

UNIVERSIDAD DE LA REPÚBLICA

TESIS DE MAESTRÍA

Coding of Multichannel Signals with Irregular Sampling Rate

Autor:

Pablo Cerveñansky

Supervisores:

Álvaro Martín

Gadiel Seroussi

Núcleo de Teoría de la Información

Facultad de Ingeniería

February 23, 2021

Abstract

The staggering development of mobile communications and the Internet has 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 to reduce the number of bits needed to represent certain digital data, have become increasingly relevant. In this work, we focus on the compression of multichannel signals with irregular sampling rate and data gaps. We consider the state-of-the-art algorithms, which are used to compress gapless signals with regular sampling, adapt them to operate with signals with irregular sampling rate and data gaps, 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, and they compress signals by exploiting correlation among signal samples taken at close times (temporal correlation), and, in some cases, among samples from various signals (spatial correlation). For most algorithms we design and implement two variants: the masking (M) variant, which first encodes the position of all the gaps, and then proceeds to encode the data values separately, and the 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 blocks into which the data are partitioned for encoding. We analyze the sensibility of variant M of each algorithm 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. Our experimental results indicate that the difference in compression performance is generally rather small. The last part of our experimental analysis consists in comparing the compression performance achieved by our adapted algorithms. 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 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 variant. Nevertheless, we extract some general conclusions from our analysis: for large error thresholds, variant M of algorithm APCA achieves the best compression results, while algorithm gzip and variant M of algorithm PCA are preferred for lower threshold scenarios.

Keywords— multichannel signal compression, near-lossless compression, 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	14
3.1 Introduction	15
3.2 General Encoding Scheme	17
3.3 Encoding of Gaps in the Masking Variants	19
3.4 Algorithm Base	20
3.5 Algorithm PCA	20
3.5.1 Example	23
3.5.2 Non-Masking (<i>NM</i>) Variant	25
3.6 Algorithm APCA	26
3.6.1 Example	27
3.6.2 Non-Masking (<i>NM</i>) Variant	29
3.7 Encoding Scheme for Linear Model Algorithms	29
3.8 Algorithms PWLH and PWLHInt	30
3.8.1 Example	34
3.8.2 Differences Between Algorithms PWLH and PWLHInt	38
3.8.3 Non-Masking (<i>NM</i>) Variant	39
3.9 Algorithm CA	40
3.9.1 Example	42
3.9.2 Non-Masking (<i>NM</i>) Variant	47
3.10 Algorithm SF	47

3.10.1 Example	53
3.11 Algorithm FR	58
3.11.1 Example	61
3.12 Algorithm GAMPS	64
3.12.1 Example	65
3.12.2 Non-Masking (<i>NM</i>) variant	67
3.13 Implementation Details	67
4 Experimental Results	68
4.1 Experimental Setting	69
4.2 Comparison of Masking and Non-Masking Variants	71
4.3 Window Size Parameter	75
4.4 Algorithms Performance	79
4.4.1 Comparison With Algorithm gzip	83
5 Conclusions and Future Work	86
Bibliography	88

List of Figures

3.1	General encoding scheme for every evaluated algorithm.	17
3.2	Markov process diagram.	19
3.3	Coding routine for algorithm Base.	20
3.4	Coding routine for variant M of algorithm PCA.	22
3.5	Decoding routine for variant M of algorithm PCA.	22
3.6	Example: Signal consisting of 12 samples.	23
3.7	Example: Algorithm PCA with $\epsilon = 1$ and $w = 4$. Step 1.	23
3.8	Example: Algorithm PCA with $\epsilon = 1$ and $w = 4$. Step 2.	24
3.9	Example: Algorithm PCA with $\epsilon = 1$ and $w = 4$. Step 3.	24
3.10	Coding routine for variant NM of algorithm PCA.	25
3.11	Coding routine for variant M of algorithm APCA.	26
3.12	Auxiliary routine EncodeWindow for algorithm APCA.	26
3.13	Decoding routine for variant M of algorithm APCA.	27
3.14	Example: Algorithm APCA with $\epsilon = 1$ and $w = 256$. Step 1.	27
3.15	Example: Algorithm APCA with $\epsilon = 1$ and $w = 256$. Step 2.	28
3.16	Example: Algorithm APCA with $\epsilon = 1$ and $w = 256$. Step 3.	28
3.17	Auxiliary routine FillSegment for linear model algorithms.	29
3.18	Example of the auxiliary routine FillSegment for linear model algorithms.	30
3.19	Coding routine for variant M of algorithm PWLH.	31
3.20	Auxiliary routine EncodeWindow for algorithm PWLH.	31
3.21	Example: checking the valid hull condition. Algorithm PWLH with $\epsilon = 1$. Step 1.	32
3.22	Example: checking the valid hull condition. Algorithm PWLH with $\epsilon = 1$. Step 2.	32
3.23	Auxiliary routine EncodeLastWindow for algorithm PWLH.	32
3.24	Decoding routine for variant M of algorithm PWLH.	33
3.25	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 1.	34
3.26	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 2.	34
3.27	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 3.	35
3.28	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 4.	35
3.29	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 5.	36
3.30	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 6.	36
3.31	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 7.	36
3.32	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 8.	37
3.33	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 9.	37
3.34	Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 10.	38
3.35	Coding routine for variant M of algorithm CA.	40
3.36	Auxiliary routine EncodeWinStart for algorithm CA.	41
3.37	Auxiliary routine EncodeWinEnd for algorithm CA.	41
3.38	Example: key steps of the coding algorithm. Algorithm CA with $\epsilon = 1$. Step 1.	41
3.39	Example: key steps of the coding algorithm. Algorithm CA with $\epsilon = 1$. Step 2.	41
3.40	Decoding routine for variant M of algorithm CA.	42
3.41	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 1.	43
3.42	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 2.	43

3.43	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 3.	44
3.44	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 4.	44
3.45	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 5.	45
3.46	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 6.	45
3.47	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 7.	46
3.48	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 8.	46
3.49	Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 9.	47
3.50	Coding routine for variant M of algorithm SF.	48
3.51	Auxiliary routine EncodePoint for algorithm SF.	49
3.52	Auxiliary routine EncodeWinStart for algorithm SF.	49
3.53	Auxiliary routine EncodeWinEndStart for algorithm SF.	50
3.54	Auxiliary routine EncodeWinEnd for algorithm SF.	50
3.55	Example: key steps of the coding algorithm. Algorithm SF with $\epsilon = 1$. Step 1. .	50
3.56	Example: key steps of the coding algorithm. Algorithm SF with $\epsilon = 1$. Step 2. .	50
3.57	Decoding routine for variant M of algorithm SF.	52
3.58	Auxiliary routine DecodePoint for algorithm SF.	52
3.59	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 1.	53
3.60	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 2.	53
3.61	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 3.	54
3.62	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 4.	54
3.63	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 5.	55
3.64	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 6.	55
3.65	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 7.	56
3.66	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 8.	56
3.67	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 9.	57
3.68	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 10.	57
3.69	Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 11.	58
3.70	Coding routine for variant M of algorithm FR.	59
3.71	Auxiliary routine GetDisPointsMDP for algorithm FR.	59
3.72	Decoding routine for variant M of algorithm FR.	60
3.73	Example: Algorithm FR with $\epsilon = 1$ and $w = 256$. Step 1.	61
3.74	Example: Algorithm FR with $\epsilon = 1$ and $w = 256$. Step 2.	62
3.75	Example: Algorithm FR with $\epsilon = 1$ and $w = 256$. Step 3.	62
3.76	Example: Algorithm FR with $\epsilon = 1$ and $w = 256$. Step 4.	63
3.77	Coding routine for variant M of algorithm GAMPS.	64
3.78	Example: Three signals consisting of 12 samples each.	65
3.79	Example: Algorithm GAMPS with $\epsilon = 0$ and $w = 256$. Step 1.	66
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.	72
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.	73
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.	76
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 algorithm PCA we highlight with a red circle the marker for the maximum value (10.6%) obtained for all the tested CAIs.	77

-
- 4.5 CR and window size parameter plots for every evaluated algorithm, for the data type “SST” 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 algorithm) 80

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 coding algorithms. For each algorithm, the table shows whether it supports lossless and near-lossless compression (second and third columns, respectively), its compression model (fourth column), whether the masking (M) and non-masking (NM) variants apply (fifth and sixth columns, respectively), and whether the algorithm depends on a window size parameter (w) (last column).	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.	74
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.	78
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.	81
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.	82
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.	84
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.	85

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 ZB¹ 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 to reduce the number of bits needed to represent certain 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).

The role of data compression has become increasingly relevant, in both the business 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 reduce 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].

¹A *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 preventing it from running out of battery [12, 13].
- NASA's space missions involve transmitting information back to Earth from space probes in far away places, which implicates consuming 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 key using 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 transmitted², and whether the energy resources are critical or not³.

In this work, we focus on the compression of multichannel signals with irregular sampling rate and data gaps. There is a wide range of research on compression algorithms for multichannel signals with *regular sampling rate* (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 rate*. This occurs frequently in datasets gathered by WSNs⁴, 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, the state-of-the-art algorithms used 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. Thus, in this thesis we design and implement a number of variants of the state-of-the-art algorithms, which are able to encode multichannel signals with irregular sampling rate and data gaps, then analyze their compression performance.

