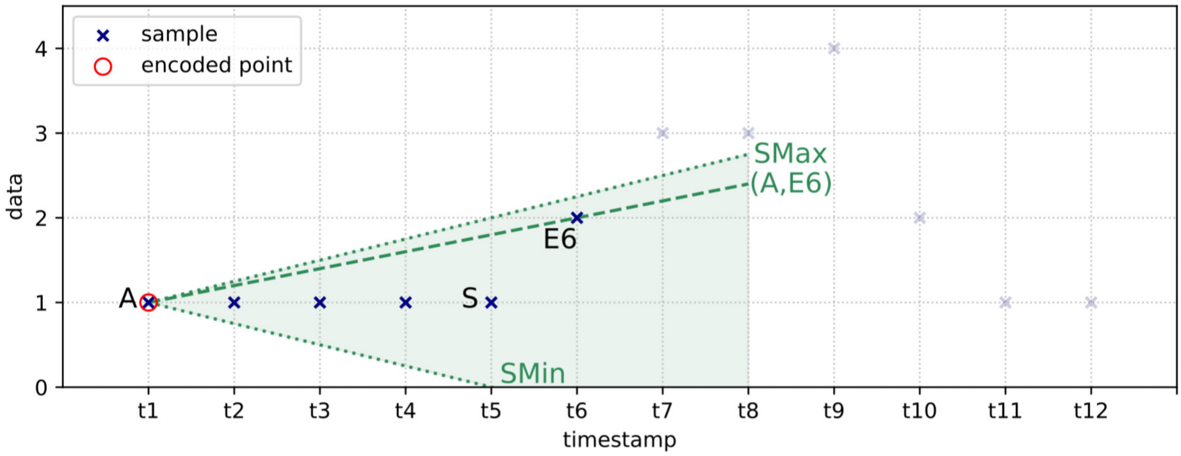

FIGURE 3.45: Example: Algorithm variant CA_M with $\epsilon = 1$ and $w = 256$. Step 5.

E6 lies within the cone, so the condition in Line 13 evaluates to false. Therefore, *S* is made equal to *E6*, and it is added to *win*. In this case, *SMin* is updated, but *SMax* remains unchanged. Figure 3.46 shows the information after the 6th iteration is completed.

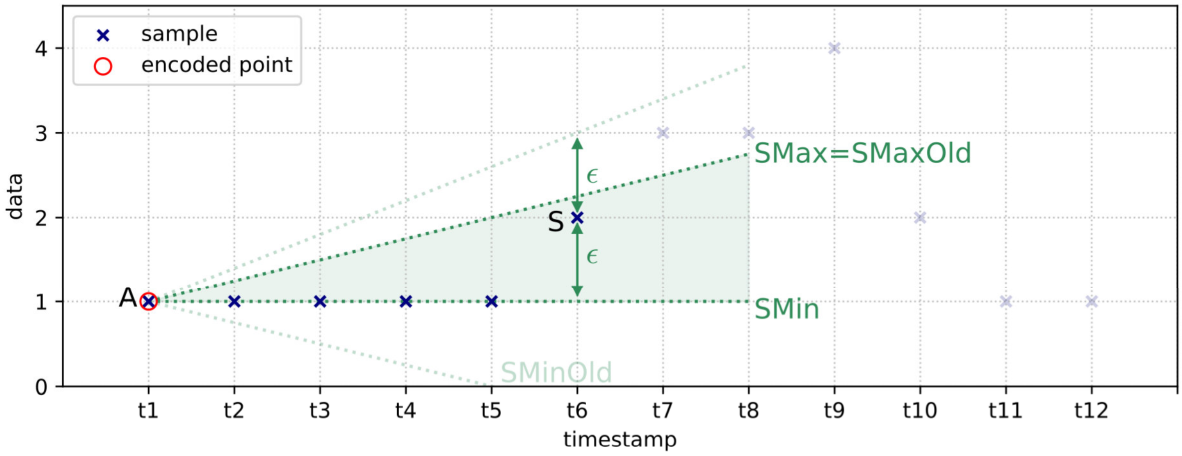

FIGURE 3.46: Example: Algorithm variant CA_M with $\epsilon = 1$ and $w = 256$. Step 6.

In the 7th iteration, sample value 3 is processed. In Figure 3.47, point *E7* and segment (*A*,*E7*) are shown.

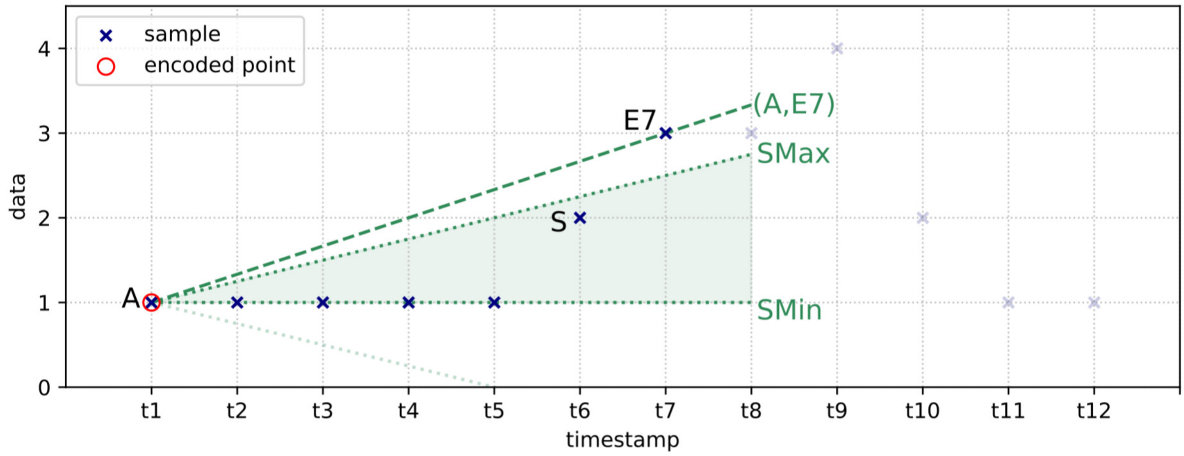

FIGURE 3.47: Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 7.

For the first time, an incoming point, *E7*, lies outside of the cone. In this case, (*A*,*E7*) has a greater slope than *SMax*, so the condition in Line 13 becomes true. Therefore, auxiliary routine EncodeWindow is invoked, which uses $\lceil \log_2 w \rceil = 8$ bits for encoding the size (5) of *win*, and B_c bits for encoding the y-coordinate (2) of *S*, its last point. Next, *win* is emptied, *A* is made equal to *E7*, and it is encoded via the routine EncodeArchivedPoint, using B_c bits for encoding its y-coordinate (i.e. 3). In the 8th iteration, *win* is empty, so the condition in Line 7 is satisfied. Thus, *S* is made equal to *E8*, it is added to *win*, and *SMin* and *SMax* are defined once more. Figure 3.48 shows the information after the 8th iteration is completed.

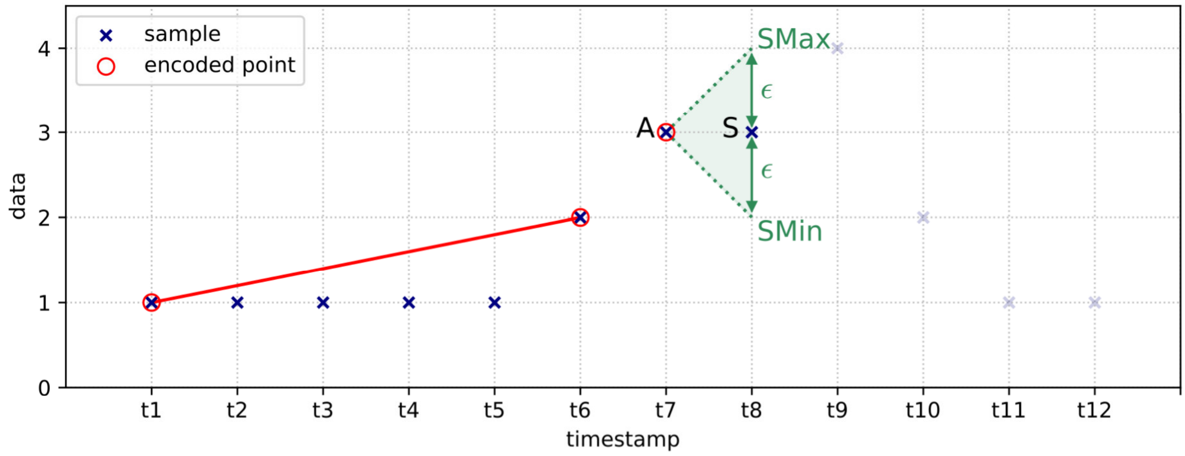

FIGURE 3.48: Example: Algorithm variant CA$_M$ with $\epsilon = 1$ and $w = 256$. Step 8.

In Figure 3.49 we present the information after the coding routine has finished. After the last iteration of the loop is executed, *win* is not empty, so routine EncodeWindow is invoked in Line 26. Besides showing the encoded points and their associated line segments, in Figure 3.49 we also display the values of the decoded samples, which are the values that the decoding routine, shown in Figure 3.40, writes to the decoded CSV data file. Overall, each of the routines EncodeArchivedPoint and EncodeWindow is invoked three times in this example. Thus, a total of $3 * B_c + 3 * (\lceil \log_2 w \rceil + B_c) = 24 + 6 * B_c$ bits are required for encoding the 12 samples.

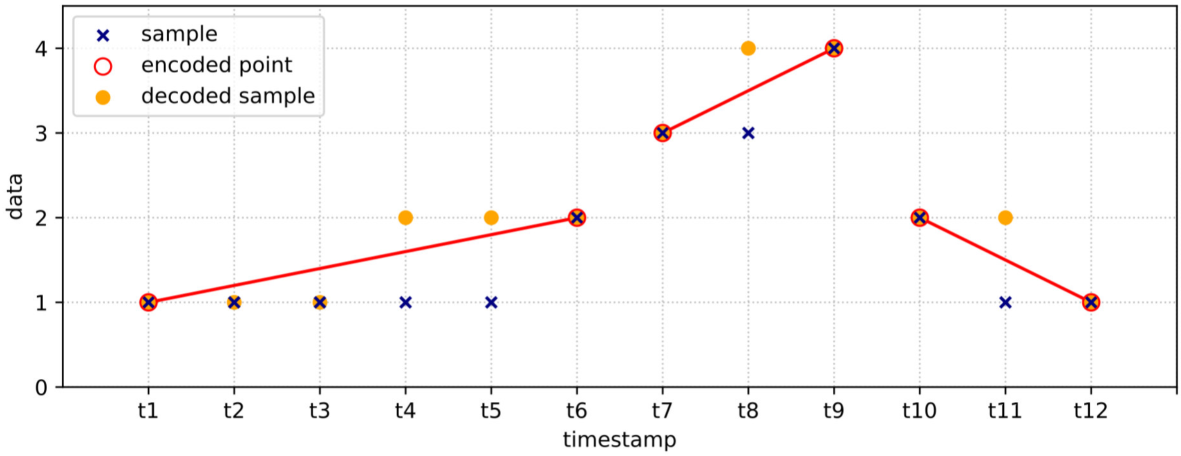

FIGURE 3.49: Example: Algorithm variant CA_M with $\epsilon = 1$ and $w = 256$. Step 9.

3.9.2 Non-Masking (*NM*) Variant

The coding and decoding routines for variant *NM* of algorithm CA are similar to their variant *M* counterparts, the difference being that the former routines are able to handle both sample values and gaps. Recall that, in the coding routine for variant *M*, a window is encoded when point E lies outside of the cone determined by the two rays departing from point A, or when the maximum window size is reached (Line 13 in Figure 3.35). The coding routine for variant *NM* must also encode a window if the incoming column entry is character "N" (gap in the data) and the other entries in the window are sample points, or vice versa. We point out that, in the former case, it doesn't make sense defining point A, so Line 16 is not executed in this scenario. For the same reason, Line 7 should not be executed if the first column entry is character "N". Auxiliary routine EncodeWindow uses the same number of bits for encoding a window that consists of gaps than for encoding a window that consists of sample points, i.e. $\lceil \log_2 w \rceil + B_c$, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and B_c bits are used for encoding the special integer *NO_DATA*. We point out that, in variant *NM*, every gap is encoded via auxiliary routine EncodeWindow; this is true even for a single gap burst appearing between two sample values, in which case the corresponding encoded window size is one.

3.10 Algorithm SF

Algorithm *Slide Filter (SF)* [38] is a linear model algorithm that supports lossless and near-lossless compression. For SF we define a single variant, *M*. Since SF is a linear model algorithm, its encoding process involves the encoding of a sequence of line segments. In Figure 3.50 we show the coding routine for variant *M*, in which all the column entries are integer values (the gaps are encoded separately). It consists of a loop that iterates over all column entries, parsing them into consecutive windows of variable size (up to a maximum size w), such that all the sample points in a window lie within vertical distance ϵ from the segment used to approximate said points. An important feature of algorithm SF is that, given two subsequent windows, it allows for the two segments approximating the sample points in each respective window, to be connected. In fact, this algorithm prioritizes finding such connected segments, over finding disconnected segments that minimize the per-sample absolute error between the original and the decoded samples in each window. We point out that, compared to the rest of the evaluated linear model algorithms, the fact that algorithm SF allows for subsequent line segments to be connected, affects its compression performance in opposite ways. On one hand, the total number of encoded endpoints is expected to be reduced, which improves it. On the other hand, more bits are required for encoding each endpoint (see details in the current section), which worsens it. There is a trade-off between these two factors, and algorithm SF is expected to outperform the rest of the evaluated linear algorithms in cases in which the effect of the former factor outweighs that of the latter one.

```
input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm variant $SF_M$
        $\epsilon$: maximum error threshold
        w: maximum window size
        t_col: timestamp column
Create an empty window, win
Let SMin and SMax be objects that will represent straight line rays
foreach entry y in column do
    Obtain timestamp for y, $t_y$, from t_col
    Let E be the point with coordinates ($t_y$, y)
    if y is the first entry in column then
        Append E to win, then continue
    else if $|win| == 1$ then
        Let S be the single point in win
        Let SMin be the ray with initial point $(S.x, S.y + \epsilon)$, through point $(E.x, E.y - \epsilon)$
        Let SMax be the ray with initial point $(S.x, S.y - \epsilon)$, through point $(E.x, E.y + \epsilon)$
        Append E to win, then continue
    end
    Let I be the point of intersection of SMin and SMax
    Let (I,E) be the segment with initial point I that passes through point E
    if not slope(SMin) $\leq$ slope((I,E)) $\leq$ slope(SMax) or $|win| == w$ then
        Let LS be the set of line segments such that $s \in LS$ iff s has initial point I and
          slope(SMin) $\leq$ slope(s) $\leq$ slope(SMax)
        if not $segment_{prev}$ exists then
            segment = EncodeWinStart(LS, win, t_col, out) // rout. in Fig.3.52
            Make $segment_{prev}$ = segment
        else
            Let $segment_{conn} \in LS$ be a line segment connected to $segment_{prev}$, which
              approximates the points in win
            if $segment_{conn}$ exists then
                EncodeWinEndStart($segment_{prev}$, $segment_{conn}$, out) // rout. in Fig. 3.53
                Make $segment_{prev}$ = $segment_{conn}$
            else
                EncodeWinEnd($segment_{prev}$, out)// routine shown in Figure 3.54
                segment = EncodeWinStart(LS, win, t_col, out) // rout. in Fig.3.52
                Make $segment_{prev}$ = segment
            end
        end
        Set win to an empty window, then append E to win
    else
        Append E to win
        Update SMin and SMax, considering the points in win
    end
end
```

FIGURE 3.50: Coding routine for algorithm variant SF_M.

Another key difference with the rest of the implemented linear algorithms, is that in SF the x-coordinate for the endpoint of an encoded line segment does not necessarily coincide with a timestamp (integer) value. This allows for a larger universe of segments in which to search for connected segments. Therefore, the x-coordinates of the endpoints of every line segment are encoded as floats. The y-coordinates are also encoded as floats (recall that this is also the case in algorithm PWLH, presented in Section 3.8). An auxiliary method EncodePoint, presented in Figure 3.51, is used for encoding the endpoints of every segment. Observe that, besides the coordinates for both axis, a bit, labeled *connected*, which indicates whether the endpoint belongs to a connected segment or not, is output.

```
input  : P: point to encode
         out: binary file encoded with algorithm variant $SF_M$
         connected: boolean indicating if the segment is connected
Output bit connected to out
Encode $P.x$ as a float, using 32 bits
Encode $P.y$ as a float, using 32 bits
```

FIGURE 3.51: Auxiliary routine EncodePoint for algorithm variant SF_M.

The coding routine operates by successively growing the current window, *win*, adding one sample point at a time until it is complete. While *win* is being filled, the line segment that approximates the points in the previous window, $segment_{prev}$, is not yet fully encoded: its slope is known, and its first endpoint is encoded, but its last endpoint is not. Since the algorithm prioritizes finding a segment that is connected to $segment_{prev}$, to approximate the points in *win*, the last endpoint of $segment_{prev}$ is defined (and encoded) only after *win* is complete. If such connected segment, $segment_{conn}$, exists, i.e. the condition in Line 23 is satisfied, then the intersection point is encoded (using an auxiliary routine EncodeWinEndStart, shown in Figure 3.53). In this case, bit *connected* is encoded as 1, so that the decoding routine knows that this point is both the last endpoint of the segment approximating the points in the previous window, as well as the first endpoint of the segment approximating the points in *win*. On the other hand, if a connected segment cannot be found, then the last endpoint of $segment_{prev}$ is encoded (using an auxiliary routine EncodeWinEnd, shown in Figure 3.54), a new segment, *segment*, which approximates the points in *win*, is computed, and its first endpoint is encoded (using an auxiliary routine EncodeWinStart, shown in Figure 3.52). In this case, since $segment_{prev}$ is not connected, its last endpoint is encoded with bit *connected* as 0, while the first endpoint of *segment* is encoded with bit *connected* as 1. Whether a connected segment can be found or not, the segment that approximates the points in *win*, of which its slope is known and its first endpoint is encoded, is always saved as $segment_{prev}$ (lines 25 and 29), so that it is available in the following iterations, when the process of filling *win*, which is emptied and added the current sample point (Line 32), continues.

```
input  : LS: set of line segments
         win: window to encode
         t_col: timestamp column
         out: binary file encoded with algorithm variant $SF_M$
output: segment: line segment that approximates the points in win
Let $segment \in LS$ be the line segment that minimizes the MSE for the points in win
   (among the segments included in LS)
Let P be the first endpoint of segment
EncodePoint(P, out, true) // routine shown in Figure 3.51
return segment
```

FIGURE 3.52: Auxiliary routine EncodeWinStart for algorithm variant SF_M.

```
input : $segment_{prev}$: line segment that approximates the points in the previous window
        $segment$: line segment that approximates the points in $win$
        $out$: binary file encoded with algorithm variant SF$_M$
Let $P$ be the point of intersection of $segment_{prev}$ and $segment$
EncodePoint($P$, $out$, true) // routine shown in Figure 3.51
```

FIGURE 3.53: Auxiliary routine EncodeWinEndStart for algorithm variant SF$_M$.

```
input : $segment_{prev}$: line segment that approximates the points in the previous window
        $out$: binary file encoded with algorithm variant SF$_M$
Let $P$ be the last endpoint of $segment_{prev}$
EncodePoint($P$, $out$, false) // routine shown in Figure 3.51
```

FIGURE 3.54: Auxiliary routine EncodeWinEnd for algorithm variant SF$_M$.

The algorithm determines when to stop growing the window, i.e. when the current window *win* is complete, based upon two points: *incoming (E)* is the sample point corresponding to the column entry for the current iteration, and *intersection (I)* is the point of intersection of *SMin* and *SMax*, which are two rays that define a cone maintained by the algorithm, such that E is added to *win* as long as it lies within the cone. This cone, in turn, is defined so that if E is indeed added to *win*, all the points in *win* are at a vertical distance of at most ϵ from the line segment (I,E).