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 and Wind [33] consist of data related to thunderstorms and tornadoes along US territory. Each experimental dataset consists of multichannel signals with either one or both of the characteristics

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

³In 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.

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

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 slowly varying or rough signals, the number of outliers, periodicity), varies considerably among different signals, which is beneficial for analyzing the performance of the evaluated algorithms. The datasets come from multiple sources [29–33], each using a different data representation format. Thus, we transformed all of the data into an homogeneous format, which we define ourselves, and can be easily adapted to represent different kind 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 0, the compression is lossless, i.e., the decompressed and the original signals are identical. Besides, the original algorithms and our proposed variants follow a model-based compression approach that compresses signals by exploiting correlation among signal samples taken at close times (*temporal correlation*) and, in some cases, among samples from various signals (*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 yield poor compression results [27]. 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).

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 a practical scenario. 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 equal to the size of any given data compressed by said algorithm variant, divided by the size of the original uncompressed data. Thus, smaller values of CR correspond to a better performance. To compare the compression performance between a pair of algorithm variants, we consider the *relative difference (RD)* metric, which is equal to the difference between the sizes of certain data compressed by each variant, divided by the size of

the data compressed by one of the variants, expressed as a percentage. In this case, given a pair of algorithm variants, a_1 and a_2 , a_1 has a better performance than a_2 iff $\text{RD}(a_1, a_2)$ is positive.

In Section 4.2, for each algorithm a that supports variants M and NM ⁵, 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. $\text{RD}(a_M, a_{NM})$ 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 $\text{RD}(a_M, a_{NM})$ 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 blocks into which the data are partitioned for encoding. In algorithm PCA, it defines a *fixed block size*, while in the rest of the algorithms it defines a *maximum block size*. In Section 4.3 we analyze the sensibility of variant M of each algorithm⁶ 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 97.7% of the experimental cases. Therefore, we could fix the window size parameter in advance, for example by optimizing over a training set, without 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 in comparing the compression performance of our adapted algorithm variants, among each other, and with the general-purpose compression algorithm gzip [10]. Taking into account the results presented 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 selection of an algorithm 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.

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

⁵Algorithms SF and FR only support variant M , while the rest of the algorithms support variants M and NM .

⁶Due 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 either 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 all the data into a consistent, homogeneous 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 the following eight sections we present every dataset, 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. In Section 2.10 we summarize the information presented in this chapter, specifying the number of files, the data types, and the number of gaps 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 “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 set in the buoy. Additional information on dataset ElNino 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 figures, 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 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 ($^{\circ}\text{C}$), 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 $^{\circ}\text{C}$).

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 $^{\circ}\text{C}$), 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)	0	1,004	9	186.0	269.7
DNI	0 (0.0)	0	958	3	178.3	272.8
DHI	0 (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 (0.0)	2,570	4,932	3,845	3,854.7	475.7
Long (coord. degrees)	0 (0.0)	-12,442	-6,783	-9,676	-9,487.4	841.4
Size (1/100 inch)	0 (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 (0.0)	2,456	4,891	3,688	3,693.9	491.1
Long	0 (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 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 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)	0	4,899	3,776	3,778.4	460.8
Long (coord. degrees)	0 (0.0)	-12,432	-8	-8,732	-8,870.1	968.4
Speed (mph)	0 (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 characteristics 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	Characteristics	#Files	#Types	Data Types
IRKIS [29]	Many gaps	7	1	VWC
SST [31]	Many gaps	3	1	SST
ADCP [31]	Many gaps	3	1	Vel
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 displayed in the CSV file for each dataset. 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
IRKIS	Time Delta	minutes	1	0	13,1071
	VWC	dimensionless	1,000	0	600
SST	Time Delta	seconds	1	0	13,1071
	SST	°C	1,000	0	40,000
ADCP	Time Delta	minutes	1	0	13,1071
	Vel	m/s	1,000	-1,100	2,700
ElNino	Time Delta	hours	1	0	13,1071
	Lat	coord. degrees	100	-1,000	1,000
	Long	coord. degrees	100	-18,000	18,000
	Zon. Wind	m/s	10	-150	150
	Mer. Wind	m/s	10	-150	150
	Humidity	%	10	0	1,000
	Air Temp.	°C	100	0	4,000
	Sea Temp.	°C	100	0	4,000
Solar	Time Delta	minutes	1	0	13,1071
	GHI	W/m ²	1	0	1,020
	DNI	W/m ²	1	0	970
	DHI	W/m ²	1	0	800
Hail	Time Delta	minutes	1	0	13,1071
	Lat	coord. degrees	100	2,500	5,000
	Long	coord. degrees	100	-12,500	6,700
	Size	1/100 inch	1	0	700
Tornado	Time Delta	minutes	1	0	13,1071
	Lat	coord. degrees	100	2,400	5,000
	Long	coord. degrees	100	-12,500	-6,800
Wind	Time Delta	minutes	1	0	13,1071
	Lat	coord. degrees	100	0	5,000
	Long	coord. degrees	100	-12,500	0
	Speed	mph	10	0	2,400

TABLE 2.14: Data types overview.

Chapter 3

Algorithms

In this chapter we present the coding algorithms that we have implemented and evaluated. In Section 3.1 we give an overview of the various algorithms, including their parameters and masking/non-masking variants. In Section 3.2 we describe the general encoding scheme used for every evaluated algorithm, while in Section 3.3 we propose a general strategy to compress gaps, 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 each of the eight implemented algorithms, showing their coding and decoding routines, and describing an example that shows the encoding process step by step. In Section 3.13 we present additional implementation details.

3.1 Introduction

TODO: REWRITE. El texto acá es palabra por palabra idéntico al de la introducción. Eso debería evitarse.

The state-of-the-art algorithms used 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. However, this is sometimes not true for real-world datasets. For example, all the datasets presented in Chapter 2 consist of signals that miss either one or both of these characteristics. We propose a number of variants of state-of-the-art algorithms, which are able to encode this type of data.

We focus on algorithms that support near-lossless compression. Near-lossless compression guarantees a bounded per-sample absolute error between the decompressed and the original signals. The error threshold can be specified by the user via a parameter, denoted ϵ . When ϵ is equal to 0, the compression is lossless, i.e., the decompressed and the original signals are identical.

The algorithms follow a model-based compression approach that compresses signals by exploiting correlation among signal samples taken at close times (*temporal correlation*) and, in some cases, among samples from various signals (*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 yield poor compression results [27].

For most algorithms we propose 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. On the other hand, variant NM encodes the gaps and the data values together. Implementation details are presented in the remaining sections of this chapter. We point out that the gaps in a decoded file match the gaps in the original file exactly, regardless of the value of the error threshold parameter (ϵ). 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 (described in Section 3.4), all the algorithms depend on a window size parameter, denoted w , which defines the size of the blocks into which the data are partitioned for encoding. In one of the algorithms, PCA (presented in Section 3.5), parameter w defines a *fixed block size*, while in the rest of the algorithms it defines a *maximum block size*. More details are presented with the specific description of each algorithm, later in this chapter.

In Table 3.1 we outline some characteristics of the evaluated algorithms and the proposed 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 coding algorithms. For each algorithm, the table shows whether it supports lossless and near-lossless compression (second and third columns, respectively), its compression model (fourth column), whether the masking (M) and non-masking (NM) variants apply (fifth and sixth columns, respectively), and whether the algorithm depends on a window size parameter (w) (last column).

3.2 General Encoding Scheme

Figure 3.1 shows a general encoding scheme used for most of the evaluated algorithms. The decoding scheme is symmetric. Constant and linear model algorithms only exploit the temporal correlation in the data, thus they iterate through the data columns and encode each independently. Since correlation models also exploit the spatial correlation (i.e. the data columns are *not* encoded independently), algorithm GAMPS follows a different scheme, which we present in Section 3.12.

In Figure 3.1, the inputs for the coding routine are a CSV data file in the format presented in Chapter 2, 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.

```

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  $in$ 
1 Create output file  $out$ 
2 Encode an algorithm identification key, and parameter  $w$  (if applies)
3 Encode the header of the input file
4 Encode the timestamp column using a lossless code
5 if  $v == M$  then
6   | Encode gap locations in each signal column of the input file (independently) into  $out$ 
7 end
8 if we are using a constant or linear model algorithm then
9   | Encode each signal column of the input file (independently) into  $out$ , using a coding
   | routine for a specific algorithm (i.e. Base, PCA, APCA, PWLH, PWLHInt, CA,
   | SF, FR)
10 else if we are using a correlation model algorithm then
11   | Group the signal columns of the input file into disjoint subsets, then encode each
   | subset independently into  $out$ , using a coding routine for a specific algorithm (i.e.
   | GAMPS)
12 end
13 return  $out$ 

```

FIGURE 3.1: General encoding scheme for every evaluated algorithm.

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). This is done using a lossless code in which every integer is encoded independently, using a fixed number of bits, B_c , which we define next.

Definition 3.2.1. The number of bits B_c required to encode a specific value of data type z of a certain dataset d is given by

$$B_c(z, d) = \lceil \log_2 (\max(z, d) - \min(z, d) + N_c(z, d)) \rceil, \quad (3.1)$$

where $\max(z, d)$ and $\min(z, d)$ are the maximum and minimum values allowed, respectively, for the data type, and $N_c(z, d)$ is 1 if the data type admits gaps, and 0 otherwise.

Recall, from Section 2.1, that the maximum and minimum values allowed for each data type are specified in the header of the dataset CSV file, making this information known to the encoder. Also, recall 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 data type, and 1 for the rest of the data types in each dataset. Notice that N_c is a constant that accounts for an extra symbol needed to encode a gap.

We focus on the compression of the sample columns (i.e. the rest of the columns in the data file), and do not delve into the optimization of timestamp compression, which we leave for future work. When the masking variant of the 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, $x_1 \dots x_n$, 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, $p(\cdot|x_1 \dots x_{i-1})$, for the symbol in position i given the past symbols $x_1 \dots x_{i-1}$, $1 \leq i \leq n$, an AC generates a lossless encoding bit stream for $x_1 \dots x_n$, of length $-\log P(x_1 \dots x_n) + O(1)$, where $P(x_1 \dots x_n) = \prod_{i=1}^n p(x_i|x_1 \dots x_{i-1})$. This code length is optimal for this probability assignment, up to an additive constant [46].

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.

TODO: explicar en qué sentido, y dar referencia.

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 , $1 \leq i \leq n$

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

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

A first-order Markov process 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 process. A KT probability assignment for a first-order Markov process 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.

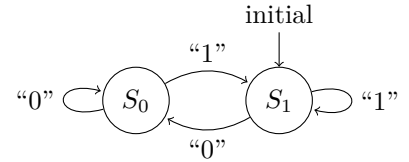


FIGURE 3.2: Markov process diagram.

Analyzing the datasets presented in Chapter 2, we notice that the positions of the data gaps may follow different patterns for different datasets, but, in general, the gaps occur in bursts and the number of gaps is smaller than the number of data values. With this in mind, we model the sequence of gaps as a first-order Markov process, and we use the KT probability assignment for the symbols that occur in each state.

TODO: explicar por qué tiene sentido usar un modelo de Markov—puede capturar “burstiness”. No queda claro por qué se menciona la otra condición (hay meos gaps que data). No parece relevante al desempeño de compresión.

3.4 Algorithm Base

Algorithm *Base* is a trivial lossless coding algorithm that serves as a base ground for comparing the performance of the rest of the algorithms. It 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
1 foreach entry in column, entry, do
2   if entry == “N” then
3     value =  $NO\_DATA$ 
4   else
5     value = entry
6   end
7   Encode value using  $B_c$  bits
8 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. It has a window size parameter, w , as defined in Section 3.1, used as described below. 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). A window 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 the window 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 window values is encoded separately (lines 9-12). Notice that, in the former case, the absolute error between the encoded and the original values in w 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 $x \in \mathbb{R}$, such that $m \leq x \leq M$. Then $|x - mid_range| \leq \epsilon$.

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

If $x' \geq mid_range$, then $|x' - mid_range| > \epsilon \implies x' - mid_range > \epsilon$.

If $m + M$ is even, then $x' - mid_range > \epsilon \implies x' - (m + M)/2 > \epsilon \implies 2x' - m - M > 2\epsilon$.

Since $M \geq x'$, then $2x' - m - M > 2\epsilon \implies 2M - m - M = M - m > 2\epsilon$, which contradicts one of the hypothesis.

If $m + M$ is odd, then...

If $x' < mid_range$, the proof is ...

To prove it by contradiction try and assume that the statement is false, proceed from there and at some point you will arrive to a contradiction. \square

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 (Line 4 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 fixed number of bits is read, from which a value is decoded and written to the decoded CSV data file (lines 9-12).

```

input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm PCA
         $\epsilon$ : 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 variant *M* of algorithm PCA.

```

input : in: binary file coded with algorithm PCA
        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 += size
15 end

```

FIGURE 3.5: Decoding routine for variant *M* of algorithm PCA.

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.

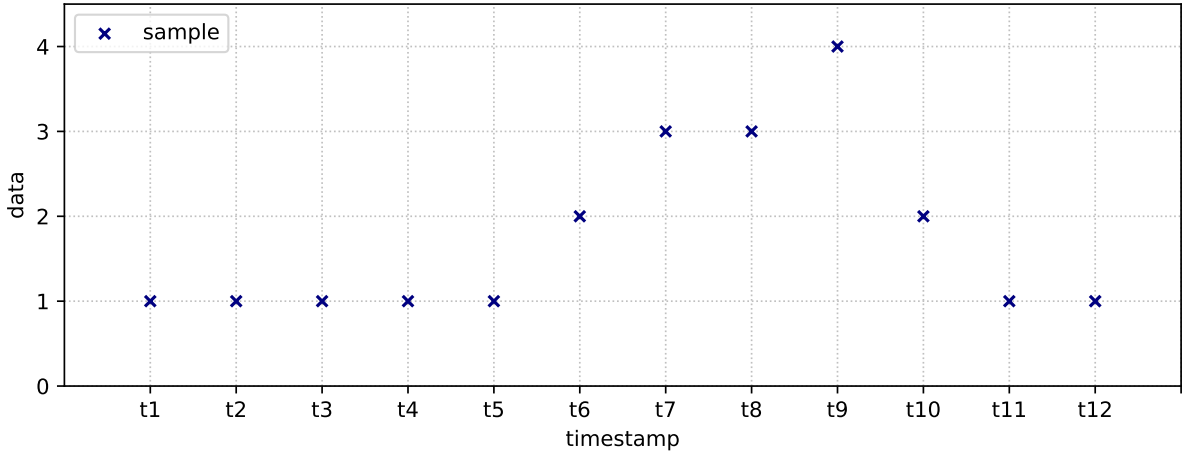


FIGURE 3.6: Example: Signal consisting of 12 samples.

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 .

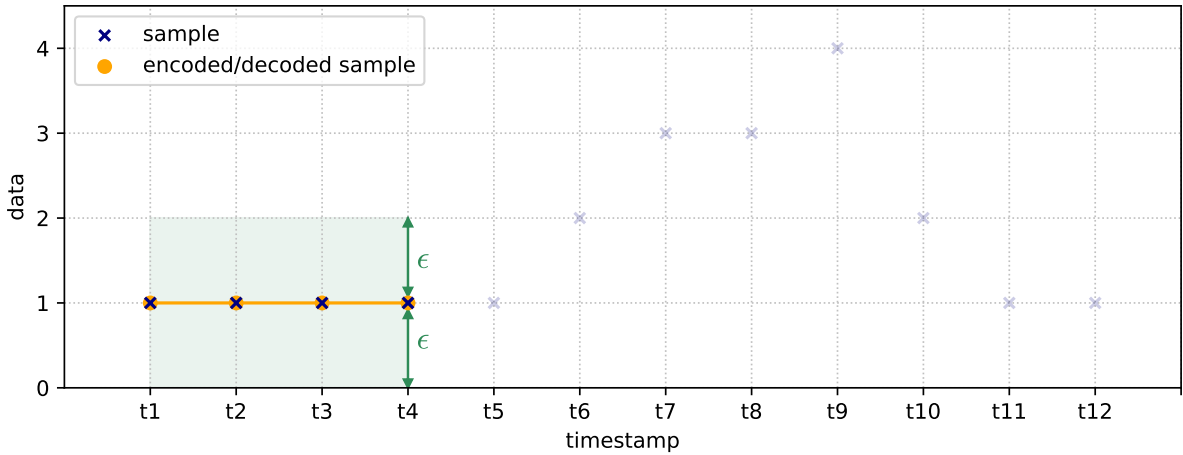


FIGURE 3.7: Example: Algorithm PCA with $\epsilon = 1$ and $w = 4$. Step 1.

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.

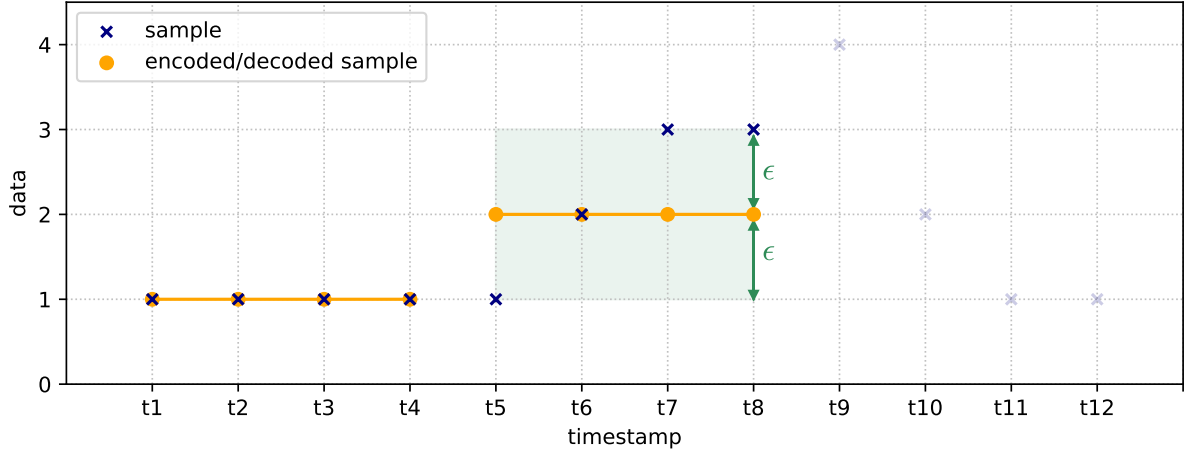


FIGURE 3.8: Example: Algorithm PCA with $\epsilon = 1$ and $w = 4$. Step 2.

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.

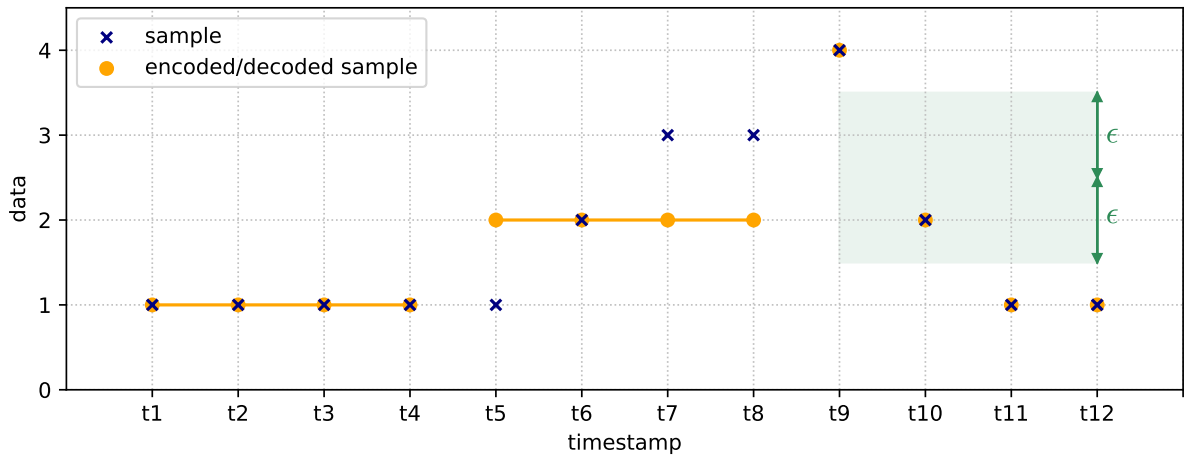


FIGURE 3.9: Example: Algorithm PCA with $\epsilon = 1$ and $w = 4$. Step 3.

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, which is $1 + C$, where C is B_c bits. On the other hand, since the threshold condition does not hold for the last window, it is encoded with $1 + w * C$ bits. Thus, windows consisting of adjacent samples are encoded using less bits. This example illustrates why algorithm PCA is expected to achieve better compression performances on slowly varying signals rather than rough signals.

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 PCA
         $\epsilon$ : 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, entry, do
15        if entry == “N” then
16          value = NO_DATA
17        else
18          value = entry
19        end
20        Encode value using  $B_c$  bits
21      end
22    end
23  end
24 end