In figures 3.55 and 3.56 we present an example that illustrates the key steps of the coding algorithm. In Figure 3.55, the two points, E and I, as well as the two rays, *SMin* and *SMax*, are shown. In the first iteration, the incoming point is added to *win* (Line 7), while in the second iteration the two rays are defined (lines 9-11), and the incoming point is added to *win* (Line 12). E is the incoming point in the third iteration, where line segment (I, E) is defined in Line 15. The slope of this segment is between the slopes of *SMin* and *SMax*, i.e. E lies within the cone determined by the two rays, so the condition in Line 16 evaluates to false (we assume that $|win| < w$), i.e. *win* is not complete. Therefore, lines 34-35 are executed: E is added to *win* (Line 34), and *SMin* and *SMax* are updated (Line 35). The updated information is shown in Figure 3.56.

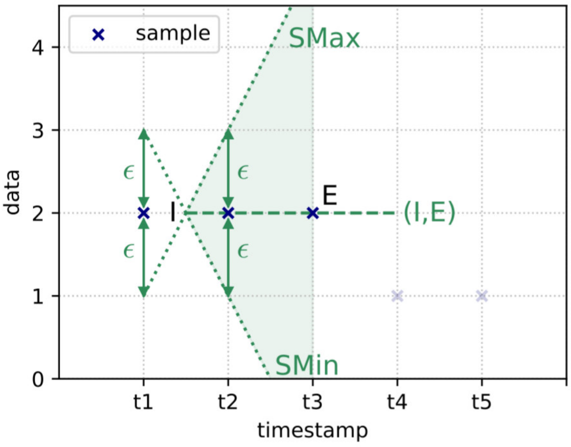

FIGURE 3.55: Example: key steps of the coding algorithm. Variant SF$_M$ with $\epsilon = 1$. Step 1.

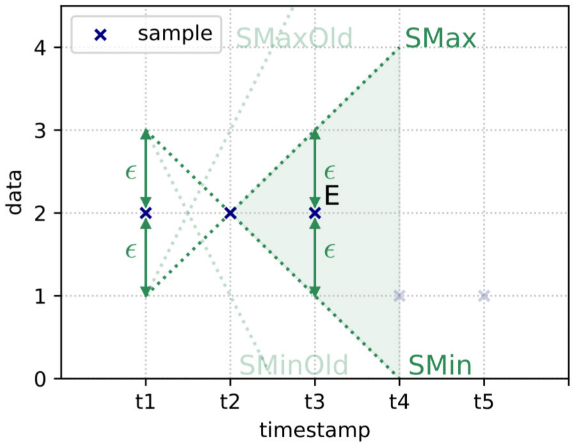

FIGURE 3.56: Example: key steps of the coding algorithm. Variant SF$_M$ with $\epsilon = 1$. Step 2.

In the successive iterations, the angle of the cone keeps decreasing, until eventually either point E lies outside of the cone, or *win* reaches its maximum size, making *win* complete. Recall that this process is similar in algorithm CA, presented in Section 3.9. The differences are that, in algorithm SF, the intersection point of the two rays, *SMin* and *SMax*, is not fixed (it increases its x-coordinate in the successive iterations), and the procedure for updating the two rays is more complex, since it always considers all the points in *win* [38]. We point out that in the first iteration *win* is added its first point, and every iteration that ends with *win* having a single point is followed by an iteration in which *SMin* and *SMax* are updated. Thus, the cone is properly define every time the condition in Line 16 is checked.

Auxiliary routine EncodeWinStart, shown in Figure 3.52, is invoked after *win* is complete, in two different scenarios. The first scenario is when *win* is the first completed window. In this case, no segment has yet been encoded, so the condition in Line 18 is satisfied, and EncodeWinStart is invoked in Line 19. The second scenario is when a connected segment approximating the points in *win* cannot be found. In this case, the condition in Line 23 evaluates to false, and auxiliary routines EncodeWinEnd and EncodeWinStart are invoked (lines 27-28). In both scenarios, the inputs for EncodeWinStart are the same: *LS* is the set of line segments defined in Line 17, which consists of every segment with initial point I that is within the cone; *win* is the current window; *t_col* is the timestamp column; *out* is the encoded binary file. Recall that routine EncodeWinStart computes a new segment, *segment*, which approximates the points in *win*, and then encodes its first endpoint. *segment* is the line segment that minimizes the MSE for the points in *win*, among the segments included in *LS*. *segment* must belong to *LS* because this guarantees that each point in *win* is at a vertical distance of at most ϵ from *segment.*

The decoding routine for variant M is shown in Figure 3.57. We point out that this is the only decoding routine for any of the implemented algorithms that doesn't have a window size parameter (w) input, which is unnecessary since the x-coordinates of the encoded points are encoded as floats (recall that the rest of the algorithms require parameter w in order to decode the variable size of each encoded window, reading $\lceil \log_2 w \rceil$ bits, or to know the fixed window size in the case of algorithm PCA). The decoding routine consists of a loop that repeats until every entry in the column has been decoded, which occurs when condition in Line 2 becomes false.

Each iteration of the loop starts by calling the auxiliary routine DecodePoint (Line 3), shown in Figure 3.58, which returns a decoded point, P, and the *connected* flag. This flag indicates whether the encoded segment to which the point belongs to is connected to the subsequent segment or not. In the first iteration, P is set to be the initial point of the segment, P_o (Line 5). On the other hand, in the rest of iterations, P is set to be the final point of the segment, P_f (Line 8), and the auxiliary routine FillSegment is called, with the coordinates of both endpoints of the segment as inputs (Line 9). As we recall from Figure 3.17, routine FillSegment returns a list consisting of the sample values that are decoded from the segment, which are then written in the decoded CSV data file (lines 10-12). We point out that FillSegment always returns a list of integers, even though in this case the x and y-coordinate inputs are floats. If the segment is connected, no additional point must be decoded, since in that case the first endpoint of the subsequent encoded segment is equal to the last endpoint of the last decoded segment (Line 14). Otherwise, if the segment is not connected, the first endpoint of the subsequent encoded segment must be decoded (Line 16).

```
input  : in: binary file encoded with algorithm variant $SF_M$
         out: decoded column of CSV data file
         col_size: number of entries in the column
         t_col: timestamp column
$n = 0$
while $n <$ col_size do
    P, connected = DecodePoint() // routine shown in Figure 3.58
    if $n == 0$ then
        Let $P_o = P$
        continue
    end
    Let $P_f = P$
    samples = FillSegment(t_col, $P_o.x$, $P_f.x$, $P_o.y$, $P_f.y$) // routine in Figure 3.17
    foreach sample in samples, value, do
        Output value to out
    end
    if connected then
        Let $P_o = P_f$
    else
        $P_o$, connected = DecodePoint() // routine shown in Figure 3.58
    end
    $n$ += |samples|
end
```

Figure 3.57: Decoding routine for algorithm variant SF_M.

```
output: P: decoded point
        connected: boolean indicating if the segment is connected
Decode connected using a single bit
Decode $x$ as a float, using 32 bits
Decode $y$ as a float, using 32 bits
Let $P$ be the point with coordinates $(x, y)$
return P, connected
```

Figure 3.58: Auxiliary routine DecodePoint for algorithm variant SF_M.

3.10.1 Example

Next we present an example of the encoding of the same 12 samples of previous examples, presented in Figure 3.6, and also shown in Figure 3.59 below. For this example we let the interval between consecutive timestamps be equal to 60, the error threshold parameter (ϵ) equal to 1, and the maximum window size (w) equal to 256.

Since there are only 12 samples to encode, no window in this example can reach the maximum size (256). Therefore, an incoming point (E) is added to the current window, *win*, as long as it lies within the cone determined by the two rays, *SMin* and *SMax*. In the first iteration, the condition in Line 6 is satisfied, so the first incoming point, *E1*, is added to *win* (Line 7). In the second iteration, since *win* has a single element, the condition in Line 8 is satisfied, so using points *S* and *E2*, and parameter ϵ, rays *SMin* and *SMax* are defined (lines 9-11). Both rays are shown in Figure 3.59. *E2* is also added to *win* (Line 12).

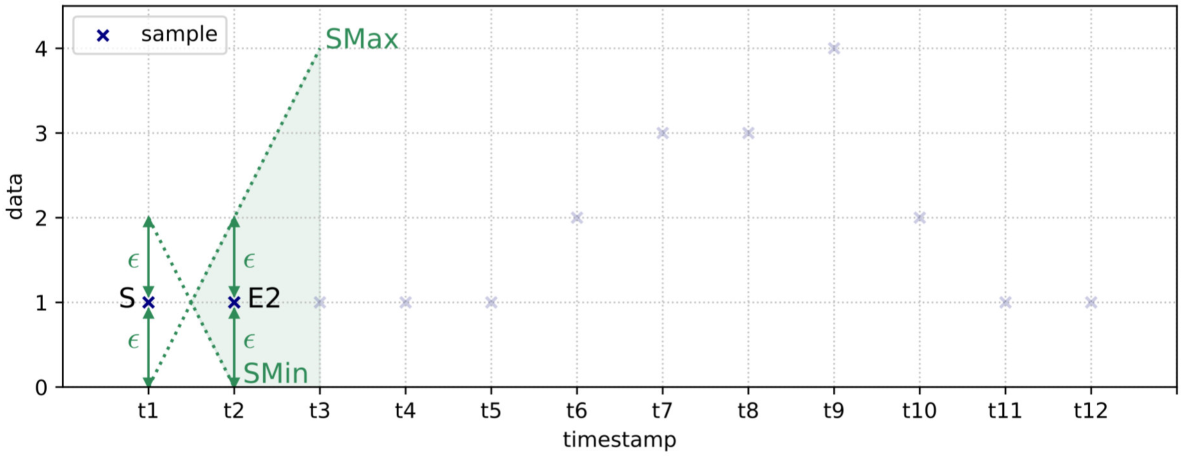

FIGURE 3.59: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 1.

In the third iteration, another sample value equal to 1 is processed. In Figure 3.60, point *E3*, as well as segment (*I*, *E3*), defined in lines 14-15, is shown.

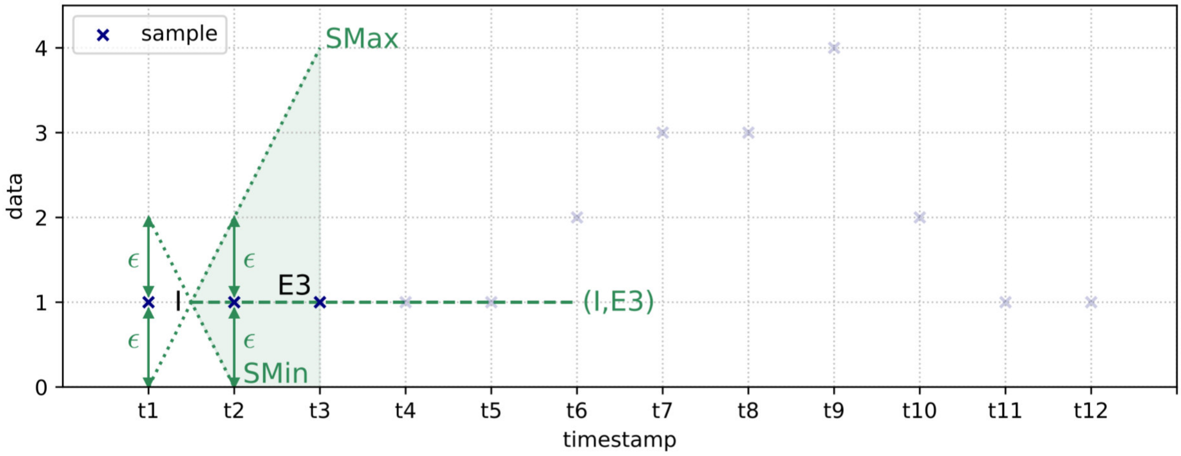

FIGURE 3.60: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 2.

Since *E3* lies within the cone determined by the two rays, the condition in Line 16 evaluates to false. Therefore, *E3* is added to *win* (Line 34), and *SMin* and *SMax* are updated (Line 35). Figure 3.61 shows the information after the third iteration is completed.

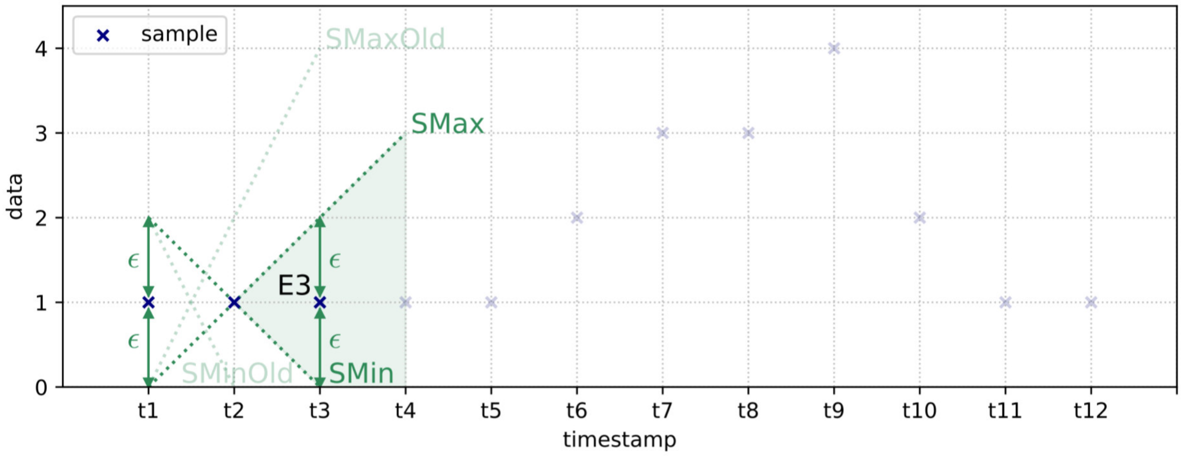

FIGURE 3.61: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 3.

The following six iterations are similar to the previous one. In each iteration, the respective incoming point is added to the window, and both rays are updated, decreasing the angle of the cone. Figure 3.62 shows the information after the 9th iteration is completed. Up to this point, *win* includes the first 9 sample points, and no bits have yet been written in the binary file.

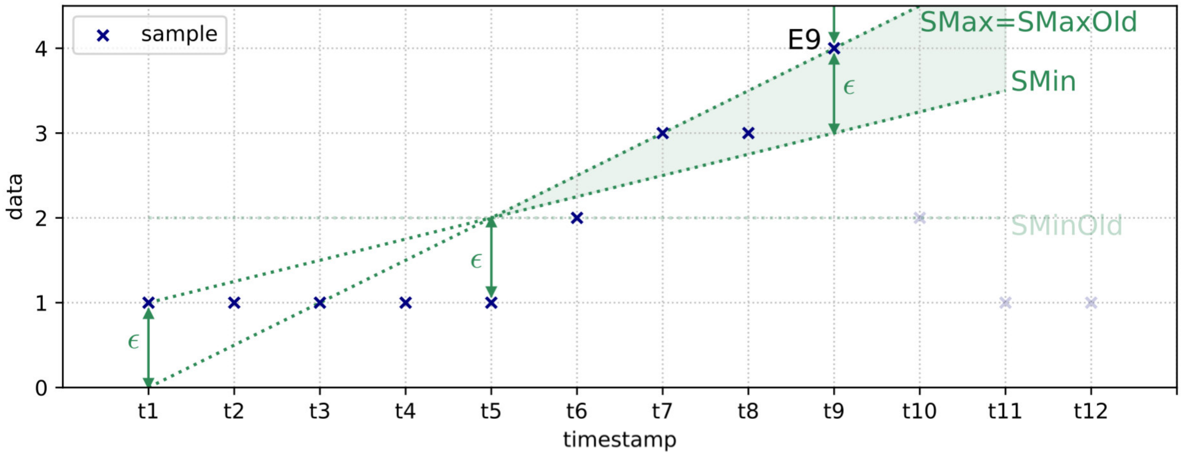

FIGURE 3.62: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 4.

Eventually, in the 10th iteration, sample value 2 is processed. In Figure 3.63, point *E10* and segment (*I*,*E10*) are shown.

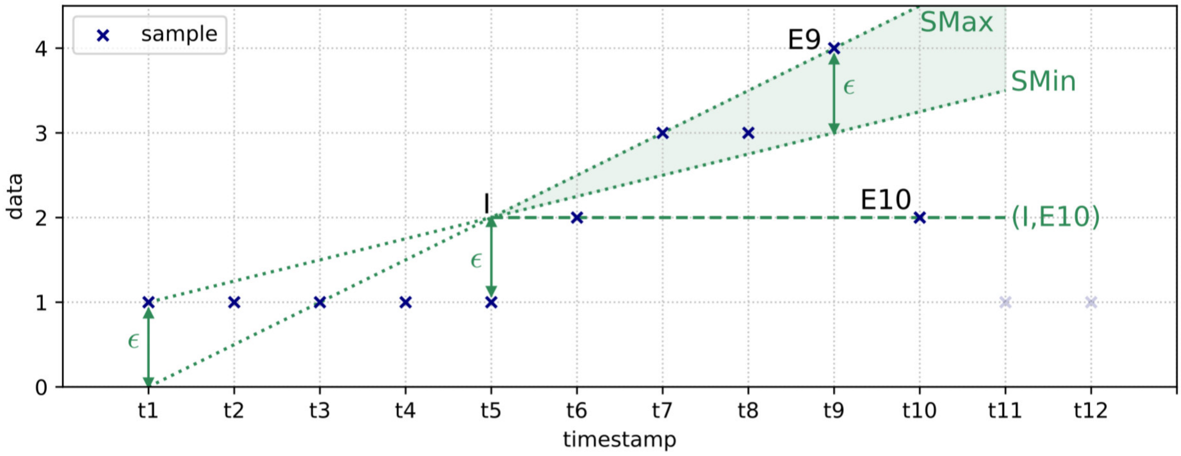

FIGURE 3.63: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 5.

For the first time, an incoming point, *E10*, lies outside of the cone. In this case, (*I*,*E10*) has a smaller slope than *SMin*, so the condition in Line 16 becomes true. Since no segment has yet been encoded, the condition in Line 18 is satisfied, and auxiliary routine EncodeWinStart is invoked, with one of the arguments being *LS*, which is a set of line segments defined in Line 17. Routine EncodeWinStart finds the segment included in *LS* that minimizes the MSE for the 9 sample points in *win* (among the segments included in *LS*), and encodes its first endpoint via EncodePoint, using a single bit for encoding the *connected* flag, and 32 bits for encoding each coordinate. In Figure 3.64 the segment and its encoded endpoint are shown. Observe that, since this segment is included in set *LS*, it passes through point *I*, and its slope is between the slopes of rays *SMin* and *SMax*. The point encoded next, to be determined in a future iteration, must also belong to this segment, and it will also belong to the subsequent segment if an appropriate connected segment can be found. Finally, the segment is saved as $segment_{prev}$ (Line 20), and *w* is emptied and added *E10* (Line 32).

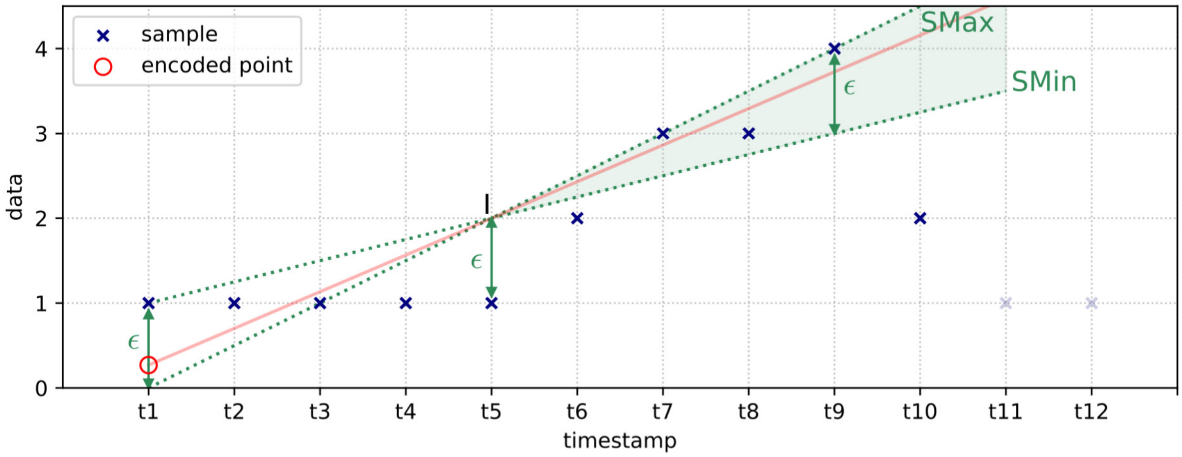

FIGURE 3.64: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 6.

In the 11th iteration, since *win* has a single element, the condition in Line 8 is satisfied, so using points *S* and *E11*, and parameter ϵ, rays *SMin* and *SMax* are defined (lines 9-11). Both rays are shown in Figure 3.65. *E11* is added to *win* (Line 12).

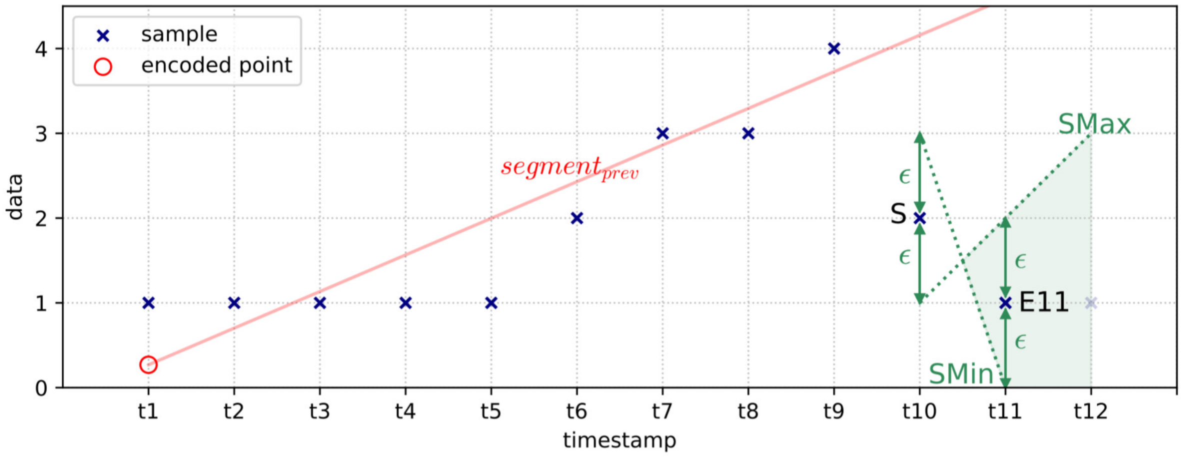

FIGURE 3.65: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 7.

In the 12th and last iteration, another sample value equal to 1 is processed. In Figure 3.66, point *E12*, as well as segment (*I*,*E12*), defined in lines 14-15, is shown.

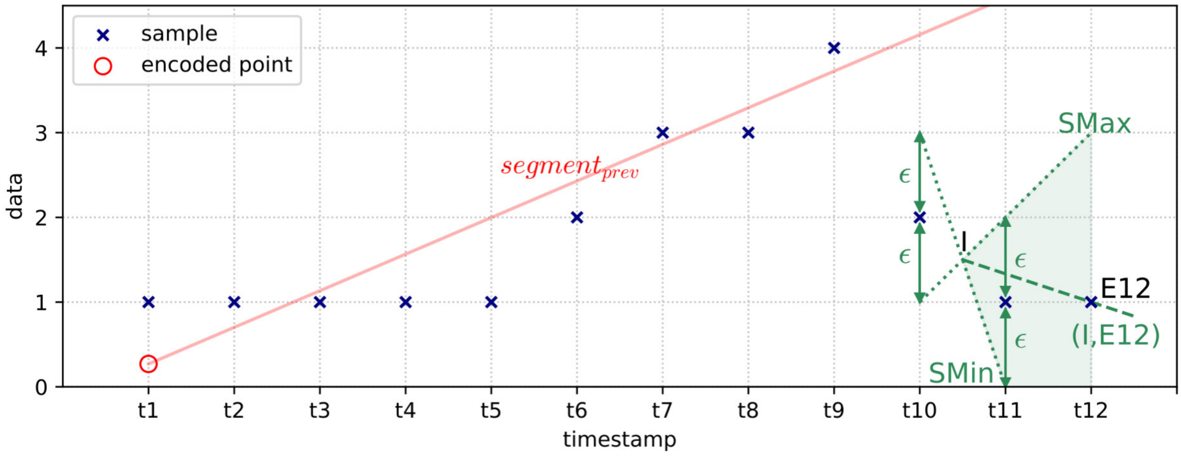

FIGURE 3.66: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 8.

Since *E12* lies within the cone determined by the two rays, the condition in Line 16 evaluates to false. Therefore, *E12* is added to *win* (Line 34), and *SMin* and *SMax* are updated (Line 35). Figure 3.67 shows the information after the 12th iteration is completed.

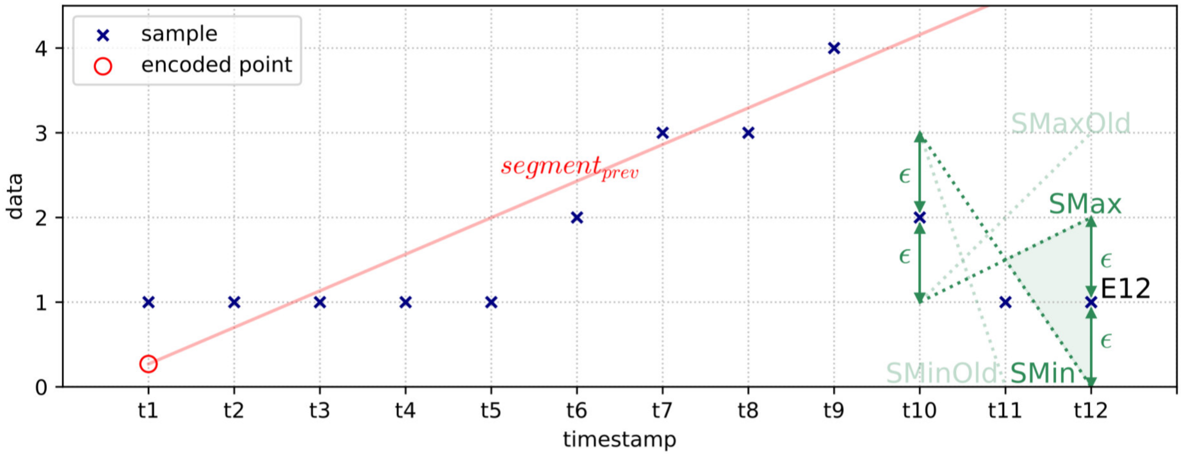

FIGURE 3.67: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 9.

After executing the last iteration of the loop, *win* is not empty (it consists of the last three sample points), so a segment approximating the points in *win* must be encoded, executing the same code as in lines 17-31 (this was left out of the coding routine in Figure 3.50 for clarity). In this case, the algorithm finds a connected segment, $segment_{conn}$, included in set *LS*, so the condition in Line 23 is satisfied, and the intersection point between $segment_{prev}$ and $segment_{conn}$ is encoded by the auxiliary routine EncodeWinEndStart, shown in Figure 3.53. Since the last sample is already processed, the last endpoint of the connected segment must also be encoded, in this case by the auxiliary routine EncodeWinEnd, shown in Figure 3.54 (this step was also left out of the coding routine for clarity). In Figure 3.68, the three encoded points and the two connected segments are shown. The three points are encoded with auxiliary method EncodePoint, in the first two cases with bit *connected* as 1, and in the last case with bit *connected* as 0. Thus, a total of $3 * (1 + 2 * 32) = 195$ bits are required for encoding the 12 samples. We point out that, even though in this example the x-coordinate of the intersection endpoint coincides with a timestamp (integer) value, this is not necessarily always the case.

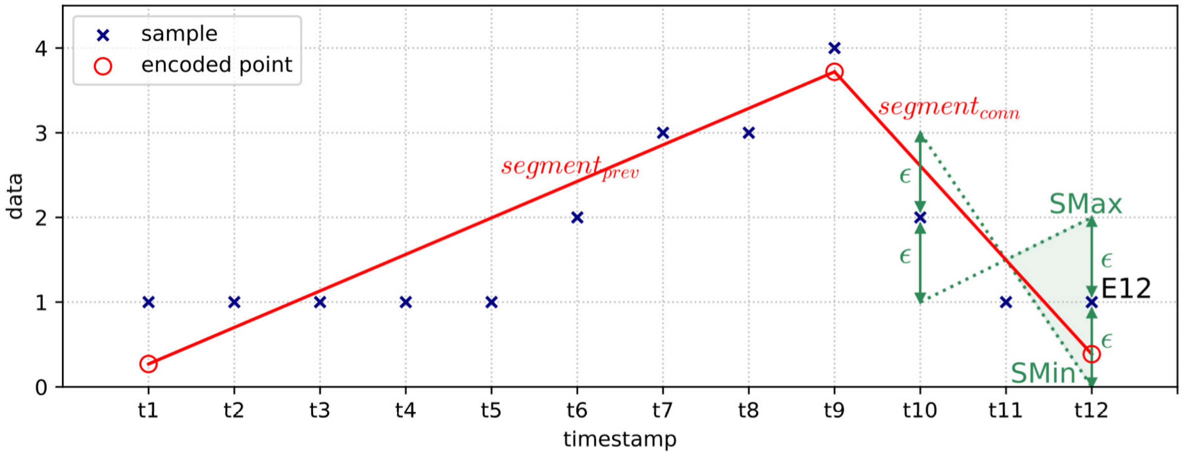

FIGURE 3.68: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 10.

Finally, in Figure 3.69 we also display the values of the decoded samples, which are the values that the decoding routine, shown in Figure 3.57, writes to the decoded CSV data file.

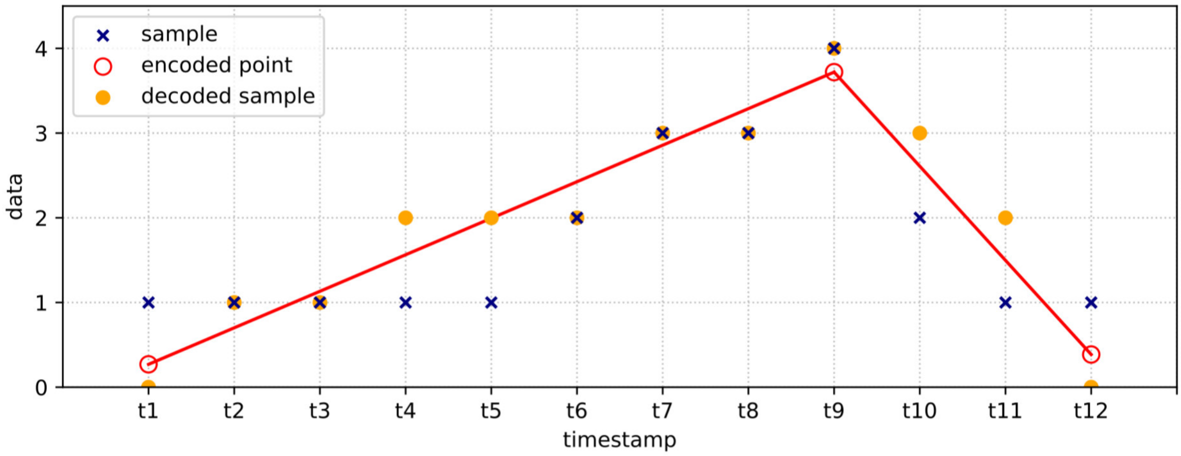

FIGURE 3.69: Example: Algorithm variant SF_M with $\epsilon = 1$ and $w = 256$. Step 11.

3.11 Algorithm FR

Algorithm *Fractal Resampling (FR)* [39] is a linear model algorithm that supports lossless and near-lossless compression. Its computational complexity is fairly low, since it was designed by the European Space Association (ESA) to be run on spacecraft and planetary probes. For FR we define a single variant, M.

FR is a linear model algorithm, so its encoding process involves encoding a sequence of line segments. In Figure 3.70 we show the coding routine for variant M, in which all the column entries are integer values (the gaps are encoded separately). The column entries are parsed into consecutive non-overlapping windows of size w (Line 1). The sample points in each window are modeled by a total of between one and $w - 1$ connected line segments, whose respective endpoints, called the *displaced points*, are obtained using a simple recursive technique called *mid-point displacement (MPD)*. In each iteration, the set of displaced points obtained for the window, *win*, is a subset of the sample points in *win*.

Since parameter w is always greater than 1, the condition in Line 3 might only be satisfied in the last iteration, which corresponds to the last window. In the rest of the cases, an empty list, *displaced_points*, is created (Line 7), and passed to a recursive auxiliary routine GetDisPointsMDP (Line 8). This routine fills the list with the indexes (relative to *win*, in ascending order) corresponding to the displaced points obtained for *win*. The first and last entries in *displaced_points* are always 0 and $w - 1$, respectively, since the first endpoint of the first segment is always the first point in *win*, and the last endpoint of the last segment is always the last point in *win*. In lines 9-12, each of the displaced points obtained for *win* is encoded, using $\lceil \log_2 w \rceil$ bits for encoding its index (relative to *win*), *index*, and B_c bits for encoding its y-coordinate.

```
input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm variant $FR_M$
        $\epsilon$: maximum error threshold
        w: maximum window size
        t_col: timestamp column
Parse column into consecutive non-overlapping windows of size w, except possibly for
  the last window that may consist of fewer sample points
foreach window in the parsing, win, do
    if $|win| == 1$ then
        Encode the y-coordinate of the single point in win using $B_c$ bits
        return
    end
    Create an empty list, displaced_points
    GetDisPointsMDP($win, \epsilon$, displaced_points, $0, |win| - 1$) // routine in Fig. 3.71
    foreach entry in displaced_points, index, do
        Encode index using $\lceil \log_2 w \rceil$ bits
        Encode the y-coordinate of point win[index] using $B_c$ bits
    end
end
```

FIGURE 3.70: Coding routine for algorithm variant FR_M.

In Figure 3.71 we present the recursive routine GetDisPointsMDP. In each execution, a line segment, with endpoints P_o and P_f, defined in lines 3-4, is evaluated as a candidate to model all the points in *win* whose index is between i_o and i_f. If the segment satisfies the *valid segment condition*, defined in Line 5, the absolute error between the encoded and the original samples corresponding to that segment is guaranteed to be less than or equal to ϵ, so no further steps are required. If that is not the case, two recursive calls are made to the routine (lines 8-9). In each call, a line segment is evaluated as a candidate to model a certain set of points in *win*. Notice that the segments evaluated in the first and in the second call are connected by the middle point (see *half* definition in Line 7). This is what gives the MPD technique its name.

```
input : win: window to encode
        $\epsilon$: maximum error threshold
        displaced_points: list including the indexes (relative to win, in ascending order) of
the respective displaced points
        $i_o$, $i_f$: indexes (relative to win) of the endpoints of the candidate line segment
1  If they are not already included, add $i_o$ and $i_f$ to displaced_points, in ascending order
2  If $i_o + 1 \geq i_f$ then return // base case
3  Let $P_o$= win[$i_o$], and let $P_f$= win[$i_f$]
4  Let segment be the line segment whose endpoints are $P_o$ and $P_f$
5  Let valid_segment be true iff for every point $P_i$ in win, the vertical distance between
     segment and $P_i$ is less than or equal to $\epsilon$
6  if not valid_segment then
7      Let half = $\lfloor (i_o + i_f)/2 \rfloor$
8      Recursively call GetDisPointsMDP(win, $\epsilon$, displaced_points, $i_o$, half)
9      Recursively call GetDisPointsMDP(win, $\epsilon$, displaced_points, half, $i_f$)
10 end
```

FIGURE 3.71: Auxiliary routine GetDisPointsMDP for algorithm variant FR_M.

The decoding routine for variant M is shown in Figure 3.72. It consists of a loop that keeps running until every entry in the column has been decoded, which occurs when condition in Line 2 becomes false. In each iteration of the loop, a window is decoded (lines 3-26). Since w is always greater than 1, the condition in Line 4 may only be satisfied in the last iteration (notice that lines 4-8 are correlated with lines 3-6 in the coding routine). In the rest of the cases, the size of the encoded window is always equal or greater than 2, so the data samples are decoded by means of obtaining the endpoints of the one or more line segments that model the points in a window. The first endpoint of the first segment is decoded in lines 9-12, while the last endpoint of each segment, which later becomes the first endpoint of each respective subsequent segment (Line 24), is decoded in lines 14-17. Next, auxiliary routine FillSegment is called, with the coordinates of both endpoints of the segment as inputs (Line 18). Routine FillSegment (recall Figure 3.17) returns a list consisting of the (integer) sample values that are decoded from the segment, which are then written in the decoded CSV data file (lines 19-23). The conditional statement in Line 20 is meant to avoid writing the sample value associated to the last endpoint of a segment, for every segment that models the points in the window, except in the case of the last segment. Otherwise, those sample values would be written in the decoded data file twice, since in each window, the last endpoint of a segment always coincides with the first endpoint of the subsequent segment.

```
input : in: binary file encoded with algorithm variant $FR_M$
        out: decoded column of CSV data file
        w: maximum window size
        col_size: number of entries in the column
        t_col: timestamp column
1  $n = 0$
2  while $n <$ col_size do
3      size = min{w, col_size − n}
4      if size == 1 then
5          Decode value using $B_c$ bits
6          Output value to out
7          return
8      end
9      Decode index using $\lceil \log_2 w \rceil$ bits
10     Decode value using $B_c$ bits
11     Obtain timestamp for value, $t_{value}$, from t_col
12     Let $P_o$ be the point with coordinates ($t_{value}$, value)
13     while index < size do
14         Decode index using $\lceil \log_2 w \rceil$ bits
15         Decode value using $B_c$ bits
16         Obtain timestamp for value, $t_{value}$, from t_col
17         Let $P_f$ be the point with coordinates ($t_{value}$, value)
18         samples = FillSegment(t_col, $P_o.x$, $P_f.x$, $P_o.y$, $P_f.y$) // routine in Figure 3.17
19         foreach sample in samples, value, do
20             if value is not the last sample in samples or index == size then
21                 Output value to out
22             end
23         end
24         Let $P_o$=$P_f$
25     end
26     n += size
27 end
```

Figure 3.72: Decoding routine for algorithm variant FR_M.

3.11.1 Example

Next we present an example of the encoding of the same 12 samples of previous examples, presented in Figure 3.6, and also shown in Figure 3.73 below. For this example we let the interval between consecutive timestamps be equal to 60, the error threshold parameter (ϵ) equal to 1, and the maximum window size (w) equal to 256.

Since there are only 12 samples to encode, a single window, *win*, including the 12 sample points, is created in Line 1 of the coding routine. In this case, since $|win|$ is greater than 1, the window is encoded in lines 7-12.

In Line 8, the auxiliary routine GetDisPointsMDP, presented in Figure 3.71, is invoked with the following inputs: *win*, $\epsilon = 1$, $displaced\_points = []$, $i_o = 0$, and $i_f = 11$. Figure 3.73 shows the information after lines 1-5 in said routine are executed. *displaced_points* is equal to $[(t_1, 1), (t_{12}, 1)]$, and *segment* is the line segment whose endpoints are $(t_1, 1)$ and $(t_{12}, 1)$. In this case, the valid segment condition is not satisfied, i.e. *valid_segment* is false, since there are three points in *win*, for which the vertical distance to *segment* is greater than ϵ. These points are $(t_7, 3)$, $(t_8, 3)$, and $(t_9, 4)$. Since *valid_segment* is false, $half = \lfloor 11/2 \rfloor = 5$ is obtained (Line 7), and two recursive calls to routine GetDisPointsMDP are made (lines 8-9).