```

FIGURE 3.10: Coding routine for variant *NM* of algorithm PCA.

A window can be encoded in three different ways. If every entry 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 the window 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 the window is encoded (lines 9-11). In every other case, bit 1 is output, and each of the window entries 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., $1 + C$, where C is B_c bits, while in the last case the window is encoded with $1 + w * C$ 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 operates similarly to algorithm PCA, the difference being that in APCA the size of the blocks in which the data are separately encoded is not fixed, but variable. In this case, the window size parameter (w) establishes a maximum block size for the algorithm. 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 window violate the error threshold constraint (i.e. the difference between its maximum and minimum values is 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.

```

input : column: column of the CSV data file to be encoded
         out: binary file encoded with algorithm APCA
          $\epsilon$ : maximum error threshold
          $w$ : maximum window size
1 Create an empty window, win
2 foreach entry in column, entry, do
3   Let  $m$  and  $M$  be the minimum and maximum sample values in  $win \cup entry$ , 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   Add entry 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 variant M of algorithm APCA.

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. $\lceil \log_2 w \rceil + C$, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and C bits, where C is B_c bits, are used for encoding its mid-range.

```

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

```

FIGURE 3.12: Auxiliary routine EncodeWindow for algorithm APCA.

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 coded with algorithm APCA
          $out$ : 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    $n += size$ 
7 end

```

FIGURE 3.13: Decoding routine for variant M of algorithm APCA.

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.

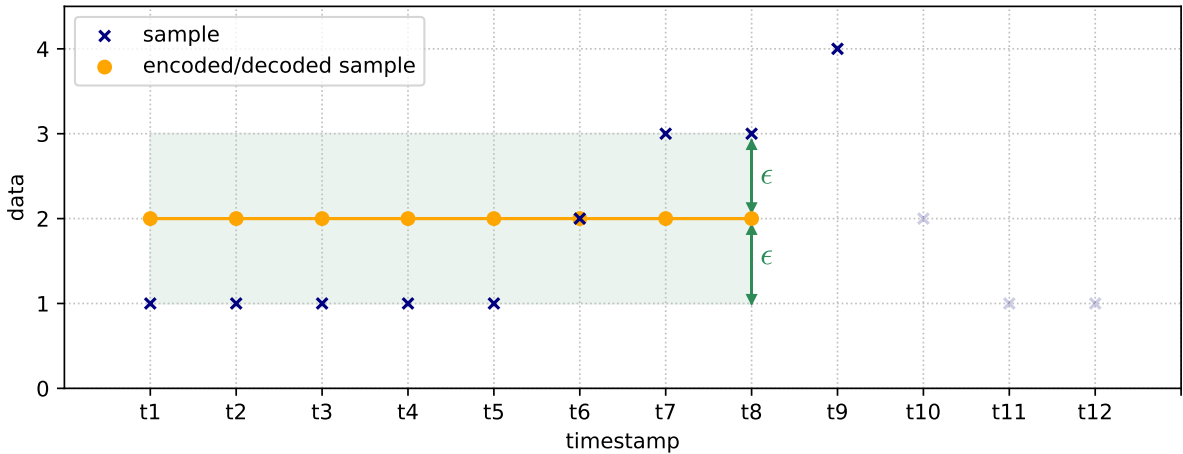


FIGURE 3.14: Example: Algorithm APCA with $\epsilon = 1$ and $w = 256$. Step 1.

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.

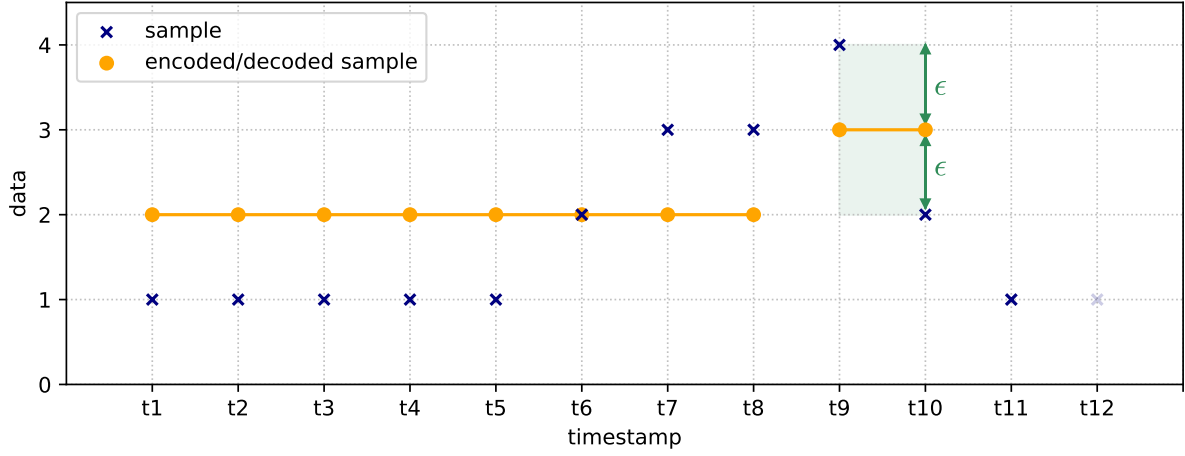


FIGURE 3.15: Example: Algorithm APCA with $\epsilon = 1$ and $w = 256$. Step 2.

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.

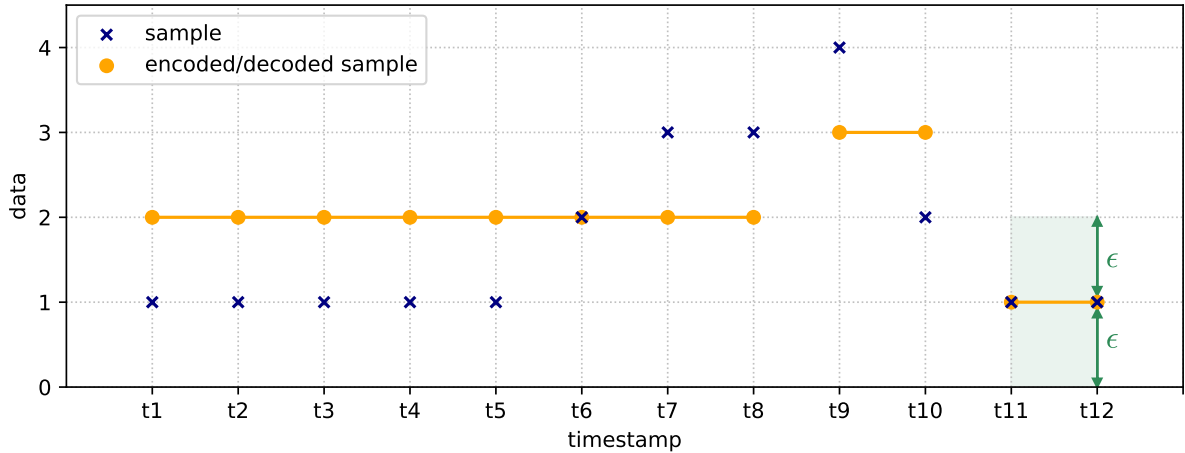


FIGURE 3.16: Example: Algorithm APCA with $\epsilon = 1$ and $w = 256$. Step 3.