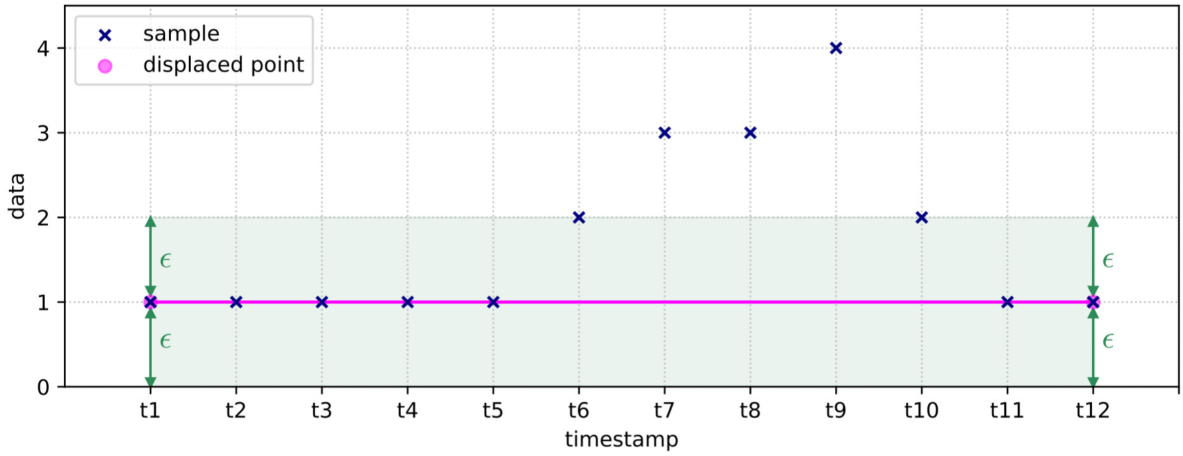

FIGURE 3.73: Example: Algorithm variant FR_M with $\epsilon = 1$ and $w = 256$. Step 1.

The first recursive call has inputs $displaced\_points = [(t_1, 1), (t_{12}, 1)]$, $i_o = 0$ and $i_f = 5$, and it adds a single point to *displaced_points*, making it equal to $[(t_1, 1), (t_6, 2), (t_{12}, 1)]$. In this case, *segment* is the line segment whose endpoints are $(t_1, 1)$ and $(t_6, 2)$, and the valid segment condition is satisfied, since there are no points in *win*, with indexes between 0 and 5, for which the vertical distance to *segment* is greater than ϵ. This is shown in Figure 3.74. Since the valid segment condition is satisfied in the execution of the first recursive call, no further calls are made to the recursive routine.

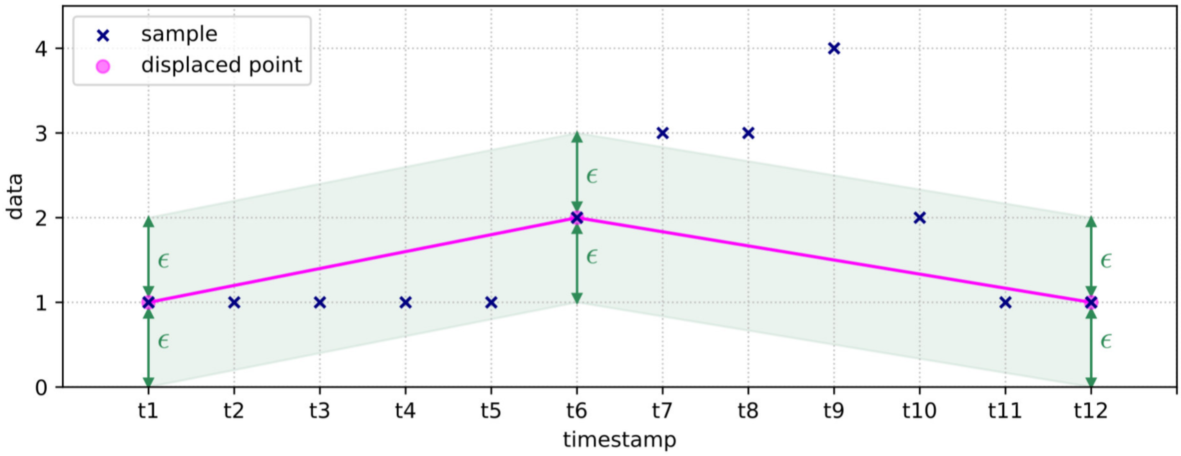

FIGURE 3.74: Example: Algorithm variant FR_M with $\epsilon = 1$ and $w = 256$. Step 2.

However, this is not the case in the execution of the second recursive call, which has inputs $displaced\_points = [(t_1, 1), (t_6, 2), (t_{12}, 1)]$, $i_o = 5$ and $i_f = 11$. Here, Figure 3.74 shows that the valid segment condition is violated by three points in the window (the same three points as before). Once again, *valid_segment* is false, $half = \lfloor(5 + 11)/2\rfloor = 8$ is obtained, and two recursive calls to the routine GetDisPointsMDP are made. In the first execution, point $(t_9, 4)$ is added to *displaced_points*. The valid segment condition is satisfied in both executions, so in both cases no further calls are made to the recursive routine.

After the original invocation of the routine GetDisPointsMDP is completed (Line 8 in the coding routine, shown in Figure 3.70), list *displaced_points* is equal to $[(t_1, 1), (t_6, 2), (t_9, 4), (t_{12}, 1)]$. This information is shown in Figure 3.75. Observe that, for every point in *win*, its vertical distance to the corresponding segment is at most ϵ.

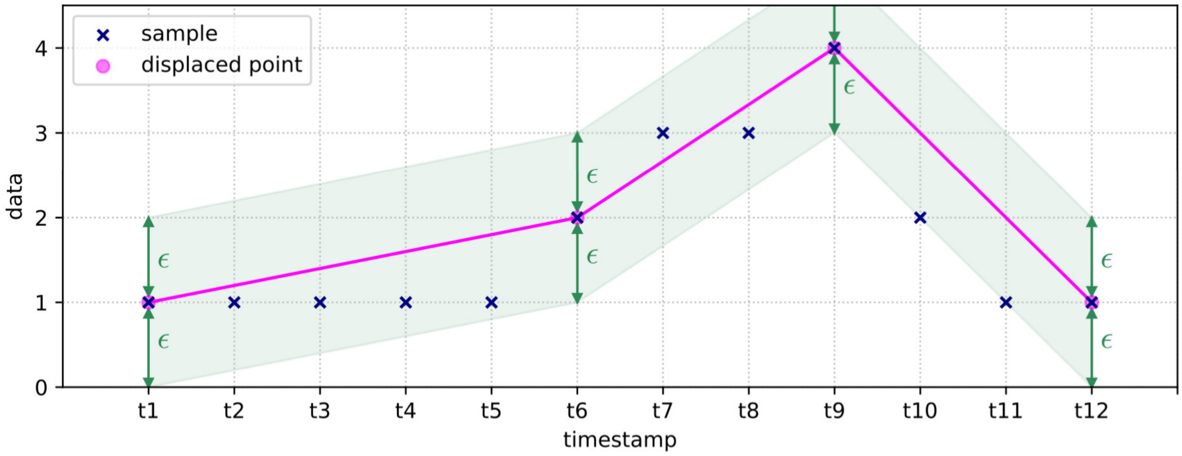

FIGURE 3.75: Example: Algorithm variant FR_M with $\epsilon = 1$ and $w = 256$. Step 3.

In this example, the sample points in *win* are modeled by three connected line segments. Their respective endpoints are the four displaced points obtained with the recursive MPD technique, i.e. $(t_1, 1)$, $(t_6, 2)$, $(t_9, 4)$, and $(t_{12}, 1)$, which are encoded in lines 9-12 of the coding routine. Encoding an endpoint requires $\lceil \log_2 w \rceil = \log_2 256 = 8$ bits for encoding its *win* index, and B_c bits for encoding its y-coordinate. Thus, a total of $4 * (\lceil \log_2 w \rceil + B_c) = 32 + 4 * B_c$ bits are required for encoding the 12 samples. The four encoded endpoints and the associated segments are shown in Figure 3.76. In this figure we also display the values of the decoded samples, which are the values that the decoding routine, shown in Figure 3.72, writes to the decoded CSV data file.

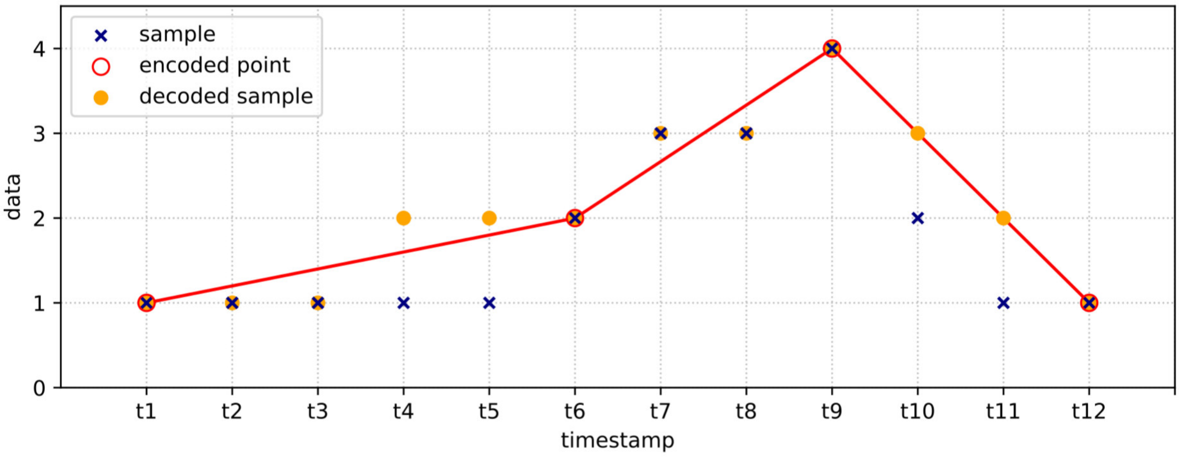

FIGURE 3.76: Example: Algorithm variant FR_M with $\epsilon = 1$ and $w = 256$. Step 4.

3.12 Algorithm GAMPS

Algorithm *Grouping and AMPlitude Scaling (GAMPS)* [40] is a correlation model algorithm that supports lossless and near-lossless compression. For GAMPS we define both variants, *M* and *NM*.

Algorithm GAMPS follows the general scheme presented in Figure 3.1, with a specific coding routine, for variant *M*, shown in Figure 3.77. Notice that the first input consists of all the columns in the CSV data file, in contrast to constant and linear model algorithms, presented in previous sections, which receive a single column. This is due to the fact that correlation model algorithms, such as GAMPS, also exploit the spatial correlation in the data, thus they jointly encode the data columns.

Before encoding the data columns, algorithm GAMPS groups them into disjoint subsets of spatially correlated columns. In each subset, a single *base column* is defined, while the rest of its columns are referred to as *ratio columns*. As we describe in the sequel, this grouping, the optimization of the grouping, and the choice of the base column for each subset, is done trough an auxiliary algorithm. The base column is encoded using algorithm APCA_F, which is quite similar to algorithm APCA, presented in Chapter 3.6, the only differences being that in the auxiliary routine EncodeWindow, shown in Figure 3.12, in the APCA_F case, floor is not applied when calculating *mid_range*, and *mid_range* is encoded as a float (i.e. using 32 bits). The ratio columns are first transformed, dividing each sample by the base column sample corresponding to the same timestamp (to avoid division by zero the samples of the base column are offset), and then encoded, which is also done using APCA_F. The original ratio signals may be rough (i.e. with abrupt changes), but, since base and ratio signals in the same subset are spatially correlated, the transformed ratio signals are expected to be smooth (i.e. slowly varying). As we recall from Section 3.6, algorithm APCA achieves better compression performances on smooth rather than rough signals, and this is also the case for algorithm APCA$_\text{F}$. Therefore, the key idea behind algorithm GAMPS is that, under certain circumstances, it may be more convenient to exploit the spatial correlation in the data, and encode, for each subset, a single rough base signal and a handful of smooth transformed signals, rather than independently encode each of the original rough signals.

```
input  : columns: columns of the CSV data file to be encoded
         out: binary file encoded with algorithm variant GAMPS_M
         ϵ: maximum error threshold
         w: maximum window size
1  Group columns into disjoint subsets of spatially correlated columns, where in each
     subset, a single base column is defined, and the rest of its columns are considered ratio
     columns. Also define error threshold parameters ϵ_b and ϵ_r, for base and ratio columns,
     respectively. This is done solving the facility location problem defined in [40].
2  Encode the number of subsets using ⌈log_2 |columns|⌉ bits
3  foreach subset of columns, cols_group, do
4      Let base_col be the base column in cols_group
5      Let ratio_cols be the ratio columns in cols_group
6      Encode the index of base_col, the number of ratio columns, and the index of each
         column in ratio_cols, using ⌈log_2 |columns|⌉ bits in every case
7      Encode base_col by calling the coding routine for variant M of algorithm APCA_F
         with parameters base_col, out, ϵ_b, w
8      foreach column in ratio_cols, ratio_col, do
9          Let trans_ratio_col be obtained by applying function ratio_col/base_col
10         Encode trans_ratio_col by calling the coding routine for variant M of algorithm
             APCA_F with parameters trans_ratio_col, out, ϵ_r, w
11     end
12 end
```

FIGURE 3.77: Coding routine for algorithm variant GAMPS$_M$.

The initial step in the coding routine, which gives the first part of its name to the algorithm, consists of grouping the data columns into disjoint subsets of spatially correlated columns (Line 1). A *facility location problem*, whose goal is to find a grouping that reduces the number of bits required for encoding the columns, is solved in [40, §4.2.1]. Its inputs are the set of columns, *columns*, and the maximum error threshold, ϵ. The output is the grouping, including the base column for each group, and the two maximum error threshold parameters, ϵ_b and ϵ_r, used for encoding each base column and its ratio columns, respectively. In the case that no such grouping of spatially correlated columns is found, the trivial output consists of groups formed by a single (base) column, and $\epsilon_b = \epsilon$. We point out that we narrow the universe of solutions by only allowing columns corresponding to signals of the same data type to be grouped together. We find out that this not only reduces the size of the facility location problem, which is computationally expensive to solve, but in our case it also lead to better compression results.

The next step in the coding routine consists of encoding the number of subsets (Line 2). Afterwards, the routine iterates through each subset of columns (lines 4-11). In each iteration, all the data columns in the subset are encoded. In Line 6, the index of the base column, and the number of ratio columns and their respective indexes are encoded. This information is used by the decoding routine to recreate each subset. In Line 7, the base column is encoded by invoking the coding routine for algorithm APCA$_\text{F}$ with an error threshold equal to ϵ_b. Finally, there is a loop in lines 8-11, in which each of the ratio columns is transformed, as explained above, and then encoded by invoking the APCA$_\text{F}$ coding routine with an error threshold equal to ϵ_r. We point out that this loop is only executed if the output of the facility location problem solved in the initial step of the coding routine is non-trivial.

The decoding routine is symmetric to the coding routine. To recreate each subset, the information encoded in lines 2 and 6 of the coding routine must be decoded. For decoding the data columns in each subset, first, the base column is decoded by invoking the decoding routine for algorithm APCA_F. Then, each of the transformed ratio columns is decoded, also by invoking the APCA_F decoding routine, and the transformation applied in Line 9 is reverted, by multiplying the decoded base column by the decoded transformed ratio column.

3.12.1 Example

Next we present an example of the encoding of three signals with 12 samples each, illustrated in Figure 3.78. Recall that the specific timestamp values are irrelevant for algorithm APCA, and this is also the case for algorithm APCA_F. Therefore, they are also irrelevant for algorithm GAMPS. For this example we let the error threshold parameter (ϵ) be equal to 0, and the maximum window size (w) equal to 256. We point out that this is the only example where the coding is lossless. In this case we select $\epsilon = 0$ so that the facility location problem considered in Line 1 has a straightforward solution, and we can focus on illustrating the most important parts of the algorithm.

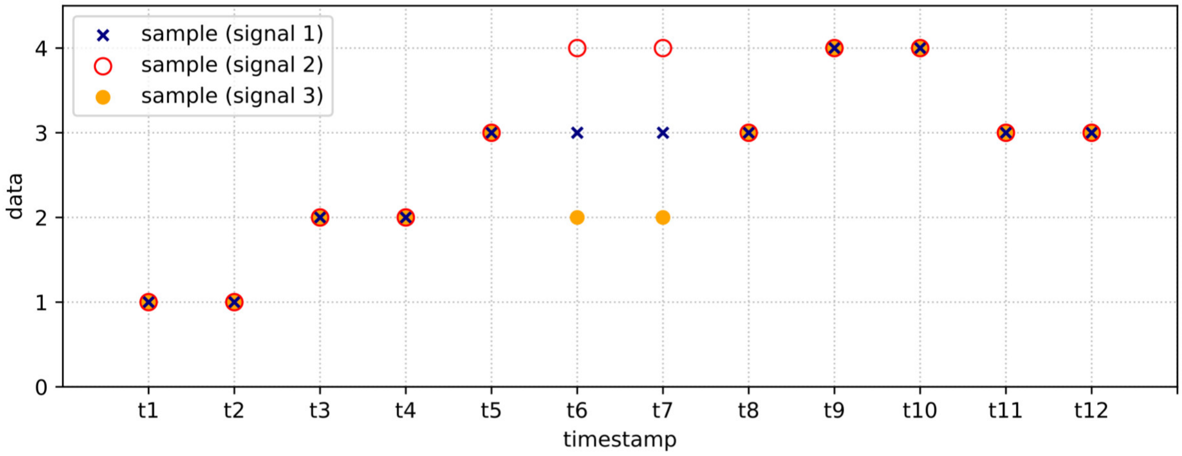

FIGURE 3.78: Example: Three signals consisting of 12 samples each.

There is a high degree of spatial correlation between the three signals, with the sample values matching for 10 of the 12 timestamps. Thus, in Line 1, a single subset including the three columns with the samples of each signal is obtained. The first column is defined as the base column, making the remaining two, ratio columns. Since ϵ is equal to 0, both ϵ_b and ϵ_r must also be 0. In Line 2, the number of subsets (i.e. 1) is encoded using $\lceil \log_2 |columns| \rceil = \lceil \log_2 3 \rceil = 2$ bits. Since there is a single subset, there is a single iteration. In Line 6, the information used by the decoding routine to recreate the subset is encoded, using $\lceil \log_2 |columns| \rceil = 2$ bits for encoding each of the following values: the index of the base column (i.e. 0), the number of ratio columns (i.e. 2), and the index of each ratio column (i.e. 1 and 2). Therefore, a total of 10 bits are required for encoding the grouping of the columns in this example.

Next, in Line 7, the base column is encoded by calling the coding routine for algorithm APCA_F. Since ϵ_b is equal to 0, the encoded samples match the original samples (i.e. [1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 3, 3]), and 5 windows are encoded. The loop in lines 8 through 11 repeats for each of the two ratio columns. In both cases, the ratio column is first transformed (Line 9), and then encoded by calling the APCA$_\mathrm{F}$ coding routine (Line 10). Since parameter ϵ_r is equal to 0, the encoded samples match the transformed samples, which are equal to $[1, 1, 1, 1, 1, 4/3, 4/3, 1, 1, 1, 1, 1]$ and $[1, 1, 1, 1, 1, 2/3, 2/3, 1, 1, 1, 1, 1]$, respectively, for each transformed ratio column, and 3 windows are encoded in each case. In Figure 3.79, the encoded samples of the three signals are shown.

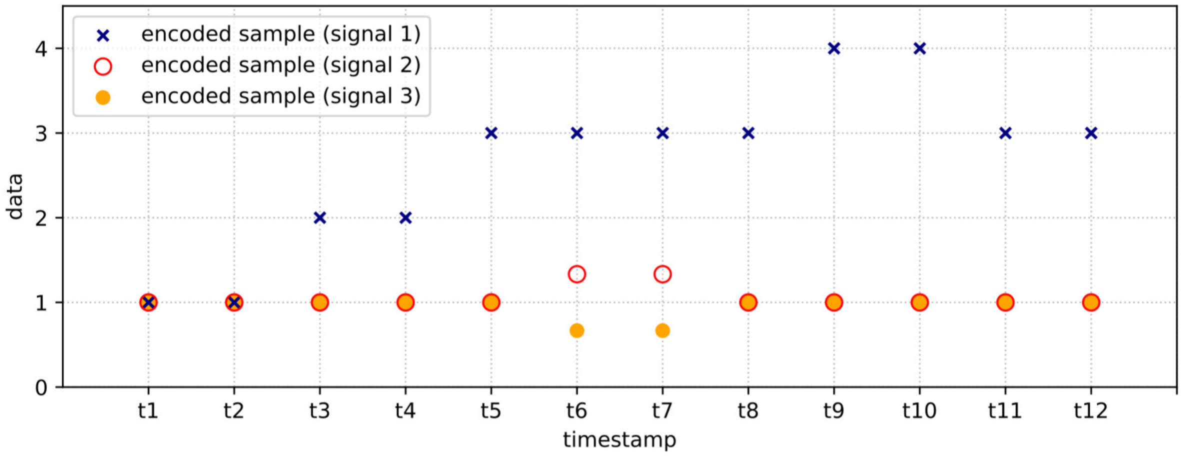

FIGURE 3.79: Example: Algorithm variant GAMPS$_M$ with $\epsilon = 0$ and $w = 256$. Step 1.

We recall that the coding routine for algorithm APCA always uses the same number of bits for encoding any window. Namely, $\lceil \log_2 w \rceil$ bits are used for encoding its size, and B_c bits are used for encoding its mid-range. Instead, the coding routine for algorithm APCA$_\mathrm{F}$ encodes the mid-range as a float (i.e. using 32 bits), so every window is encoded using exactly $\lceil \log_2 w \rceil + 32$ bits. If the three original signals were independently encoded with APCA$_\mathrm{F}$, encoding the first one would involve encoding 5 windows, while encoding the second and third signals would involve encoding 7 windows in each case. However, in this example, even though encoding the base signal also involves encoding 5 windows, encoding the transformed ratio signals involves encoding only 3 windows in each case. Therefore, grouping and encoding the transformed signals allows the algorithm to reduce the total number of encoded windows by 8 (from 19 to 11), which reduces the number of encoded bits by exactly $8 * (\lceil \log_2 w \rceil + 32) - 10$ bits (the 10 bits that are subtracted are used to encode the grouping of the columns). This is expected because, since there is a high degree of spatial correlation between the signals, the transformed ratio signals are smooth, compared to the original ratio signals, which are rough. Overall, $11 * (\lceil \log_2 w \rceil + 32) + 10 = 450$ bits are required for encoding the 12 samples in this example.

3.12.2 Non-Masking (*NM*) variant

The coding and decoding routines for variant *NM* of algorithm GAMPS are quite similar to their variant *M* counterparts, the difference being that the former routines are able to handle both sample values and gaps. The coding routine for variant *M* of algorithm GAMPS invokes the coding routine for variant *M* of algorithm APCA (lines 7 and 10 in Figure 3.77), which can only handle sample values, since the position of the gaps is already encoded (recall Line 6 in Figure 3.1). However, the coding routine for variant *NM* of algorithm GAMPS must instead invoke the coding routine for variant *NM* of algorithm APCA, presented in Subsection 3.6.2, which is able to handle both sample values and gaps.

3.13 Implementation Details

All the algorithm variants presented in this chapter are implemented in C++. The implementation of the variants for algorithms PCA, APCA, CA and FR is completely ours, while the variants for algorithms PWLH, SF and GAMPS reuse part of the source code from the framework cited in [27][2]. We implemented algorithm PWLHInt by introducing the changes described in Subsection 3.8.2 to algorithm PWLH.

In algorithms PWLH and SF, presented in sections 3.8 and 3.10, respectively, we reuse the framework source code for carrying out operations in the two-dimensional Euclidean space. In algorithm PWLH these operations include computing the convex hull of the data points by applying Graham's Scan algorithm [53] (Line 6 in Figure 3.19), and calculating the segment that minimizes the MSE for all the points in a window (Line 2 in Figure 3.22). In algorithm SF the reused operations include finding connected line segments (Line 21 in Figure 3.50), and calculating the segment that minimizes the MSE for all the points in a window (Line 1 in Figure 3.52). In algorithm GAMPS, presented in Section 3.12, we reuse the framework source code for grouping the data columns into disjoint subsets of spatially correlated columns, which involves solving a facility location problem (Line 1 in Figure 3.77).

Recall that in Section 3.3 we introduce the arithmetic coder (AC), which is employed for encoding the gaps in the masking variants. In our code we use the CACM87 implementation [54, 55] of the AC, which is written in C.

[2] The framework is available for download in the following website: http://lsirwww.epfl.ch/benchmark/

Chapter 4

Experimental Results

In this chapter we present our experimental results. The main goal of our experiments is to analyze the compression performance of each of the algorithm variants presented in Chapter 3, by encoding the various real-world datasets introduced in Chapter 2, and comparing the results.

In **Section 4.1** we describe our experimental setting, and define the evaluated combinations of algorithms, their variants and parameter values, and the figures of merit used for comparison.

In **Section 4.2** we compare the compression performance of the masking and non-masking variants implemented for each coding algorithm. The results show that, on datasets with few or no gaps, the performance of both variants is roughly the same, while on datasets with many gaps the masking variant always performs better, in some cases with a significant difference. These results suggest that the masking variant is more robust and performs better in general.

In **Section 4.3** we analyze the extent to which the window size parameter impacts the compression performance of the coding algorithm variants. We compress each dataset file, and compare the results obtained when using the optimal window size (i.e. the one that achieves the best compression) for the specific file, with the results obtained when using the optimal window size for the whole dataset, when all the files in the dataset are compressed using the same window size. The results indicate that the effect of using the optimal window size for the whole dataset, instead of the optimal window size for each file, is rather small.

Finally, in **Section 4.4** we compare the compression performance of our adapted algorithm variants, with each other, and with the general-purpose lossless compression algorithm gzip. The experimental results indicate that none of the algorithm variants obtains the best performance in every scenario. For larger error thresholds, variant APCA$_M$ achieves the best compression rates in most cases, while for lower thresholds, the same is true for algorithm gzip and variant PCA$_M$. We also study the lossless compression of the timestamp column: the results show that variant APCA$_{NM}$ achieves the best performance for this column in general.

4.1 Experimental Setting

We denote by A the set of all the coding algorithms presented in Chapter 3. For an algorithm $a \in A$, we denote by a_v its variant v, where v can be M (masking) or NM (non-masking). Recall that there exist some $a \in A$ for which either a_M or a_{NM} is invalid (see Table 3.1). We denote by V the set of variants consisting of every *valid* variant a_v for every algorithm $a \in A$. Also, we denote by A_M the subset of algorithms from A consisting of every algorithm for which both variants, a_M and a_{NM}, are valid.

We evaluate the compression performance of every algorithm $a \in A$ on the datasets described in Chapter 2. Considering that, in datasets with heterogeneous data, selecting different compression algorithms for different columns may lead to a better overall compression, we report our experimental results by data type. For each algorithm we test every valid variant a_v. We also test several combinations of algorithm parameters. Specifically, for the algorithms that admit a window size parameter w (every algorithm except Base), we test all the values of w in the set $W = \{4, 8, 16, 32, 64, 128, 256\}$. For the encoders that admit a near-lossless compression mode with an error threshold parameter ϵ (every encoder except Base), the value of ϵ is calculated as a percentage fraction, denoted e, of the standard deviation of the data being encoded. For example, for certain data with a standard deviation of 20, if $e = 10$, then $\epsilon = 2$. We test all the values of the parameter e in the set $E = \{1, 3, 5, 10, 15, 20, 30\}$.

Definition 4.1.1. We refer to a specific combination of a coding algorithm variant and its parameter values as a *coding algorithm instance (CAI)*. We define *CI* as the set of all the CAIs obtained by combining each of the variants $a_v \in V$ with the parameter values (from W and E) that are suitable for algorithm a. We denote by $c_{<a_v,w,e>}$ the CAI obtained by setting a window size parameter equal to w and an error parameter equal to e on algorithm variant a_v.

We assess the compression performance of a CAI through the compression ratio, which we define next. For this definition, we regard Base as a trivial CAI that serves as a base ground for compression performance comparison (recall the definition of algorithm Base from Section 3.4).

Definition 4.1.2. Let f be a file and z a data type of a certain dataset. We define f_z as the subset of data of type z from file f. For example, for the dataset Hail, presented in Section 2.7, the data type z may be Latitude, Longitude, or Size.

Definition 4.1.3. Let f be a file and z a data type of a certain dataset. Let $c \in CI$ be a CAI. We define $|c(z, f)|$ as the size in bits of the resulting bit stream obtained by coding f_z with c.

Definition 4.1.4. The *compression ratio (CR)* of a CAI $c \in CI$ for the data type z of a certain file f is the fraction of $|c(z, f)|$ with respect to $|\text{Base}(z, f)|$, i.e.,

$$\text{CR}(c, z, f) = \frac{|c(z, f)|}{|\text{Base}(z, f)|}. \tag{4.1}$$

Notice that smaller values of CR correspond to better performance. Our main goals are to analyze which CAIs yield the smallest values in (4.1) for the different data types, and to study how the CR depends on the different algorithms, their variants and the parameter values.

To compare the compression performance between two CAIs we calculate the relative difference, which we define next.

Definition 4.1.5. The *relative difference (RD)* between a pair of CAIs $c_1, c_2 \in CI$ for the data type z of a certain file f is given by

$$\mathrm{RD}(c_1, c_2, z, f) = 100 \times \frac{|c_2(z,f)| - |c_1(z,f)|}{|c_2(z,f)|}. \tag{4.2}$$

Notice that c_1 has a better performance than c_2 if (4.2) is positive.

In some of our experiments we consider the performance of algorithms on complete datasets, rather than individual files. With this in mind, we extend the definitions 4.1.3–4.1.5 to datasets, as follows.

Definition 4.1.6. Let z be a data type of a certain dataset d. We define $F(d, z)$ as the set of files f from dataset d for which f_z is not empty.

Definition 4.1.7. Let z be a data type of a certain dataset d. Let $c \in CI$ be a CAI. We define $|c(z, d)|$ as

$$|c(z,d)| = \sum_{f \in F(d,z)} |c(z,f)|. \tag{4.3}$$

Definition 4.1.8. The *compression ratio (CR)* of a CAI $c \in CI$ for the data type z of a certain dataset d is given by

$$\mathrm{CR}(c, z, d) = \frac{|c(z,d)|}{|\mathrm{Base}(z,d)|}. \tag{4.4}$$

Definition 4.1.9. The *relative difference (RD)* between a pair of CAIs $c_1, c_2 \in CI$ for the data type z of a certain dataset d is given by

$$\mathrm{RD}(c_1, c_2, z, d) = 100 \times \frac{|c_2(z,d)| - |c_1(z,d)|}{|c_2(z,d)|}. \tag{4.5}$$

4.2 Comparison of Masking and Non-Masking Variants

In this section, we compare the compression performance of the masking and non-masking variants of every coding algorithm in A_M (recall this definition from the first paragraph in Section 4.1). Specifically, we compare:

- PCA_M against PCA_{NM}
- APCA_M against APCA_{NM}
- CA_M against CA_{NM}
- PWLH_M against PWLH_{NM}
- PWLHInt_M against PWLHInt_{NM}
- GAMPS_M against GAMPS_{NM}

For each algorithm $a \in A_M$, and each error parameter $e \in E$, we compare the performance of a_M and a_{NM}. For the purpose of this comparison, we choose the most favorable window size for each variant a_v, in the sense of the following definition.

Definition 4.2.1. The *optimal window size (OWS)* of a coding algorithm variant $a_v \in V$, and an error parameter $e \in E$, for the data type z of a certain dataset d, is given by

$$\text{OWS}(a_v, e, z, d) = \underset{w \,\in\, W}{\arg\min} \left\{ \text{CR}(c_{<a_v, w, e>}, z, d) \right\}, \tag{4.6}$$

where we break ties in favor of smaller window sizes.

For each data type z of each dataset d, and each coding algorithm $a \in A_M$ and error parameter $e \in E$, we calculate the RD between $c_{<a_M, w^*_M, e>}$ and $c_{<a_{NM}, w^*_{NM}, e>}$, as defined in (4.5), where $w^*_M = \text{OWS}(a_M, e, z, d)$ and $w^*_{NM} = \text{OWS}(a_{NM}, e, z, d)$.

As an example, in figures 4.1 and 4.2 we show the CR and the RD, as a function of the error parameter, obtained for two data types of two different datasets. Figure 4.1 shows the results for the data type $z =$ "SST" of the dataset $d =$ SST, presented in Section 2.3, and Figure 4.2 shows the results for the data type $z =$ "Longitude" of the dataset $d =$ Tornado, presented in Section 2.8. In Figure 4.1 we observe a large RD favoring the masking variant for all tested algorithms. On the other hand, in Figure 4.2 we observe that the non-masking variant outperforms the masking variant for all algorithms. We notice, however, that the RD is very small in the latter case.

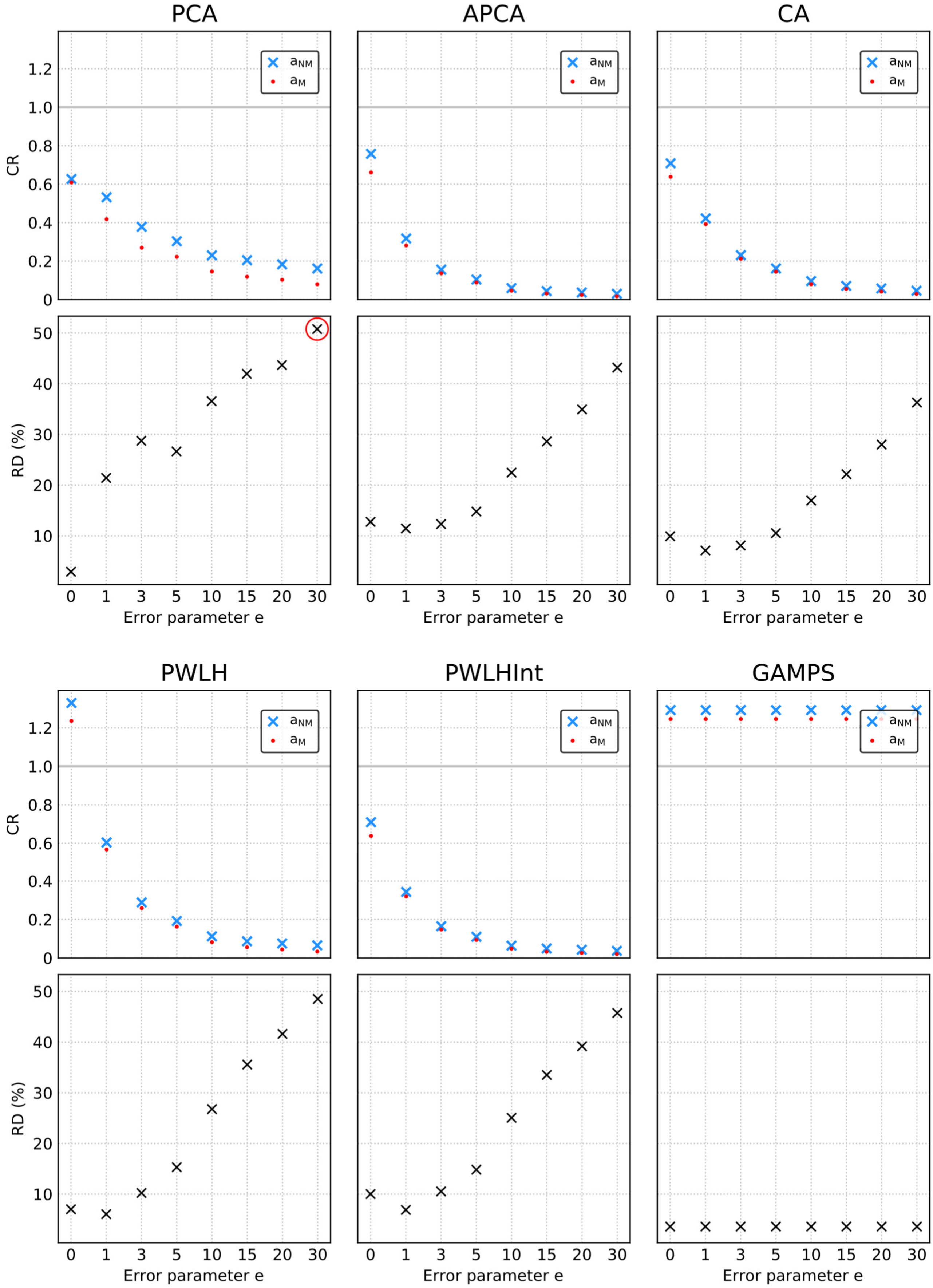

FIGURE 4.1: CR and RD plots for variants a_M and a_{NM}, for each algorithm $a \in A_M$, for the data type "SST" of the dataset SST. In the RD plot for algorithm PCA we highlight with a red circle the marker for the maximum value (50.78%) obtained for all the tested CAIs.

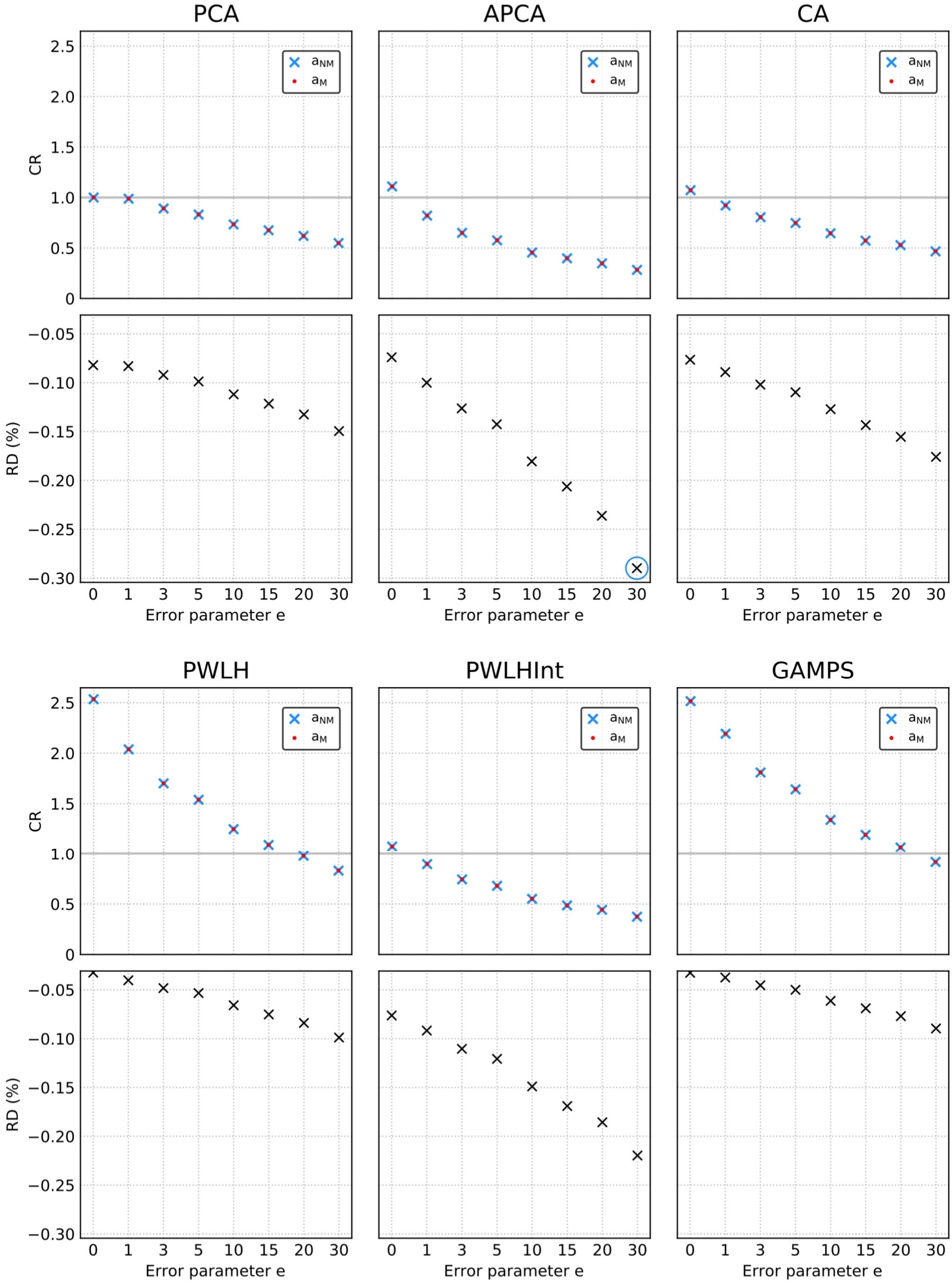

FIGURE 4.2: CR and RD plots for variants a_M and a_{NM}, for each algorithm $a \in A_M$, for the data type "Longitude" of the dataset Tornado. In the RD plot for algorithm APCA we highlight with a blue circle the marker for the minimum value (-0.29%) obtained for all the tested CAIs.