We point out that each of the three windows in this example are encoded using exactly the same number of bits. 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 slowly varying signals rather than rough signals.

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. $\lceil \log_2 w \rceil + C$, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and C bits, where C is 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, these 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-axis and y-axis corresponding to timestamps and sample values, respectively.

A linear model encoding algorithm operates by successively encoding endpoints of segments, which span samples whose 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. In Figure 3.17 we present the auxiliary routine FillSegment, which performs this interpolation; it is used by the decoding routine of all linear model algorithms. 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's endpoints (Line 1). The output is a list consisting of the 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 in  $t\_col$ ,  $t_i$ , 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    Add  $s_i$  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, 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 $[t_1, \dots, t_N]$, 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, 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.

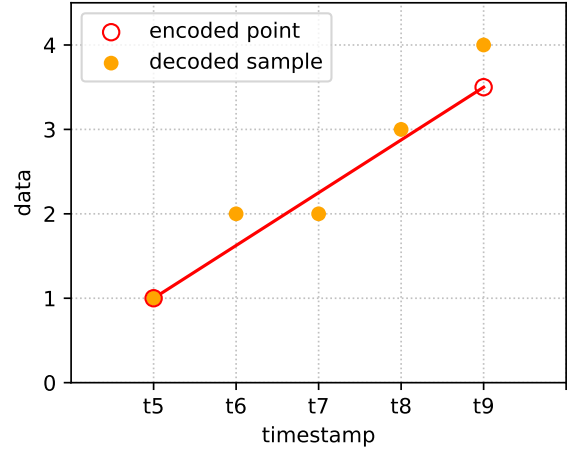


FIGURE 3.18: Example of the auxiliary routine FillSegment for linear model algorithms.

3.8 Algorithms PWLH and PWLHInt

Algorithm *PieceWise Linear Histogram (PWLH)* [36] is a linear model algorithm that supports lossless and near-lossless compression. It has a window size parameter (w) that establishes a maximum block size in which the data are separately encoded. 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 algorithms, except for the specific differences that are pointed out in Subsection 3.8.2.

Since PWLH is a linear model algorithm, 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), parsing them into consecutive windows of variable size (up to a maximum size w), such that all the points representing entries in a certain window satisfy the *valid hull condition*, defined in the next paragraph. The samples in each window are encoded via the auxiliary routine EncodeWindow, shown in Figure 3.20. This method computes the segment that minimizes the *mean square error (MSE)* for all the points representing the entries in the window, then encodes said segment through its two endpoints, using 32 bits for encoding their y-coordinates as floats (recall that the coding algorithm encodes the timestamp column, with the respective x-coordinates, first). The valid hull condition satisfied by all the points representing entries in the window, guarantees that the decoded samples satisfy the per-sample absolute error constraint established by parameter ϵ .


```

input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm PWLH
         $\epsilon$ : maximum error threshold
        w: maximum window size
        t_col: timestamp column
1  Create an empty window, win
2  foreach entry in column, entry, do
3      Obtain timestamp for entry,  $t_{entry}$ , from t_col
4      Let E be the point with coordinates ( $t_{entry}$ , entry)
5      if entry is not the first entry in column then
6          Let S be the set of points representing the entries in win, plus E, and let H be
            the convex hull of S
7          Let valid_hull be true iff H satisfies the valid hull condition, defined in 3.8.1, for  $\epsilon$ 
8          if not valid_hull or  $|win| == w$  then
9              EncodeWindow(win, out, w, t_col) // routine shown in Figure 3.20
10             Set win to an empty window
11         end
12     end
13     Add entry to win
14 end
15 if win is not empty then
16     EncodeLastWindow(win, out, w, t_col) // routine shown in Figure 3.23
17 end

```

FIGURE 3.19: Coding routine for variant *M* of algorithm PWLH.

```

input : win: window to encode
        out: binary file encoded with algorithm PWLH
        w: maximum window size
        t_col: timestamp column
1  Encode  $|win|$  using  $\lceil \log_2 w \rceil$  bits
2  Let segment be the line segment that minimizes the MSE for the points representing the
    entries in win
3  Let  $s_o$  and  $s_f$  be the y-coordinates of the endpoints of segment
4  Encode  $s_o$  as a float, using 32 bits
5  Encode  $s_f$  as a float, using 32 bits

```

FIGURE 3.20: Auxiliary routine EncodeWindow for algorithm PWLH.

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 distance from any of the points in *H* to said edge line is less than or equal to 2ϵ .

In figures 3.21 and 3.22 we present an example that illustrates how the valid hull condition is checked in the coding routine. In Figure 3.21, set *S*, which is defined in Line 6, contains the points representing the three entries in the current window, *win*, plus the incoming point, *E*₄. 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 8 evaluates to false (we assume that $|win| < w$), and the current column entry is added to *win* (Line 13).

In the next step, presented in Figure 3.22, set *S* contains the points representing the four entries in *win*, plus *E*₅. In this case, convex hull *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 8 is satisfied, so win is encoded (Line 9), emptied (Line 10) and added the current column entry (Line 13).

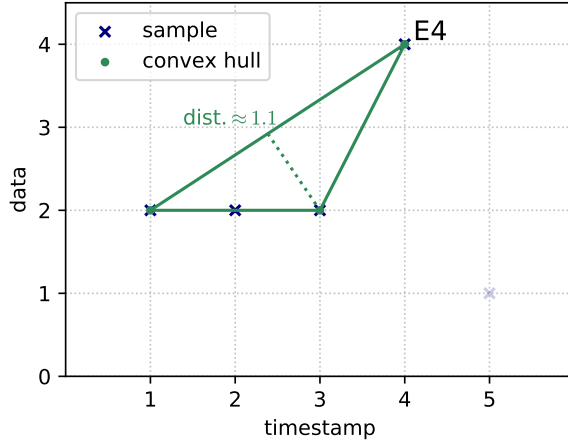


FIGURE 3.21: Example: checking the valid hull condition. Algorithm PWLH with $\epsilon = 1$. Step 1.

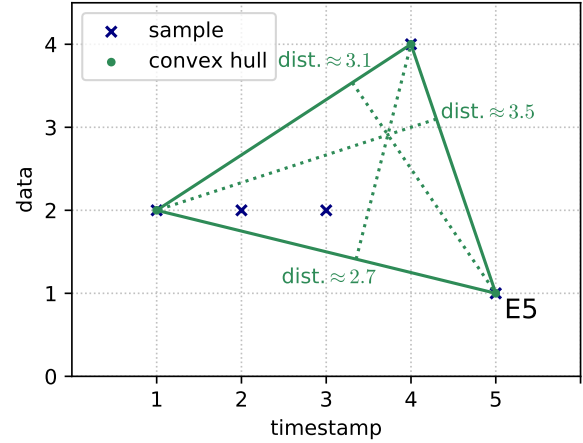


FIGURE 3.22: Example: checking the valid hull condition. Algorithm PWLH with $\epsilon = 1$. Step 2.

We point out that calculating the segment that minimizes the MSE for a set of points is computational 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 an entry should be added to the window, while the segment that minimizes the MSE is only computed for the points representing the entries in a complete window.

If win is not empty after executing the last iteration of the loop in the coding routine, it is encoded in Line 16, via the auxiliary routine `EncodeLastWindow`, shown in Figure 3.23. If win consists of a single sample, its size is encoded using $\lceil \log_2 w \rceil$ bits, and the sample value is encoded using B_c bits. On the other hand, if win consists of multiple samples, it is encoded in the same way that the previous windows are, i.e. via the auxiliary routine `EncodeWindow`, shown in Figure 3.20.

```

input :  $win$ : window to encode
          $out$ : binary file encoded with algorithm PWLH
          $w$ : maximum window size
          $t\_col$ : timestamp column
1 if  $|win| == 1$  then
2   | Encode 1 using  $\lceil \log_2 w \rceil$  bits
3   | Let  $value$  be the single sample in  $win$ 
4   | Encode  $value$  using  $B_c$  bits
5 else
6   | EncodeWindow( $win, out, w, t\_col$ ) // routine shown in Figure 3.20
7 end

```

FIGURE 3.23: Auxiliary routine `EncodeLastWindow` for algorithm PWLH.