We analyze the experimental results to compare the performance of the masking and non-masking variants of each algorithm. For each data type, we iterate through each algorithm $a \in A_M$, and each error parameter $e \in E$, and we calculate the RD between the CAIs $c_{<a_M, w^*_M, e>}$ and $c_{<a_{NM}, w^*_{NM}, e>}$, obtained by setting the OWS for the masking variant a_M and the non-masking variant a_{NM}, respectively. Since we consider 8 different error parameter values and there are 6 algorithms in A_M, for each data type we compare a total of 48 pairs of CAIs. Table 4.1 summarizes the results of these comparisons, aggregated by dataset. The number of pairs of CAIs evaluated for each dataset depends on the number of different data types it contains.

Dataset	**Dataset Characteristic**	**Cases where a_M outperforms a_{NM} (%)**	**RD (%) Range**
IRKIS	Many gaps	48/48 (100%)	[3.04; 37.06]
SST	Many gaps	48/48 (100%)	[2.91; 50.78]
ADCP	Many gaps	48/48 (100%)	[2.44; 17.37]
ElNino	Many gaps	336/336 (100%)	[3.38; 50.5]
Solar	Few gaps	73/144 (50.7%)	[-0.26; 1.76]
Hail	No gaps	0/144 (0%)	[-0.04; 0)
Tornado	No gaps	0/96 (0%)	[-0.29; -0.01]
Wind	No gaps	0/144 (0%)	[-0.12; 0)

Table 4.1: Range of values for the RD between the masking and non-masking variants of each algorithm (last column); we highlight the maximum (red) and minimum (blue) values taken by the RD. The results are aggregated by dataset. The second column indicates the characteristic of each dataset, in terms of the number of gaps. The third column shows the number of cases in which the masking variant outperforms the non-masking variant of a coding algorithm, and its percentage among the total pairs of CAIs compared for a dataset.

Consider, for example, the results for the dataset Wind, in the last row. The second column shows that there are no gaps in any of the data types of the dataset (recall the dataset information from Table 2.13). Since the dataset has three data types, we compare a total of $3 \times 48 = 144$ pairs of CAIs. The third column reveals that in none of these comparisons the masking variant a_M outperforms the non-masking variant a_{NM}, i.e. the RD is always negative. The last column shows the range for the values attained by the RD for those tested CAIs.

Observing the last column of Table 4.1, we notice that, in every case in which the non-masking variant performs best, the RD is close to zero. The minimum value it takes is -0.29%, which is obtained for the data type "Longitude" of the dataset Tornado, with algorithm APCA, and error parameter $e = 30$. In Figure 4.2 we highlight the marker associated to this minimum with a blue circle. On the other hand, we observe that, for the datasets in which the masking variant performs best, the RD reaches high absolute values. The maximum (50.78%) is obtained for the data type "VWC" of the dataset SST, with algorithm PCA, and error parameter $e = 30$, which is highlighted in Figure 4.1 with a red circle.

The experimental results presented in this section suggest that if we were interested in compressing a dataset with many gaps, we would benefit from using the masking variant of an algorithm, a_M. However, even if the dataset didn't have any gaps, the performance would not be significantly worse than that obtained by using the non-masking variant of the algorithm, a_{NM}. Therefore, since masking variants are, in general, more robust in this sense, in the sequel we focus on the set of variants V^* that we define next.

Definition 4.2.2. We denote by V^* the set of all the masking algorithm variants a_M for $a \in A$.

Notice that V^* includes a single variant for each algorithm. Therefore, in what follows we sometimes refer to the elements of V^* simply as algorithms.

4.3 Window Size Parameter

In this section, we analyze the extent to which the window size parameter impacts on the performance of the coding algorithm variants. For these experiments we consider the set of algorithms V^*, defined in 4.2.2, and we only consider the four datasets that consist of multiple files, i.e. IRKIS, SST, ADCP and Solar (recall this information from Table 2.13). For each file, we compare the compression performance when using the OWS for the dataset, as defined in (4.6), and the LOWS for the file, defined next.

Definition 4.3.1. The *local optimal window size (LOWS)* of a coding algorithm variant $a_v \in V^*$, and an error parameter $e \in E$, for the data type z of a certain file f is given by

$$\mathrm{LOWS}(a_v, e, z, f) = \underset{w \,\in\, W}{\arg\min} \Big\{ \mathrm{CR}(c_{<a_v,w,e>}, z, f) \Big\}, \tag{4.7}$$

where we break ties in favor of smaller window sizes.

For each data type z of each dataset d, and each file $f \in F(d,z)$, coding algorithm variant $a_v \in V^*$, and error parameter $e \in E$, we calculate the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as defined in (4.2), where $w^*_{global} = \mathrm{OWS}(a_v, e, z, d)$ and $w^*_{local} = \mathrm{LOWS}(a_v, e, z, f)$. In what follows, we sometimes denote the OWS and the LOWS as w^*_{global} and w^*_{local}, respectively.

As an example, in figures 4.3 and 4.4 we show w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the error parameter e, obtained for the data type z = “VWC”, for two different files of the dataset d = IRKIS, presented in Section 2.2. Figure 4.3 shows the results for the file f = “irkis-1202.csv”, and Figure 4.4 shows the results for f = “irkis-1203.csv”. Observe that the values of w^*_{global} are the same for both figures, which is expected, since both are obtained from the same data type of the same dataset.

In Figure 4.3 we notice, for instance, that for variant APCA$_M$ the OWS and LOWS values match for every error parameter e, except 3 and 10. The OWS is larger than the LOWS when $e = 3$, but it is smaller when $e = 10$. In these two cases, the RD values are 1.52% and 1.76%, respectively. In Figure 4.4 we observe that in every case the OWS is larger than or equal to the LOWS. We highlight the marker for the maximum RD value (10.6%) obtained for all the tested CAIs, and we further comment on this point in the remaining of the section. Notice that in both figures the RD is non-negative in every plot, which makes sense, since the CR obtained with the OWS can never be lower than the CR obtained with the LOWS.

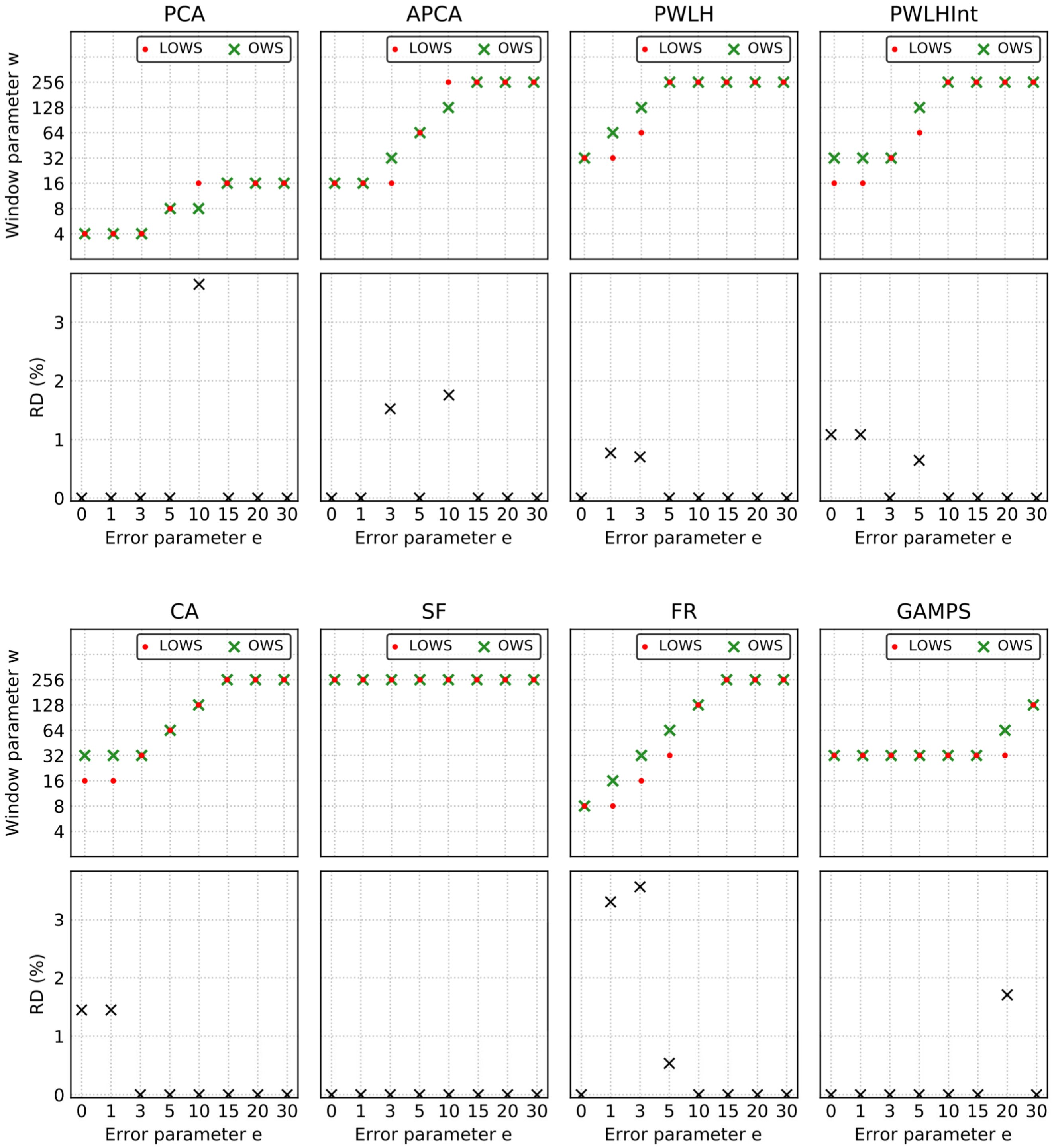

FIGURE 4.3: Plots of w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the error parameter e, obtained for the data type "VWC" of the file "irkis-1202.csv" of the dataset IRKIS.

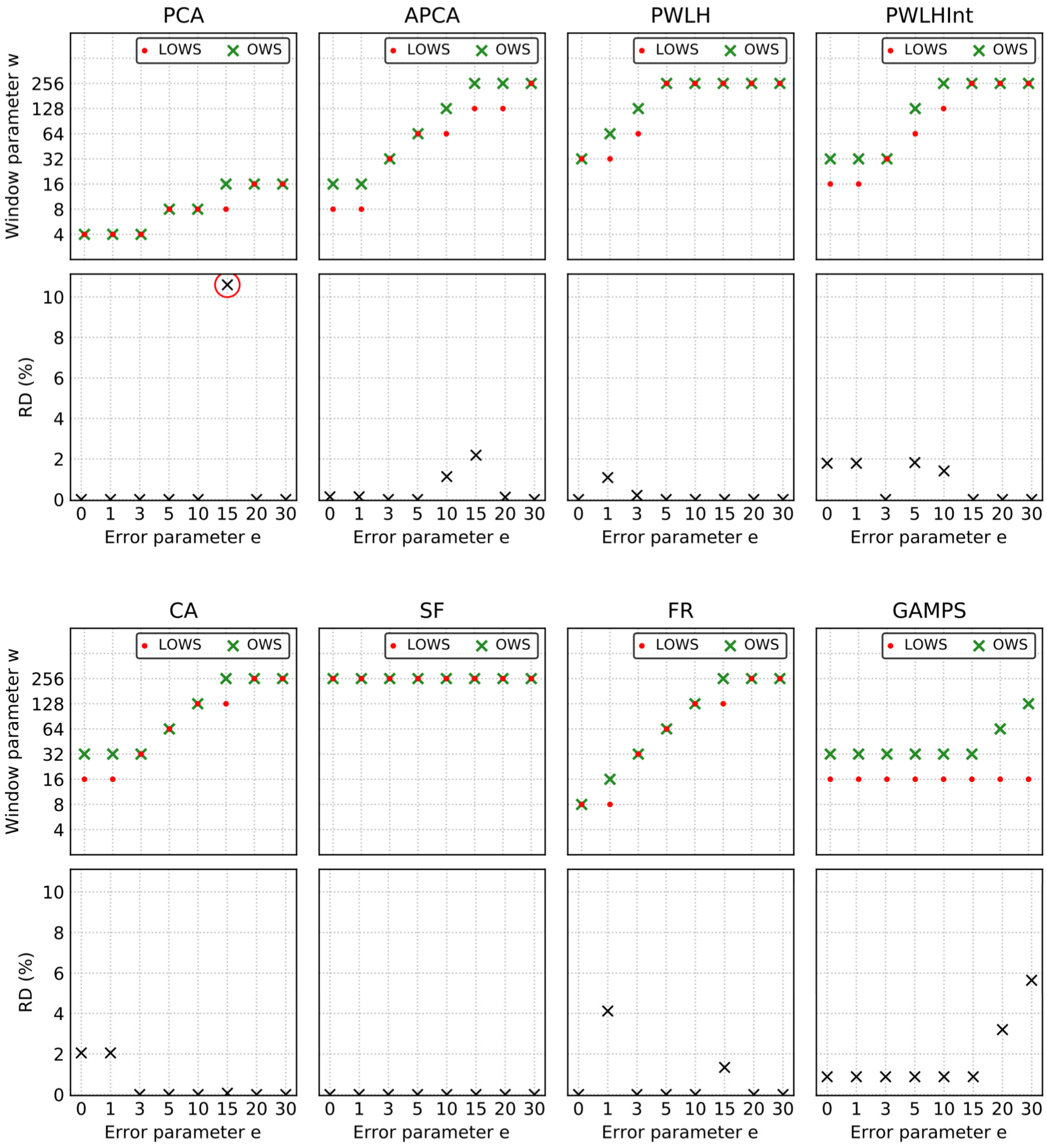

FIGURE 4.4: Plots of w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the error parameter e, obtained for the data type "VWC" of the file "irkis-1203.csv" of the dataset IRKIS. In the RD plot for variant PCA$_M$ we highlight with a red circle the marker for the maximum value (10.6%) obtained for all the tested CAIs.

We analyze the experimental results to evaluate the impact of using the OWS instead of the LOWS on the compression performance of the tested coding algorithms. For each algorithm, we iterate through each error parameter, and each data type of each file, and we calculate the RD between the CAI with the OWS and the CAI with the LOWS. Since we consider 8 different error parameter values and there are 13 files with a single data type and 4 files with 3 different data types each, for each algorithm we compare a total of $8 \times (13 + 4 \times 3) = 200$ pairs of CAIs. Table 4.2 summarizes the results of these comparisons, aggregated by algorithm and the range to which the RD belongs.

	RD (%) Range				
Algorithm	**0**	**(0,1]**	**(1,2]**	**(2,5]**	**(5,11]**
PCA	186 (93%)	3 (1.5%)	4 (2%)	2 (1%)	5 (2.5%)
APCA	174 (87%)	13 (6.5%)	7 (3.5%)	6 (3%)	0
PWLH	184 (92%)	13 (6.5%)	3 (1.5%)	0	0
PWLHInt	180 (90%)	8 (4%)	9 (4.5%)	3 (1.5%)	0
CA	181 (90.5%)	8 (4%)	6 (3%)	5 (2.5%)	0
SF	199 (99.5%)	1 (0.5%)	0	0	0
FR	171 (85.5%)	14 (7%)	8 (4%)	7 (3.5%)	0
GAMPS	167 (83.5%)	15 (7.5%)	11 (5.5%)	3 (1.5%)	4 (2%)
Total	1,442 (90.1%)	75 (4.7%)	48 (3%)	26 (1.6%)	9 (0.6%)

Table 4.2: RD between the OWS and LOWS variants of each CAI.
The results are aggregated by algorithm and the range to which the RD belongs.

For example, consider the results for algorithm CA, in the fifth row. The first column indicates that the RD is equal to 0 for exactly 181 (90.5%) of the 200 evaluated pairs of CAIs for that algorithm. The second column reveals that for 8 pairs of CAIs (4%), the RD takes values greater than 0 and smaller than or equal to 1%. The remaining three columns cover other ranges of RD values. Notice that for every row (except the last one), the values add up to a total of 200, since we compare exactly 200 pairs of CAIs for each algorithm.

The last row of Table 4.2 is obtained by adding the values of the previous rows, which combines the results for all algorithms. We notice that in 90.1% of the total number of evaluated pairs of CAIs, the RD is equal to 0. In these cases, in fact, the OWS and the LOWS coincide. Moreover, in 97.8% of the cases, the RD is less than or equal to 2%. This means that, for the vast majority of CAI pairs, either the OWS and the LOWS match or they yield roughly the same compression performance. This result suggests that we could fix the window size parameter in advance, for example by optimizing over a training set, without significantly compromising the performance of the coding algorithm. This is relevant, since calculating the LOWS for a file is, in general, computationally expensive.

We notice that there are only 9 cases (0.6%) in which the RD falls in the range (5, 11]. This only occurs for algorithms PCA and GAMPS. The maximum value taken by RD (10.6%) is obtained for the data type "VWC" of the file "irkis-1203.csv" of the dataset IRKIS, with algorithm PCA, and error parameter $e = 15$. In Figure 4.4 we highlight this maximum value with a red circle. In this case, the OWS is 16 and the LOWS is 8. According to these results, the performance of algorithms PCA and GAMPS seems to be more sensible to the window size parameter than the rest of the algorithms. Except for these few cases, we observe that, in general, the impact of using the OWS instead of the LOWS on the compression performance of coding algorithms is rather small. Therefore, in the following section, in which we compare the algorithms performance, we always use the OWS.

4.4 Algorithm Compression Performance

In this section, we compare the compression performance of the coding algorithm variants presented in Chapter 3, by encoding the various datasets introduced in Chapter 2. We begin by comparing the variants with each other, and in Subsection 4.4.1 we compare them with gzip [10], a popular general-purpose lossless compression algorithm. In Subsection 4.4.2 we study the lossless compression of the timestamp column.

We analyze the performance of the algorithm variants on complete datasets (not individual files), so we always apply definitions 4.1.6–4.1.9. Following the results obtained in sections 4.2 and 4.3, we only consider the masking variants of the evaluated algorithms (i.e. set V^*), and we always set the window size parameter to the OWS (recall Definition 4.2.1).

For each data type z of each dataset d, and each coding algorithm variant $a_v \in V^*$ and error parameter $e \in E$, we calculate the CR of $c_{<a_v, w^*_{global}, e>}$, as defined in (4.4), where $w^*_{global} = \text{OWS}(a_v, e, z, d)$. The following definition is useful for analyzing which CAI obtains the best compression result for a specific data type.

Definition 4.4.1. Let z be a data type of a certain dataset d, and let $e \in E$ be an error parameter. We denote by $c^b(z, d, e)$ the *best CAI* for z, d, e, and define it as the CAI that minimizes the CR among all the CAIs in CI, i.e.,

$$c^b(z, d, e) = \underset{c\ \in CI}{\arg\min} \left\{ \text{CR}(c_{<a_v, w^*_{global}, e>}, z, d) \right\}. \tag{4.8}$$

When $c^b(z, d, e) = c_{<a^b_v, w^{b*}_{global}, e>}$, we refer to a^b_v and w^{b*}_{global} as the *best coding variant* and the *best window size* for z, d, e, respectively.