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 representing the entries 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's 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). On the other hand, if the window size is equal to 1, a column-specific fixed number of bits is read, from which a value is decoded and written to the decoded file (lines 13-14).

```

input : in: binary file coded with algorithm PWLH
        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 variant M of algorithm PWLH.

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 distance between adjacent 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 6 of the coding routine, violates the valid hull condition in Line 7. The first iteration is the only one in which the condition in Line 5 evaluates to false, so the algorithm just adds the first sample to the window (Line 13). Figure 3.25 shows this step in the graph. Observe that, besides the sample values, the convex hull for the current window is also shown.

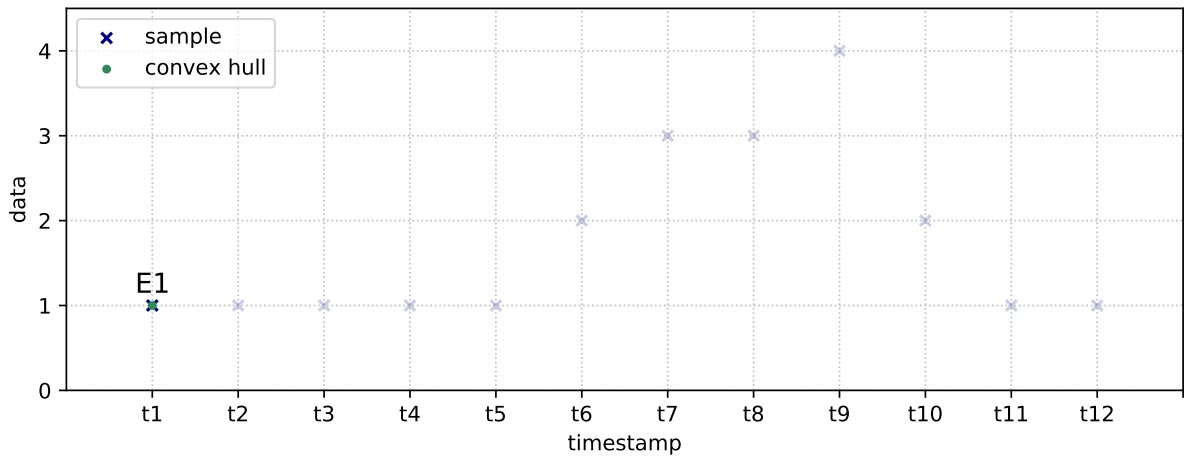


FIGURE 3.25: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 1.

The boundary of the convex hull that includes the second 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 the second sample is added to the window. This step is shown in Figure 3.26.

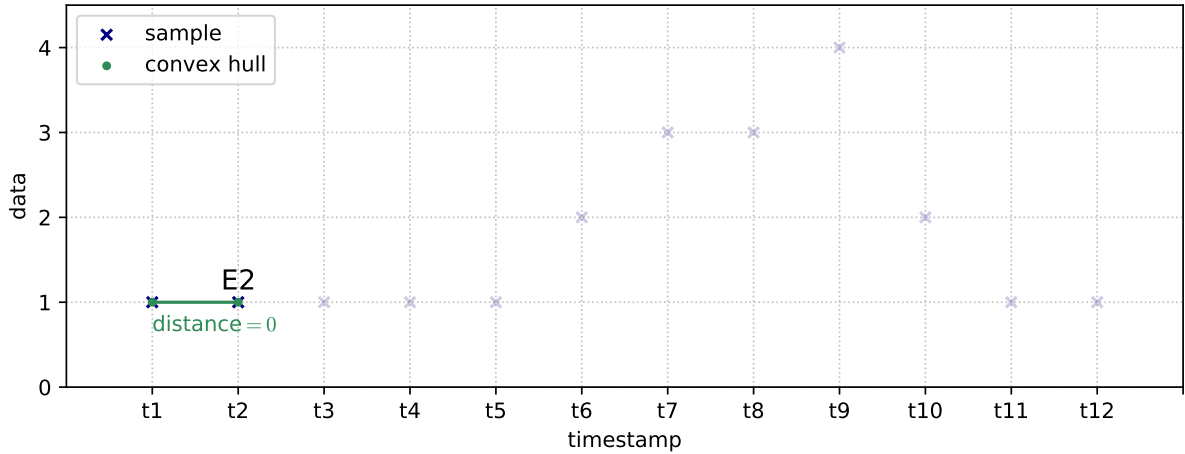


FIGURE 3.26: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 2.

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

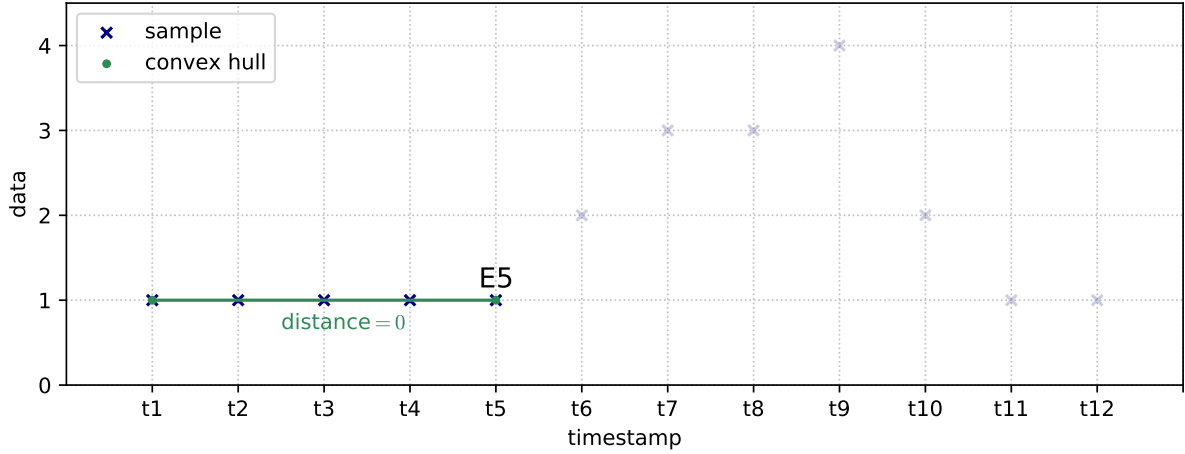


FIGURE 3.27: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 3.

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 the sample is added to the window.

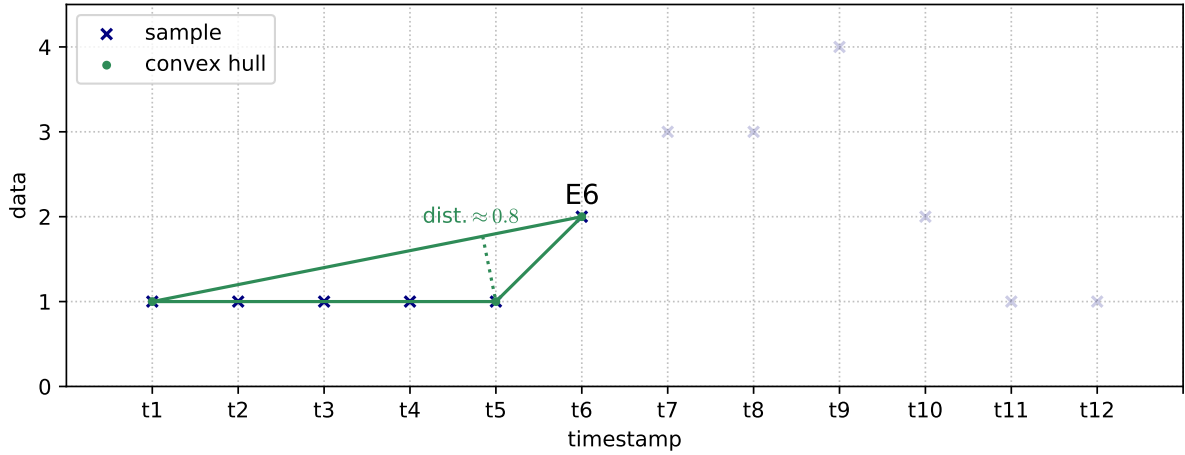


FIGURE 3.28: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 4.

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 samples are added to the window. These steps are shown in figures 3.29, 3.30 and 3.31.

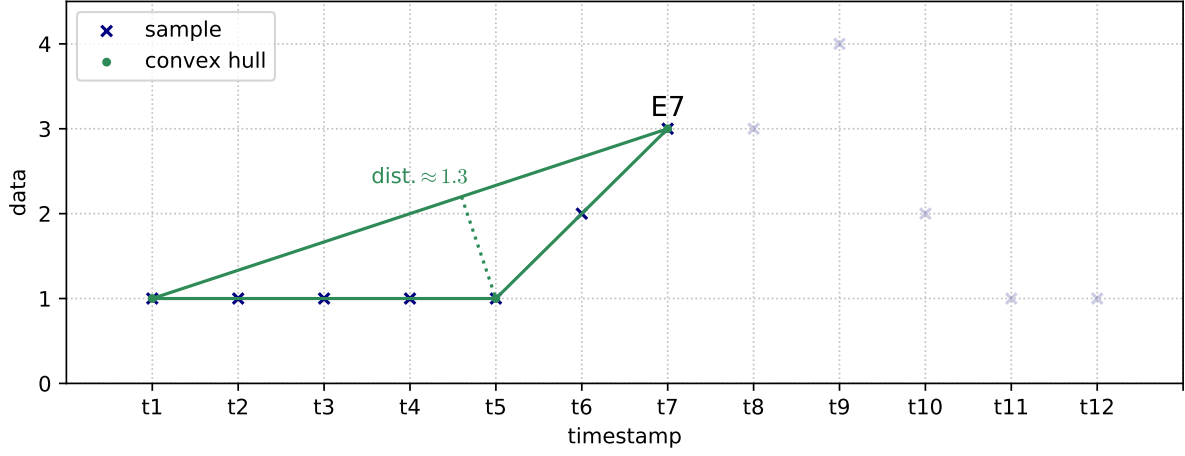


FIGURE 3.29: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 5.

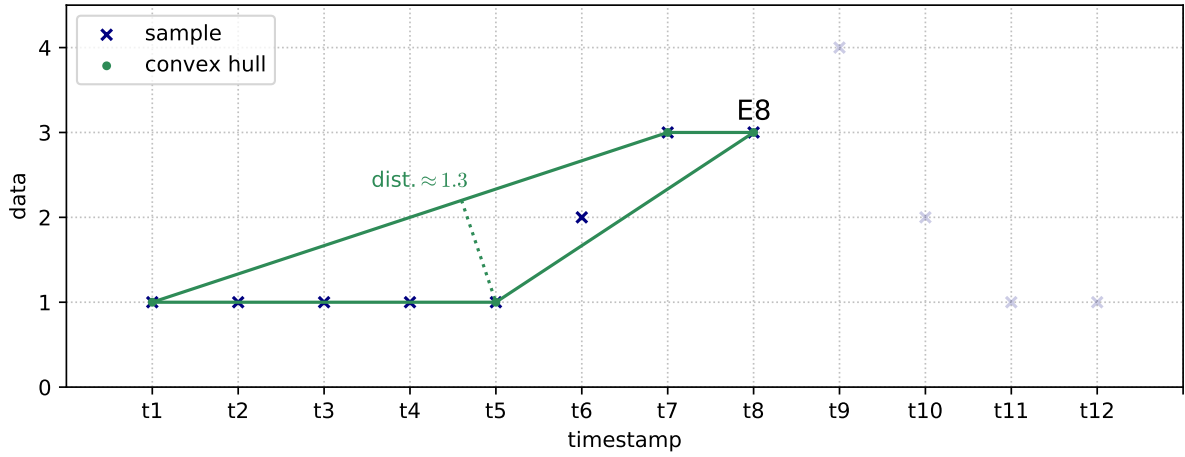


FIGURE 3.30: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 6.

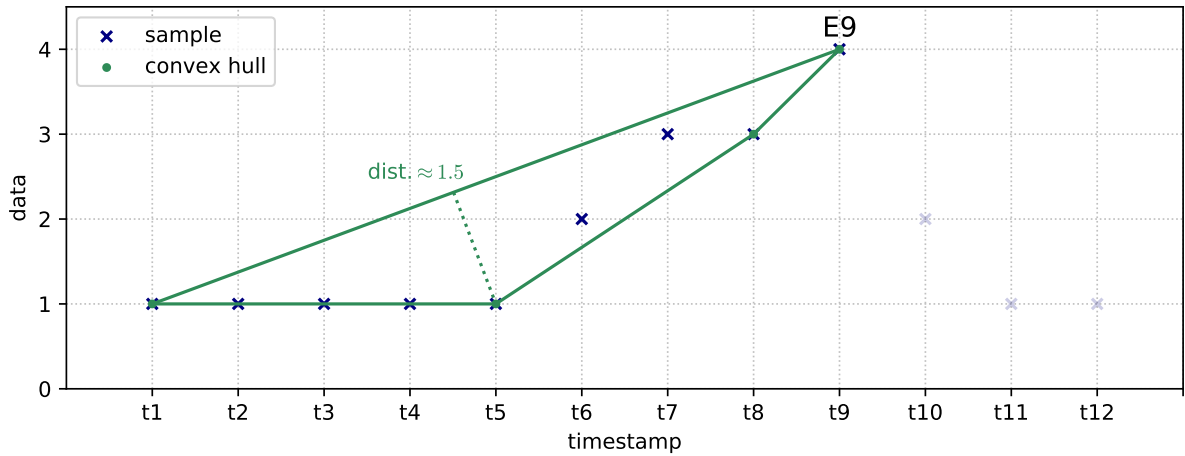


FIGURE 3.31: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 7.

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 7 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 .

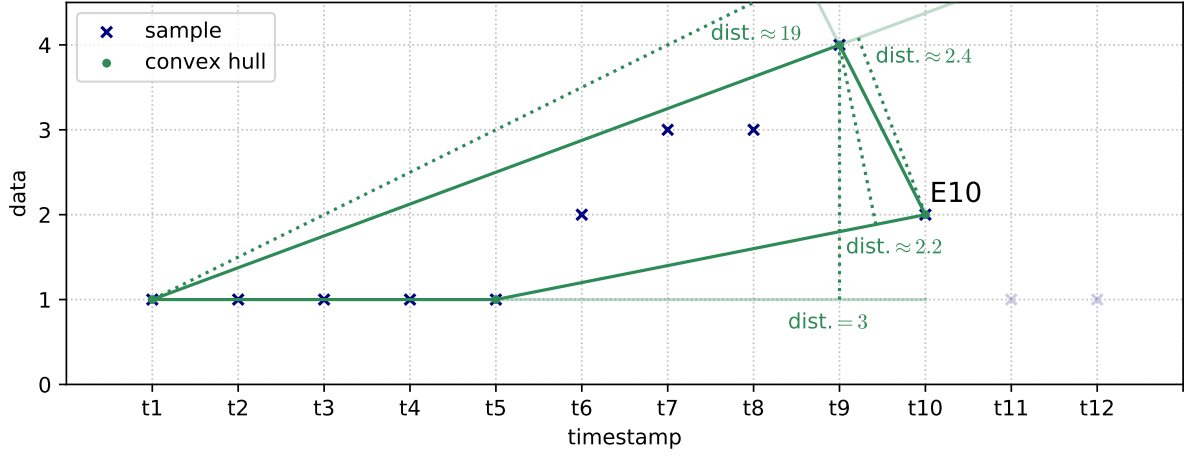


FIGURE 3.32: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 8.

Since the condition in Line 8 becomes true, the window is encoded via the auxiliary routine *EncodeWindow* (Line 9), and the window is emptied and added sample value 2 (lines 10 and 13). Routine *EncodeWindow* models the points representing entries in the window through the line segment that minimizes the MSE for said points (recall Figure 3.20). This segment and its two 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's endpoints.

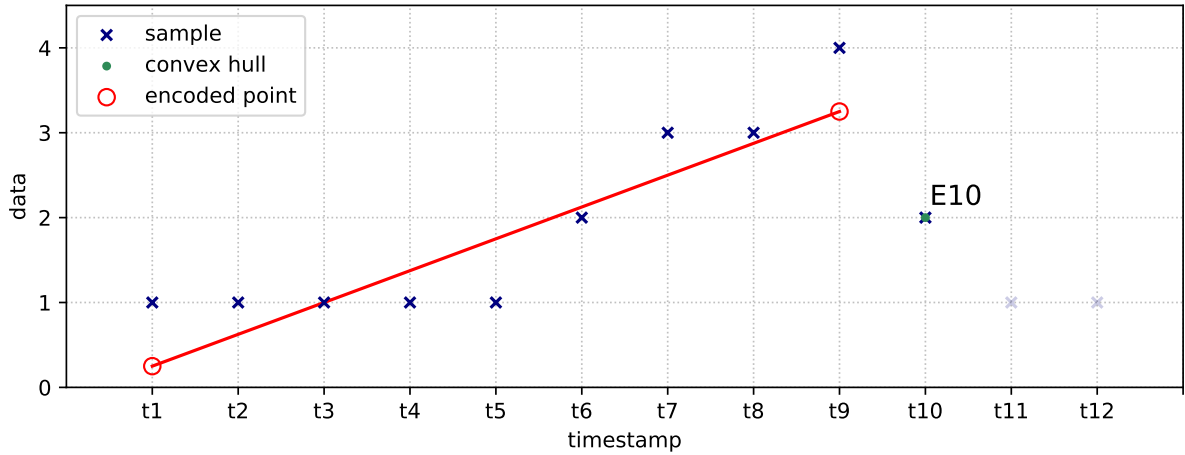


FIGURE 3.33: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 9.

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 samples, which are encoded via the auxiliary routine `EncodeLastWindow` (Line 16). The associated segment and its two endpoints are 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.

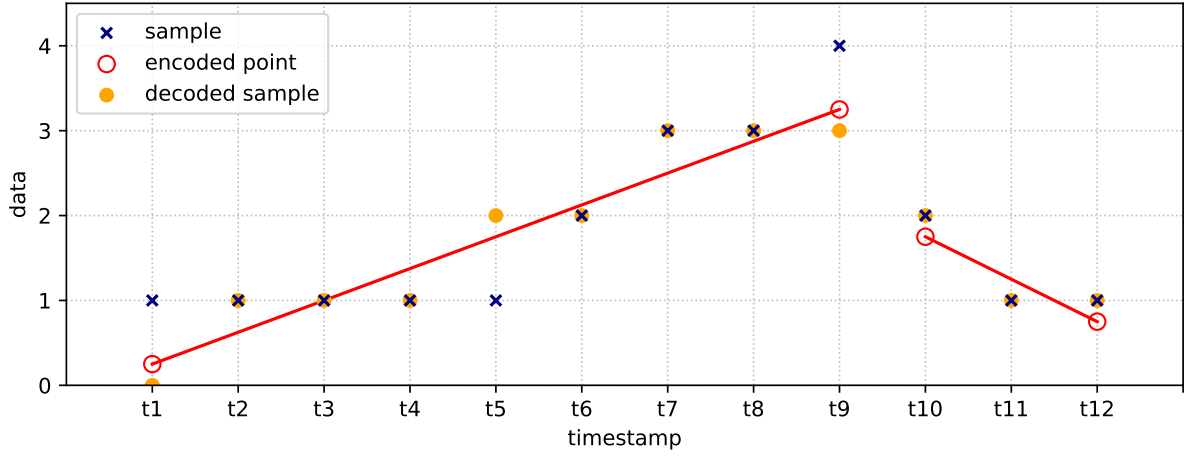


FIGURE 3.34: Example: Algorithm PWLH with $\epsilon = 1$ and $w = 256$. Step 10.

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 encodes the y-coordinates of each line segment as floats, using 32 bits (lines 4-5 in Figure 3.20). Since these y-coordinates are data samples, the compression performance of algorithm PWLH can be improved if they are instead encoded as bounded integers, using a fixed number of bits that depends on the data type and the dataset. 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 4-5 in the auxiliary routine `EncodeWindow`, shown in Figure 3.20, so that both y-coordinates are rounded to the nearest integer, and then encoded using a column-specific fixed number of 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 8 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 type being encoded. Otherwise, the fixed number of bits used for encoding the coordinates may not be enough. 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 .

In Section 4.4, in which we evaluate the coding algorithms, the experimental results suggest that the compression performance of algorithm PWLHInt is superior to that of algorithm PWLH.

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 values or gaps, but it cannot include both. Therefore, a new constraint is added to the condition in Line 8 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 entries in the window are sample values, 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. It has a window size parameter (w) that establishes a maximum block size for the algorithm. CA supports 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 followed by a window of variable size (up to a maximum size w), such that all the points representing entries in the same window lie within vertical distance ϵ from the segment whose endpoints correspond to the single sample and the last entry of the window. The routine operates by successively encoding a single sample (using an auxiliary routine `EncodeWinStart`, shown in Figure 3.36), then growing a window of subsequent samples, adding one entry at a time until it is complete, and finally encoding the size and the last sample of the window (using an auxiliary routine `EncodeWinEnd`, shown in Figure 3.37).