Our experiments include a total of 21 data types, in 8 different datasets (recall this information from Table 2.13). As an example, in Figure 4.5 we show the $\text{CR}(c_{<a_v, w^*_{global}, e>}, z, d)$ and the window size parameter w^*_{global}, as a function of the error parameter e, obtained for each algorithm $a_v \in V^*$, for the data type $z =$ “Sea Temp.” of the dataset $d =$ ElNino, presented in Section 2.5. For each error parameter $e \in E$, we use blue circles to highlight the markers for the minimum CR value and the best window size (in the respective plots corresponding to the best coding variant). For instance, for $e = 0$, the best CAI achieves a CR equal to 0.33 using algorithm PCA with a window size of 256. Thus, in this case, algorithm PCA is the best coding variant, and 256 is the best window size. For the remaining 7 values of the error parameter, the blue circles indicate that in every case the best algorithm is APCA, and the best window size ranges from 4 up to 32.

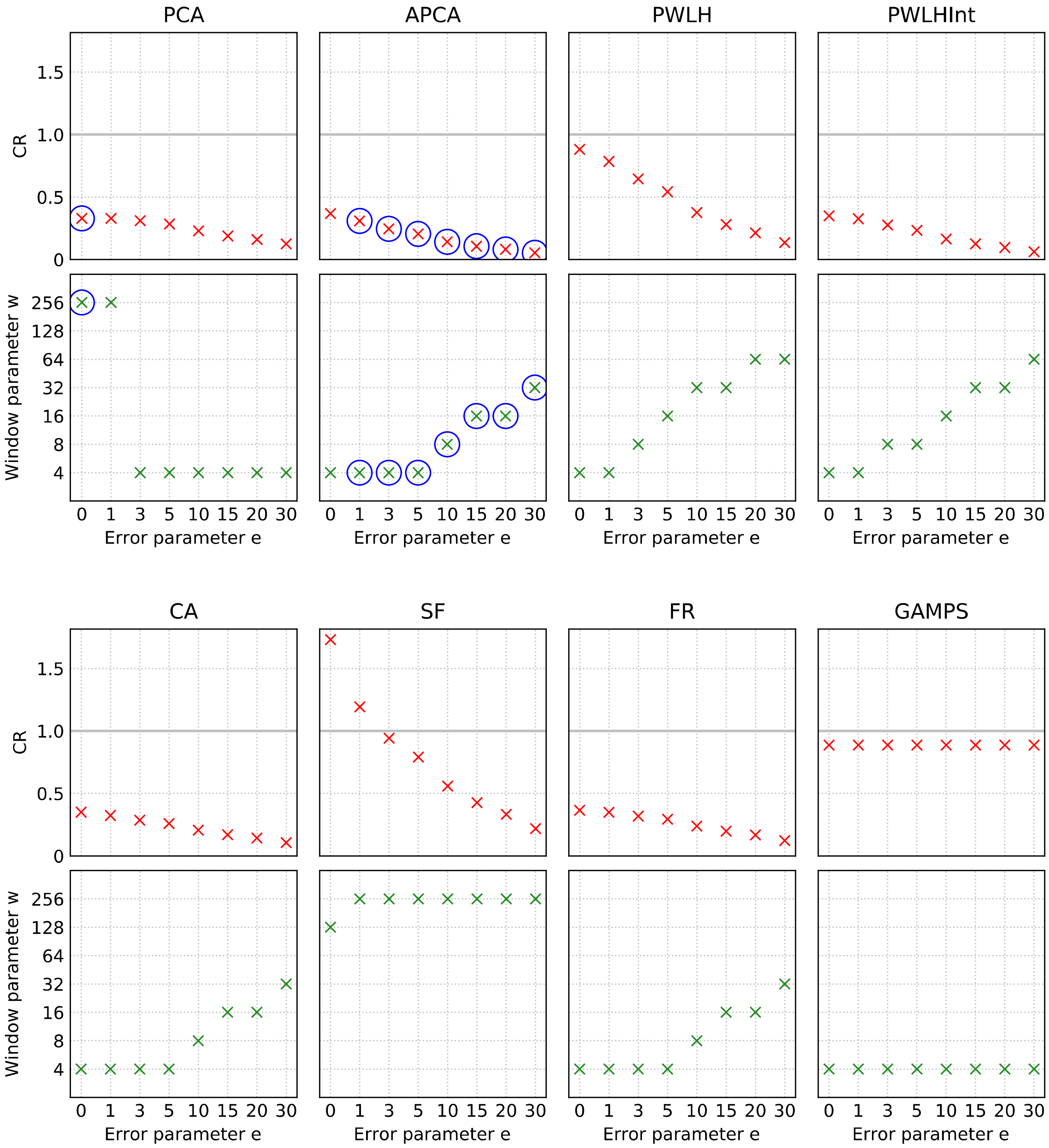

FIGURE 4.5: CR and window size parameter plots for every evaluated algorithm, for the data type “Sea Temp.” of the dataset ElNino. For each error parameter $e \in E$, we use blue circles to highlight the markers for the minimum CR value and the best window size (in the respective plots corresponding to the best coding variant).

Table 4.3 summarizes the compression performance results obtained by the evaluated coding algorithms, for each data type of each dataset. Each row contains information relative to certain data type. For example, the 13th row shows summarized results for the data type "Sea Temp." of the dataset ElNino, which are presented in more detail in Figure 4.5. For each error parameter value, the first column shows the CR obtained by the best CAI, the second column shows the base-2 logarithm of its window size parameter, and the cell color identifies the best algorithm.

PCA	APCA	CA	FR

		e = 0		e = 1		e = 3		e = 5		e = 10		e = 15		e = 20		e = 30	
Dataset	Data Type	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w
IRKIS	VWC	0.20	4	0.18	4	0.12	5	0.07	6	0.03	7	0.02	8	0.02	8	0.01	8
SST	SST	0.61	8	0.28	3	0.14	5	0.09	6	0.05	7	0.03	8	0.02	8	0.02	8
ADCP	Velocity	0.68	8	0.68	8	0.67	2	0.61	2	0.48	2	0.41	2	0.35	3	0.26	3
Solar	GHI	0.78	2	0.75	4	0.70	4	0.67	4	0.59	4	0.52	4	0.47	4	0.38	4
	DNI	0.76	2	0.72	4	0.66	4	0.61	4	0.54	4	0.49	4	0.43	4	0.36	4
	DHI	0.78	2	0.77	2	0.72	4	0.68	4	0.60	4	0.54	4	0.48	4	0.39	4
ElNino	Lat	0.16	4	0.16	4	0.16	4	0.15	4	0.12	4	0.10	5	0.09	5	0.06	6
	Long	0.17	3	0.17	4	0.13	4	0.12	5	0.09	6	0.07	6	0.05	7	0.02	8
	Zon. Wind	0.31	8	0.31	8	0.31	8	0.31	8	0.27	2	0.24	2	0.21	2	0.16	3
	Mer. Wind	0.31	8	0.31	8	0.31	8	0.31	8	0.29	2	0.26	2	0.23	2	0.19	2
	Humidity	0.23	8	0.23	8	0.23	8	0.23	8	0.21	2	0.18	2	0.16	2	0.13	2
	Air Temp.	0.33	8	0.33	8	0.30	2	0.27	2	0.22	2	0.19	3	0.17	3	0.13	4
	Sea Temp.	0.33	8	0.31	2	0.25	2	0.21	2	0.14	3	0.11	4	0.08	4	0.05	5
Hail	Lat	1.00	8	1.00	8	0.90	2	0.83	2	0.71	2	0.65	3	0.57	3	0.47	3
	Long	1.00	8	1.00	8	0.86	2	0.78	2	0.65	2	0.55	3	0.49	3	0.39	4
	Size	0.81	2	0.81	2	0.81	2	0.81	2	0.81	2	0.81	2	0.81	2	0.64	3
Tornado	Lat	1.00	8	0.85	2	0.71	2	0.65	2	0.54	3	0.47	3	0.42	4	0.33	4
	Long	1.00	8	0.82	2	0.65	2	0.58	3	0.46	3	0.40	4	0.35	4	0.28	4
Wind	Lat	1.00	8	1.00	8	0.89	2	0.81	2	0.70	2	0.62	3	0.56	3	0.47	3
	Long	1.00	8	0.95	2	0.80	2	0.73	2	0.62	3	0.54	3	0.49	3	0.40	4
	Speed	0.65	4	0.44	3	0.26	6	0.17	7	0.16	5	0.12	6	0.10	6	0.08	6

TABLE 4.3: Compression performance of the best evaluated coding algorithm, for various error values on each data type of each dataset. Each row contains information relative to certain data type. For each error parameter value, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the best window size for the best algorithm (the one that achieves the minimum CR), which is identified by a certain cell color described in the legend above the table.

We observe that there are four algorithms (PCA, APCA, CA, and FR) that obtain the best compression results in at least one of the 168 possible data type and error parameter combinations. Algorithm APCA is the best algorithm in exactly 132 combinations (79%), including every case in which $e \geq 10$, and most of the cases in which $e \in [1, 3, 5]$. PCA is the best algorithm in 31 combinations (18%), including most of the lossless cases. CA and FR are the best algorithms in only 2 (1%) and 3 (2%) combinations, respectively, in each case for a single data type.

Since there is not a single algorithm that obtains the best compression performance for every data type, it is useful to analyze how much is the RD between the best algorithm and the rest, for every experimental combination. With that in mind, next we define a pair of metrics.

Definition 4.4.2. The *maximum RD (maxRD)* of a coding algorithm $a \in A$ for certain error parameter $e \in E$ is given by

$$\text{maxRD}(a, e) = \max_{z,d} \left\{ \text{RD}(c^b(z, d, e), c_{<a_v, w^*_{global}, e>}) \right\}, \tag{4.9}$$

where the maximum is taken over all the combinations of data type z and dataset d, and we recall that $c^b(z, d, e)$ is the best CAI for z, d, e.

The maxRD metric is useful for assessing the compression performance of a coding algorithm a on the set of data types as a whole. Notice that maxRD is always non-negative. A satisfactory result (i.e. close to zero) can only be obtained when a achieves a good compression performance *for every data type*. In other words, bad compression performance even *on a single data type* yields a poor result for the maxRD metric altogether. When maxRD is equal to zero, a achieves the best compression performance for every combination. Analyzing the results in Table 4.3, we observe that $\text{maxRD}(\text{APCA}, e) = 0$ for every $e \geq 10$. Since the best algorithm is unique for every combination (i.e. exactly one algorithm obtains the minimum CR in every case), it is also true that, when $a \neq \text{APCA}$, $\text{maxRD}(a, e) > 0$ for every $e \geq 10$.

Definition 4.4.3. The *minmax RD (minmaxRD)* for certain error parameter $e \in E$ is given by

$$\text{minmaxRD}(e) = \min_{a \in A} \left\{ maxRD(a, e) \right\}, \tag{4.10}$$

and we refer to arg $\min_{a \in A}$ as the *minmax coding algorithm* for e.

Again, minmaxRD is always always non-negative. Notice that $\text{minmaxRD}(e) = 0$ for certain e, if and only if there exists a minmax coding algorithm a such that $\text{maxRD}(a, e) = 0$. Continuing the analysis from the previous paragraph, it should be clear that APCA is the minmax coding algorithm for every $e \geq 10$, since $\text{maxRD}(\text{APCA}, e) = 0$ for every $e \geq 10$.

Table 4.4 shows the $\text{maxRD}(a, e)$ obtained for every pair of coding algorithm variant $a_v \in V^*$ and error parameter $e \in E$. For each e, the cell corresponding to the $\text{minmaxRD}(e)$ value (i.e. the minimum value in the column) is highlighted.

	maxRD (%)							
Algorithm	e = 0	e = 1	e = 3	e = 5	e = 10	e = 15	e = 20	e = 30
PCA	40.51	42.27	53.11	62.01	71.71	75.28	77.15	80.21
APCA	33.25	15.64	9.00	29.96	0	0	0	0
PWLH	73.46	72.93	72.52	82.14	83.24	86.86	88.94	91.19
PWLHInt	29.72	29.85	37.17	59.17	61.68	69.96	74.72	79.89
CA	29.55	30.26	48.39	56.26	59.01	67.90	72.96	78.46
SF	85.84	85.77	85.26	85.17	84.36	83.64	83.17	82.88
FR	48.75	49.85	52.21	52.70	54.82	55.35	54.48	64.72
GAMPS	72.51	77.43	89.05	92.87	96.28	97.47	98.08	98.62

TABLE 4.4: $\text{maxRD}(a, e)$ obtained for every pair of coding algorithm variant $a_v \in V^*$ and error parameter $e \in E$. For each e, the cell corresponding to the $\text{minmaxRD}(a)$ value is highlighted.

In the lossless case, CA is the minmax coding algorithm, with minmaxRD being equal to 29.55%. This value is rather high, which means that none of the considered algorithms achieves a CR that is close to the minimum simultaneously *for every data type*. Recalling the results from Table 4.3 we notice that $e = 0$ is the only error parameter value for which the minmax coding algorithm doesn't obtain the minimum CR in any combination. In other words, when $e = 0$, CA is the algorithm that minimizes the RD with the best algorithm among every data type, even though it itself is not the best algorithm for any data type.

When $e \in [1, 3, 5]$, the minmax coding algorithm is always APCA, and the minmaxRD values are 15.64%, 9.00% and 29.96%, respectively. Again, these values are fairly high, so we would select the most convenient algorithm depending on the data type we want to compress. Notice that in the closest case (algorithm FR for $e = 5$), the second best maxRD (52.70%) is about 75% larger than the minmaxRD, which is a much bigger difference than in the lossless case.

When $e \geq 10$, the minmax coding algorithm is also always APCA, but in these cases the minmaxRD values are always 0. In the closest case (algorithm FR for $e = 20$) the second best maxRD is 54.48%. If we wanted to compress any data type with any of these error parameter values, we would pick algorithm APCA, since according to our experimental results, it always obtains the best compression results with a significant difference over the remaining algorithms.

4.4.1 Comparison with Algorithm gzip

In this subsection we consider the results obtained by the general-purpose lossless compression algorithm gzip [10]. This algorithm only operates in lossless mode (i.e. the error parameter can only be $e = 0$), and it doesn't have a window size parameter w. Therefore, for each data type z of each dataset d, we have a unique CAI (and obtain a unique CR value) for gzip.

In all our experiments with gzip we perform a column-wise compression of the dataset files, which, in general, yields a much better performance than a row-wise compression. This is due to the fact that in most of our datasets, there is a greater degree of temporal than spatial correlation between the signals. All the reported results are obtained with the "--best" option of gzip, which targets compression performance optimization [56].

Table 4.5 summarizes the compression performance results obtained by gzip and the other evaluated coding algorithms, for each data type of each dataset. Similarly to Table 4.3, each row contains information relative to a certain data type, and for each error parameter value, the first column shows the CR obtained by the best CAI, the second column shows the base-2 logarithm of its window size parameter (when applicable), and the cell color identifies the best algorithm. We point out that, for $e > 0$, we compare the gzip lossless result with the near-lossless results obtained by our algorithms. Although this comparison may seem unintuitive or even "unfair" at first, we present it to highlight the fact that there are cases where lossless gzip outperforms lossy variants.

We observe that, in the lossless case, algorithm gzip achieves the best compression results for 10 (48%) out of the 21 data types, while algorithms PCA and FR obtain the best results for 10 (48%) and 1 (4%) of the data types, respectively. Among the near-lossless cases, algorithms APCA, PCA, and FR obtain the best results in 106 (72%), 13 (9%) and 2 (1%) of the 147 possible data type and error parameter combinations. Even though gzip is a lossless algorithm, it still manages to outperform our near-lossless algorithms in the remaining 26 (18%) combinations.

We notice that gzip obtains the best relative results against the other algorithms for smaller values of e, which is expected, since the performance of near-lossless algorithms improves for larger values of e. However, even for $e = 0$, gzip is outperformed by the rest of the algorithms in about a half (11 out of 21) of the data types. There are 6 data types for which it is even outperformed by algorithm Base (these are the data types for which the minimum CR is equal to 1). Algorithm APCA is still the best algorithm for most of the cases in which $e \geq 3$. However, now there is no value of e for which APCA outperforms the rest of the algorithms for every data type, since gzip is the best algorithm for at least one data type in every case. In particular, gzip obtains the best compression results for the data type "Size" of the dataset Hail for every e.

GZIP	PCA	APCA	FR

		e = 0		e = 1		e = 3		e = 5		e = 10		e = 15		e = 20		e = 30	
Dataset	Data Type	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w
IRKIS	VWC	0.13		0.13		0.12	5	0.07	6	0.03	7	0.02	8	0.02	8	0.01	8
SST	SST	0.52		0.28	3	0.14	5	0.09	6	0.05	7	0.03	8	0.02	8	0.02	8
ADCP	Velocity	0.61		0.61		0.61		0.61	2	0.48	2	0.41	2	0.35	3	0.26	3
Solar	GHI	0.69		0.69		0.69		0.67	4	0.59	4	0.52	4	0.47	4	0.38	4
	DNI	0.67		0.67		0.66	4	0.61	4	0.54	4	0.49	4	0.43	4	0.36	4
	DHI	0.61		0.61		0.61		0.61		0.60	4	0.54	4	0.48	4	0.39	4
ElNino	Lat	0.08		0.08		0.08		0.08		0.08		0.08		0.08		0.06	6
	Long	0.07		0.07		0.07		0.07		0.07		0.07	6	0.05	7	0.02	8
	Zon. Wind	0.31	8	0.31	8	0.31	8	0.31	8	0.27	2	0.24	2	0.21	2	0.16	3
	Mer. Wind	0.31	8	0.31	8	0.31	8	0.31	8	0.29	2	0.26	2	0.23	2	0.19	2
	Humidity	0.23	8	0.23	8	0.23	8	0.23	8	0.21	2	0.18	2	0.16	2	0.13	2
	Air Temp.	0.33	8	0.33	8	0.30	2	0.27	2	0.22	2	0.19	3	0.17	3	0.13	4
	Sea Temp.	0.32		0.31	2	0.25	2	0.21	2	0.14	3	0.11	4	0.08	4	0.05	5
Hail	Lat	1.00	8	1.00	8	0.90	2	0.83	2	0.71	2	0.65	3	0.57	3	0.47	3
	Long	1.00	8	1.00	8	0.86	2	0.78	2	0.65	2	0.55	3	0.49	3	0.39	4
	Size	0.37		0.37		0.37		0.37		0.37		0.37		0.37		0.37	
Tornado	Lat	1.00	8	0.85	2	0.71	2	0.65	2	0.54	3	0.47	3	0.42	4	0.33	4
	Long	1.00	8	0.82	2	0.65	2	0.58	3	0.46	3	0.40	4	0.35	4	0.28	4
Wind	Lat	1.00	8	1.00	8	0.89	2	0.81	2	0.70	2	0.62	3	0.56	3	0.47	3
	Long	1.00	8	0.95	2	0.80	2	0.73	2	0.62	3	0.54	3	0.49	3	0.40	4
	Speed	0.65	4	0.44	3	0.26	6	0.17	7	0.16	5	0.12	6	0.10	6	0.08	6

TABLE 4.5: Compression performance of the best evaluated coding algorithm, for various error values on each data type of each dataset, including the results obtained by gzip. Each row contains information relative to certain data type. Each row contains information relative to certain data type. For each error parameter value, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the best window size for the best algorithm (the one that achieves the minimum CR), which is identified by a certain cell color described in the legend above the table. Algorithm gzip doesn't have a window size parameter, so the cell is left blank in these cases.

Similarly to Table 4.4, Table 4.6 shows the maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^* \cup \{\text{gzip}\}$ and error parameter $e \in E$. For each e, the cell corresponding to the minmaxRD(e) value is highlighted.

We observe that, for every e, the minmaxRD values are rather high, the minimum being 26.66% (algorithm gzip for $e = 0$). We conclude that none of the considered algorithms achieves a competitive CR *for every data type*, and the selection of the most convenient algorithm depends on the specific data type we are interested in compressing.

gzip is the minmax coding algorithm when $e \in [0, 1]$, and in both cases the minmaxRD values are rather high, i.e. 26.66% and 47.99%, respectively. APCA remains the minmax coding algorithm for every $e \geq 3$, but its minmaxRD values are now not only always greater than zero, but also quite high, ranging from 42.85% ($e = 30$) up to 54.36% ($e = 3$). This implies that there exist some data types for which the RD between the APCA and gzip CAIs is considerable, which means that, if we have the possibility of selecting gzip as a compression algorithm, APCA is no longer the obvious choice for compressing every data type when $e \geq 10$, as we had concluded in the previous section.

	maxRD (%)							
Algorithm	e = 0	e = 1	e = 3	e = 5	e = 10	e = 15	e = 20	e = 30
GZIP	26.66	47.99	73.79	82.94	91.10	93.94	95.40	96.69
PCA	73.93	73.54	68.27	67.21	71.71	75.28	77.15	80.21
APCA	59.11	58.39	54.36	54.35	54.34	54.33	54.32	42.85
PWLH	87.51	87.45	87.35	87.26	87.01	86.86	88.94	91.19
PWLHInt	71.26	70.81	64.80	63.74	61.97	69.96	74.72	79.89
CA	71.19	70.71	64.63	63.45	61.64	67.90	72.96	78.46
SF	93.41	92.72	92.64	92.25	91.82	91.57	91.37	90.21
FR	76.46	76.12	72.78	71.97	67.37	64.31	64.00	64.72
GAMPS	83.73	83.73	89.05	92.87	96.28	97.47	98.08	98.62

Table 4.6: maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^* \cup \{\text{gzip}\}$ and error parameter $e \in E$. For each e, the cell corresponding to the minmaxRD(a) value is highlighted.

4.4.2 Timestamp Compression

In this subsection we study the compression of the timestamp column of the various datasets introduced in Chapter 2. Recall, from Section 2.1, that this column represents the timestamps associated with the data from the rest of the columns in a dataset CSV file. Since the timestamp column does no contain any gaps, we only consider the non-masking variants of the evaluated algorithms. Notice that this is the single place in the current section in which we do so. In this case, we only evaluate the compression performance of the constant model algorithm variants, i.e. PCA$_{NM}$ and APCA$_{NM}$. Recall, from Section 3.7, that the encoding scheme for linear model algorithms requires encoding a sequence of line segments in the two-dimensional Euclidean space, where the x and y-axis correspond to timestamp and sample values, respectively. Thus, this type of algorithms cannot be used for encoding a single column. We also discarded variant GAMPS$_{NM}$, since correlation model algorithms are not ideal for encoding a single column. The timestamp column must always be losslessly compressed, so in our experiments we only considered the error parameter $e = 0$.

Table 4.7 summarizes the timestamp compression performance results obtained by PCA$_{NM}$ and APCA$_{NM}$. Each row contains information relative to certain dataset. For example, the second row shows summarized results for the timestamp column of the dataset SST. In this case, the minimum CR for variant PCA$_{NM}$ is 0.11, which is obtained with window size parameter $w = 2^4$, while the minimum CR for variant APCA$_{NM}$ is 0.01, which is obtained with $w = 2^8$. The RD between the corresponding CAIs, as defined in (4.5), is equal to -791.41%.

We observe that APCA$_{NM}$ obtains better compression results in the first five datasets, while PCA$_{NM}$ works best in the last three. This can be explained due to the characteristics of the timestamp signal: in the first five datasets it is smooth, while in the last three it is rough. The RD between the corresponding CAIs is much larger in the first five datasets, ranging from -125.77% (in dataset IRKIS) to -841% (in dataset ADCP); in the last three datasets it ranges from 1.05% (in dataset Wind) to 8.06% (in dataset Tornado). Thus, the results indicate that APCA$_{NM}$ performs better in general, obtaining significantly better compression results in some cases, and marginally worse results in others. Since it is the most robust variant for timestamp compression among our evaluated algorithm variants, our general encoding scheme always encodes the timestamp column using APCA$_{NM}$ (recall Line 4 in Figure 3.1).

	PCA$_{NM}$		APCA$_{NM}$		
Dataset	CR	w	CR	w	RD(PCA$_{NM}$, APCA$_{NM}$) (%)
IRKIS	0.01	7	0.01	8	-125.77
SST	0.11	4	0.01	8	-791.41
ADCP	0.08	5	0.01	8	-841
Solar	0.03	6	0.01	8	-319.83
ElNino	0.04	6	0.01	8	-443.72
Hail	1.00	8	1.06	2	5.74
Tornado	1.00	8	1.09	2	8.06
Wind	1.00	8	1.01	2	1.05

Table 4.7: Lossless compression performance of algorithm variants PCA$_{NM}$ and APCA$_{NM}$, for the timestamp column of each dataset. For each variant, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the best window size (the window size that achieves the minimum CR). For each dataset, the cells corresponding to the algorithm that obtains the best result are colored, and the last column shows the RD between the corresponding CAIs, as defined in (4.5).

Chapter 5

Conclusions and Future Work

In this work, we study the compression of multichannel signals with irregular sampling rates and with data gaps. Our experiments consider different real-word datasets consisting of signals with these characteristics, which are often gathered by WSNs. In such case, the irregular sampling rates can be caused by different groups of sensors being out of sync or malfunctioning. Besides, errors might arise when acquiring, transmitting or storing the data, causing gaps in the data. Nevertheless, state-of-the-art algorithms designed for sensor data compression reported in the literature [27, 28] assume, in general, that the signals gave regular sampling rate and that there are no data gaps. Thus, our main contributions are introducing and analyzing the compression performance of a number of variants of state-of-the-art algorithms, which we design and implement so that they are able to encode multichannel signals with irregular sampling rates and data gaps.

The experimental datasets come from multiple sources [29–33], each using a different data representation format. Thus, we define our own format, into which we transformed all the experimental datasets. This format can be used to represent various kinds of datasets, and our implemented algorithm variants should be able to encode any data file that follows its specifications.

For most algorithms we implement two variants, *masking (M)* and *non-masking (NM)*. They differ in the technique used for encoding the position of the data gaps. In variant M, the position of all the data gaps is encoded first, and the data values are encoded separately, next. Our proposed strategy to compress the gaps in variant M uses arithmetic coding [41, 42] combined with a Krichevsky-Trofimov probability assignment [43] over a Markov model. On the other hand, in variant NM, the gaps and the data values are encoded together. With both variants, the gaps in a decompressed signal always match the gaps in the original signal, regardless of the value of the error threshold parameter (ϵ).

The main objective of our experimental work is to analyze the compression performance of our implemented algorithm variants by encoding our various experimental datasets. For each variant, we test several combinations of parameters. In particular, we consider an extended range of values for the error threshold parameter, including the *lossless compression* case (i.e. $\epsilon = 0$). We asses the compression performance of an algorithm variant through the *compression ratio (CR)* metric (see Definition 4.1.4), and we compare the compression performance between a pair of algorithm variants through the *relative difference (RD)* metric (see Definition 4.1.5).

Our first analysis consists of comparing the compression performance of our implemented variants a_M and a_{NM}, for each algorithm a that supports both variants. The results show that, on experimental datasets with few or no gaps, both variants have roughly the same performance, i.e. $\mathrm{RD}(a_M, a_{NM})$ is always close to zero, ranging between -0.29 and 1.76%. However, on datasets with many gaps, a_M always performs better, sometimes with a significant difference, with $\mathrm{RD}(a_M, a_{NM})$ ranging between 2.44 and 50.78%. Thus, our experimental results suggest that variant M is more robust, and performs better in general.

Both state-of-the-art algorithms and our implemented variants depend on a window size parameter, w. It defines a *fixed window size* for both variants of algorithm PCA, and a *maximum window size* for the rest of the variants. Our experiments consider an extended range of values for parameter w, which allows us to analyze the sensitivity of the different algorithm variants to this parameter. For each dataset, we compress each data file, and compare the results obtained when using a window size optimized for said specific file, against the results obtained when using a window size optimized for the whole dataset. The results indicate that the difference in compression performance is generally rather small, with the RD being less than or equal to 2% in 97.8% of the experimental cases. Obtaining the optimal w for a specific data file is, in general, a computationally expensive task, so these results have meaningful practical consequences, since they imply that we could fix parameter w in advance, for example by optimizing over a training set, without compromising the overall compression performance of an algorithm variant.

The most significant part of our experimental analysis consists of comparing the compression performance of our implemented algorithm variants, with each other, and with the general-purpose lossless compression algorithm gzip [10]. Following the experimental results presented in the previous two paragraphs, for this analysis we only consider variant M of each algorithm, and we always use the optimal window size for the whole dataset. Our experimental results reveal that none of the variants obtains the best compression performance in every scenario, which means that the optimal selection of a variant depends on the characteristics of the data to be compressed, and the error threshold (ϵ) that is allowed. In some cases, even a general-purpose compression algorithm such as gzip outperforms the specific variants. Nonetheless, some general conclusions can be obtained from our analysis. For large error thresholds, variant APCA_M achieves the best results, obtaining the minimum CR in 90.5% of the experimental cases. On the other hand, algorithm gzip and variant PCA_M are preferred for lower thresholds scenarios, since they obtain the minimum CR in 42.9% and 40.5% of the cases, respectively. We also analyze the lossless compression of the timestamp column, and the results indicate that variant APCA_{NM} generally achieves the best results.

We conclude this chapter suggesting a couple of ideas to develop as future work:

- In our experimental analysis, which is presented in Chapter 4, we asses the compression performance of the implemented algorithm variants through the CR metric (recall Definition 4.1.4). It would be interesting to consider additional metrics, such as the computational time and the sensitivity to outliers [27]. We could also consider additional datasets, which would increase the running time of our experiments, but may provide new insights regarding the compression performance of the implemented algorithms.

- The results presented in Section 4.4 reveal *which* of the implemented algorithm variants obtains better compression results for each data type in the experimental datasets. This information could be taken into account as part of a strategy to implement a universal coder (i.e. a coder that compresses signals from which no information is known a priori). For this purpose, it would be useful to analyze *why* it is the case that a certain variant achieves better compression results for a certain data type over other variants. In order to do this, we would need to examine the signals for each data type, analyze their characteristics (e.g. whether they are smooth or rough signals, the number of outliers, periodicity), and observe if there exists a relation between these characteristics and the algorithm variant that obtains the best compression performance. This would be useful for predicting which variant is the best for compressing certain signal, only by analyzing the signal characteristics. If, given certain statistics of a signal, we could programmatically select a good compression algorithm variant for the signal, this could prove to be beneficial for online compression, as it would allow a universal coder to select a different variant as the trends in the signal vary over time.

Acknowledgements

The author would like to express his sincere gratitude to his supervisors, Álvaro Martín and Gadiel Seroussi, for their time and knowledge.

He would also like to thank *Programa de Desarrollo de las Ciencias Básicas (PEDECIBA)* and *Universidad de la República (Udelar)* for their support.

Bibliography

[1] V. Turner, D. Reinsel, J.F. Gantz, and S. Minton. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. *IDC Analyze the Future*, 2014.

[2] D. Reinsel, J. Gantz, and J. Rydning. The Digitization of the World: From Edge to Core. *IDC White Paper*, 2018.

[3] T. Pendlebury. Spotify vs. Apple Music: Is there a difference in sound quality? http://www.cnet.com/news/apple-music-vs-spotify-is-there-a-difference-in-sound-quality/, July 5, 2015. [Accessed May 5, 2022].

[4] G. Morrison. What is HEVC? High Efficiency Video Coding, H.265, and 4K compression explained. http://www.cnet.com/news/what-is-hevc-high-efficiency-video-coding-h-265-and-4k-compression-explained/, April 18, 2014. [Accessed May 5, 2022].

[5] M. Robin and M. Poulin. *Digital Television Fundamentals.* McGraw-Hill Education, 2nd edition, 2000.

[6] T. Hoff. A Short on How Zoom Works. http://highscalability.com/blog/2020/5/14/a-short-on-how-zoom-works.html, May 14, 2020. [Accessed May 5, 2022].

[7] B. Sat and B.W. Wah. Analysis and Evaluation of the Skype and Google-Talk VoIP Systems,. *Proc. 2006 IEEE International Conference on Multimedia and Expo (ICME 2006)*, pages 2153–2156, 2006.

[8] Nikon Corporation. DSLR Camera Basics: Image Quality and File Type (NEF/RAW, JPEG, and TIFF). https://imaging.nikon.com/lineup/dslr/basics/26/01.htm. [Accessed May 5, 2022].

[9] B. Carnathan. Should I Use Canon's C-RAW Image File Format? https://www.the-digital-picture.com/Canon-Cameras/Canon-C-RAW-Image-File-Format.aspx. [Accessed May 5, 2022].

[10] GNU Gzip. https://www.gnu.org/software/gzip/, 2020. [Accessed May 5, 2022].

[11] WinRAR at a glance. https://www.win-rar.com/features.html, 2021. [Accessed May 5, 2022].

[12] A.J. Casson, D.C. Yates, S.J.M. Smith, J.S. Duncan, and E. Rodriguez-Villegas. Wearable electroencephalography: What Is It, Why Is It Needed, and What Does It Entail? *IEEE Eng. Med. Biol. Mag.*, 29(3):44–56, 2010.

[13] H. Mamaghanian, N. Khaled, D. Atienza, and P. Vandergheynst. Compressed Sensing for Real-Time Energy-Efficient ECG Compression on Wireless Body Sensor Nodes. *IEEE Trans. Biomed. Eng.*, 58(9):2456–2466, 2011.

[14] J. Beckett. HP on Mars: Labs technology used to send most accurate images possible. https://www.hpl.hp.com/news/2004/jan-mar/hp_mars.html, January 2004. [Accessed May 5, 2022].

[15] M. Tramz. Photographing Pluto: This Is How New Horizons Works. https://time.com/3944157/new-horizons-pluto/, July 14, 2015. [Accessed May 5, 2022].

[16] K. Sohraby, D. Minoli, and T. Znati. *Wireless Sensor Networks: Technology, Protocols, and Applications.* Wiley-Interscience, 2007.

[17] A. Ali, Y. Ming, S. Chakraborty, and S. Iram. A Comprehensive Survey on Real-Time Applications of WSN. *Future Internet*, 9(4):77, 2017.

[18] J. Uthayakumar, T. Vengattaraman, and P. Dhavachelvan. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University - Computer and Information Sciences*, page 77, 2018.

[19] A.J. Hussain, A. Al-Fayadh, and N. Radi. Image compression techniques: A survey in lossless and lossy algorithms. *Neurocomputing*, 300:44–69, 2018.

[20] G. Vijayvargiya, S. Silakari, and R. Pandey. A Survey: Various Techniques of Image Compression. *International Journal of Computer Science and Information Security (IJCSIS)*, 11(10), 2013.

[21] M. Hans and R.W. Schafer. Lossless compression of digital audio. *IEEE Signal Processing Magazine*, 18(4):21–32, 2001.

[22] K. Brandenburg. MP3 and AAC explained. *AES 17th International Conference on High-Quality Audio Coding*, 1999.

[23] J.-S. Lee and T. Ebrahimi. Perceptual Video Compression: A Survey. *IEEE Journal of Selected Topics in Signal Processing*, 6(6):684–697, 2012.

[24] R.V. Babu, M. Tom, and P. Wadekar. A survey on compressed domain video analysis techniques. *Multimedia Tools and Applications*, 75(2):1043–1078, 2014.

[25] I. Capurro, F. Lecumberry, A. Martin, I. Ramirez, E. Rovira, and G. Seroussi. Efficient Sequential Compression of Multichannel Biomedical Signals. *IEEE Journal of Biomedical and Health Informatics*, 21(4):904–916, 2017.

[26] A. Naït-Ali and C. Cavaro-Ménard. *Compression of Biomedical Images and Signals.* ISTE-Wiley, 2008.

[27] N.Q.V. Hung, H. Jeung, and K. Aberer. An Evaluation of Model-Based Approaches to Sensor Data Compression. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2434–2447, 2013.

[28] T. Bose, S. Bandyopadhyay, S. Kumar, A. Bhattacharyya, and A. Pal. Signal Characteristics on Sensor Data Compression in IoT - An Investigation. *2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–6, 2016.

[29] N. Wever. IRKIS Soil moisture measurements Davos. SLF. https://doi.org/10.16904/17, 2017. [Accessed May 5, 2022].

[30] El Nino Data Set. https://archive.ics.uci.edu/ml/datasets/El+Nino, 1999. [Accessed May 5, 2022].

[31] NOAA - TAO Data Download. https://tao.ndbc.noaa.gov/tao/data_download/search_map.shtml, 2016. [Accessed May 5, 2022].

[32] SolarAnywhere - Data. https://data.solaranywhere.com/Data, 2020. [Accessed May 5, 2022].

[33] Storm Prediction Center - Severe Weather Database Files (1950-2017). https://www.spc.noaa.gov/wcm/#data, 2016. [Accessed May 5, 2022].

[34] I. Lazaridis and S. Mehrotra. Capturing Sensor-Generated Time Series with Quality Guarantees. *Proc. ICDE*, pages 429–440, 2003.

[35] E. Keogh, K. Chakrabarti, S. Mehrotra, and M. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *ACM Transactions on Database Systems*, 27(2):188–228, 2002.

[36] C. Buragohain, N. Shrivastava, and S. Suri. Space Efficient Streaming Algorithms for the Maximum Error Histogram. *Proc. ICDE*, pages 1026–1035, 2007.

[37] G.E. Williams. Critical Aperture Convergence Filtering and Systems and Methods Thereof. *US Patent 7,076,402*, Jul. 11, 2006.

[38] H. Elmeleegy, A.K. Elmagarmid, E. Cecchet, W.G. Aref, and W. Zwaenepoel. Online Piecewise Linear Approximation of Numerical Streams with Precision Guarantees. *Proc. VLDB Endowment*, 2(1):145–156, 2009.

[39] J.A.M. Heras and A. Donati. Fractal Resampling: time series archive lossy compression with guarantees. *PV 2013 Conference*, 2013.

[40] S. Gandhi, S. Nath, S. Suri, and J. Liu. GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling. *Proc. ACM SIGMOD International Conference on Management of Data*, pages 771–784, 2009.

[41] Arithmetic Coding + Statistical Modeling = Data Compression. https://marknelson.us/posts/1991/02/01/arithmetic-coding-statistical-modeling-data-compression.html, 1991. [Accessed May 5, 2022].

[42] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2nd edition, 2006.

[43] R.E. Krichevsky and V.K. Trofimov. The performance of universal encoding. *IEEE Transactions on Information Theory*, 27(2):199–207, 1981.

[44] N. Wever, F. Comola, M. Bavay, and M. Lehning. Simulating the influence of snow surface processes on soil moisture dynamics and streamflow generation in an alpine catchment. *Hydrol. Earth Syst. Sci.*, 21:4053–4071, https://doi.org/10.5194/hess-21-4053-2017, 2017.

[45] TAO - History of the Array. https://tao.ndbc.noaa.gov/proj_overview/taohis_ndbc.shtml, 2013. [Accessed May 5, 2022].

[46] A.F. Cheng, S.E. Hawkins III, L. Nguyen, C.A. Monaco, and G.G. Seagrave. Data Compression Using Chebyshev Transform. *US Patent 7,249,153*, 2007.

[47] T. Sheltami, M. Musaddiq, and E. Shakshuki. Data Compression Techniques in Wireless Sensor Networks. *Future Generation Computer Systems*, 64:151–162, 2016.

[48] G.K. Wallace. The JPEG Still Picture Compression Standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[49] J.J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

[50] J. Rissanen. Universal Coding, Information, Prediction, and Estimation. *IEEE Transactions on Information Theory*, 30(4):629–636, 1984.

[51] E.N. Gilbert. Capacity of a Burst-Noise Channel. *Bell Syst. Tech. J.*, 39(5):1253–1265, 1960.

[52] E.O. Elliott. Estimates of Error Rates for Codes on Burst-Noise Channels. *Bell Syst. Tech. J.*, 42(5):1977–1997, 1963.

[53] R. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1:132–133, 1972.

[54] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, 1987.

[55] Data Compression With Arithmetic Coding. `https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html`, 2014. [Accessed May 5, 2022].

[56] GNU Gzip Manual - Invoking gzip. `https://www.gnu.org/software/gzip/manual/html_node/Invoking-gzip.html`, 2018. [Accessed May 5, 2022].