```

input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm CA
         $\epsilon$ : maximum error threshold
         $w$ : maximum window size
         $t\_col$ : timestamp column

1 Create an empty window, win
2 foreach entry in column, entry, do
3   Obtain timestamp for entry,  $t_{entry}$ , from  $t\_col$ 
4   Let  $E$  be the point with coordinates  $(t_{entry}, entry)$ 
5   if entry is the first entry in column then
6     | Make  $A = E$ , then EncodeWinStart(entry, out,  $w$ ) // routine in Figure 3.36
7   else if win is empty then
8     | Make  $S = E$ , then add entry to win
9     | 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
10  else
11    | Let  $(A, E)$  be the segment with initial point  $A$  that passes through point  $E$ 
12    | if not  $\text{slope}(SMin) \leq \text{slope}((A, E)) \leq \text{slope}(SMax)$  or  $|win| == w$  then
13      |   EncodeWinEnd(win, out,  $w$ ) // routine shown in Figure 3.37
14      |   Set win to an empty window
15      |   Make  $A = E$ , then EncodeWinStart(entry, out,  $w$ ) // routine in Figure 3.36
16    | else
17      |   Make  $S = E$ , then add entry to win
18      |   Let  $SMinOld$  and  $SMaxOld$  be rays equal to  $SMin$  and  $SMax$ , respectively
19      |   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
20      |   Let  $SMin = \max\{SMinOld, SMin\}$  and  $SMax = \min\{SMaxOld, SMax\}$ 
21    | end
22  end
23 end
24 if win is not empty then
25   | EncodeWinEnd(win, out,  $w$ ) // routine shown in Figure 3.37
26 end

```

FIGURE 3.35: Coding routine for variant M of algorithm CA.

```

input : value: value to encode
          out: binary file encoded with algorithm CA
          w: maximum window size
1 Encode 1 using  $\lceil \log_2 w \rceil$  bits
2 Encode value using  $B_c$  bits

```

FIGURE 3.36: Auxiliary routine EncodeWinStart for algorithm CA.

```

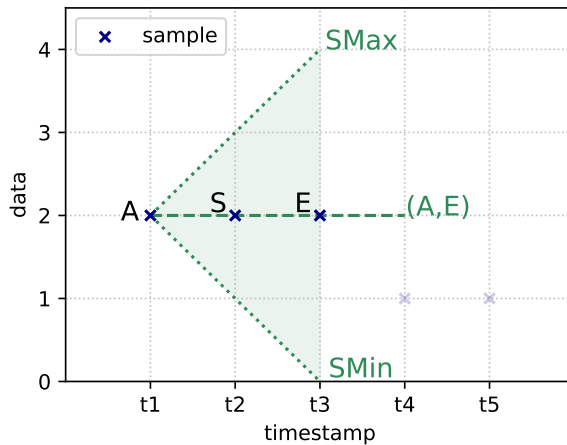
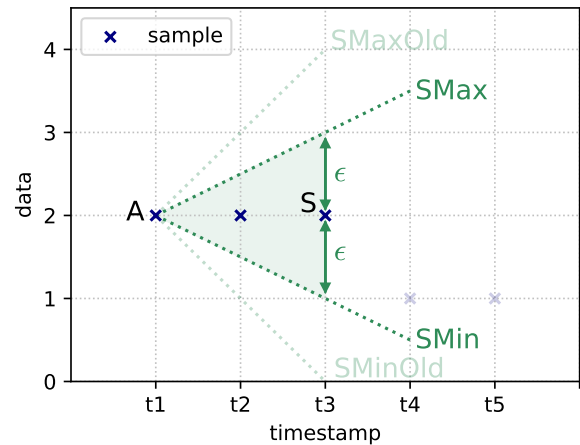
input : win: window to encode
          out: binary file encoded with algorithm CA
          w: maximum window size
1 Encode  $|win|$  using  $\lceil \log_2 w \rceil$  bits
2 Let value be the last sample in win
3 Encode value using  $B_c$  bits

```

FIGURE 3.37: Auxiliary routine EncodeWinEnd for algorithm CA.

The algorithm determines when to stop growing the window, i.e., when the current window *win* is complete, based upon three points: *archived* (*A*) is the point corresponding to a single sample that precedes a window; *snapshot* (*S*) is the point corresponding to the entry most recently added to *win*; *incoming* (*E*) is the point associated to the column entry for the current iteration. The algorithm maintains a cone, determined by two rays departing from *A*, *SMin* and *SMax*, such that the current column entry is added to *win* as long as the point *E* lies within the cone. This cone, in turn, is defined so that if the current column entry is indeed added to *win*, all the points representing the entries in *win* are at a vertical distance at most ϵ from the segment (*A*, *E*).

In figures 3.38 and 3.39 we present an example that illustrates the key steps of the coding algorithm. In Figure 3.38 the three points, *A*, *S*, and *E*, as well as the two rays departing from *A*, *SMin* and *SMax*, are shown. *A* is defined in the first iteration (Line 6), while *S* and the two rays are defined in the second iteration (lines 8-9). In the third iteration, *E* lies within the cone determined by the two rays, so the condition in Line 12 evaluates to false (we assume that $|win| < w$), i.e. *win* is not complete, and lines 17-20 are executed: *S* is made equal to *E*, the current column entry is added to *win* (Line 17), and *SMin* and *SMax* are updated (lines 18-20). The updated information is shown in Figure 3.39.

FIGURE 3.38: Example: key steps of the coding algorithm. Algorithm CA with $\epsilon = 1$. Step 1.FIGURE 3.39: Example: key steps of the coding algorithm. Algorithm CA with $\epsilon = 1$. Step 2.

In the successive iterations, the cone keeps reducing its slope, 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 12 becomes true, so the size and last sample of win are encoded (Line 13), win is emptied (Line 14), and the process starts again, by defining point A and encoding the column entry for the current iteration (Line 15).

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, two values, $size$ and $value$, are decoded from the binary file. $value$ corresponds to the y-coordinate of a point, whose x-coordinate is obtained from the timestamp column (Line 5). If $size$ is equal to 1, $value$ is the result of an encoding produced by routine `EncodeWinStart` (recall Figure 3.36), 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 7-8). On the other hand, if $size$ is greater than one, $value$ comes from an encoding produced by routine `EncodeWinEnd` (recall Figure 3.37), so it corresponds to the y-coordinate of the last snapshot point (S) whose column entry was added to the window. In this case, the points representing the entries 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 11). Routine `FillSegment` (recall Figure 3.17) 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 12-14).

```

input : in: binary file coded with algorithm CA
         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      Decode value using  $B_c$  bits
5      Obtain timestamp for value,  $t_{value}$ , from t_col
6      if size == 1 then
7          Let  $A$  be the point with coordinates  $(t_{value}, value)$ 
8          Output value to out
9      else
10         Let  $S$  be the point with coordinates  $(t_{value}, value)$ 
11         samples = FillSegment(t_col,  $A.x$ ,  $S.x$ ,  $A.y$ ,  $S.y$ ) // routine in Figure 3.17
12         foreach sample in samples, value, do
13             Output value to out
14         end
15     end
16     n += size
17 end

```

FIGURE 3.40: Decoding routine for variant M of algorithm CA.

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 distance between adjacent 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, the current column entry is added to the current window, win , as long as the incoming point (E) lies within the cone determined by the two rays departing from the archived point (A).

In the first iteration, the condition in Line 5 is satisfied, so A is made equal to $E1$, and the current sample is encoded via the auxiliary routine `EncodeWinStart`, using $\lceil \log_2 w \rceil = \log_2 256 = 8$ bits for encoding a 1, and B_c bits for encoding its value, also 1. In the second iteration, win is empty, so the condition in Line 7 is satisfied. Therefore, the snapshot point (S) is made equal to $E2$, the current sample (1) 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.

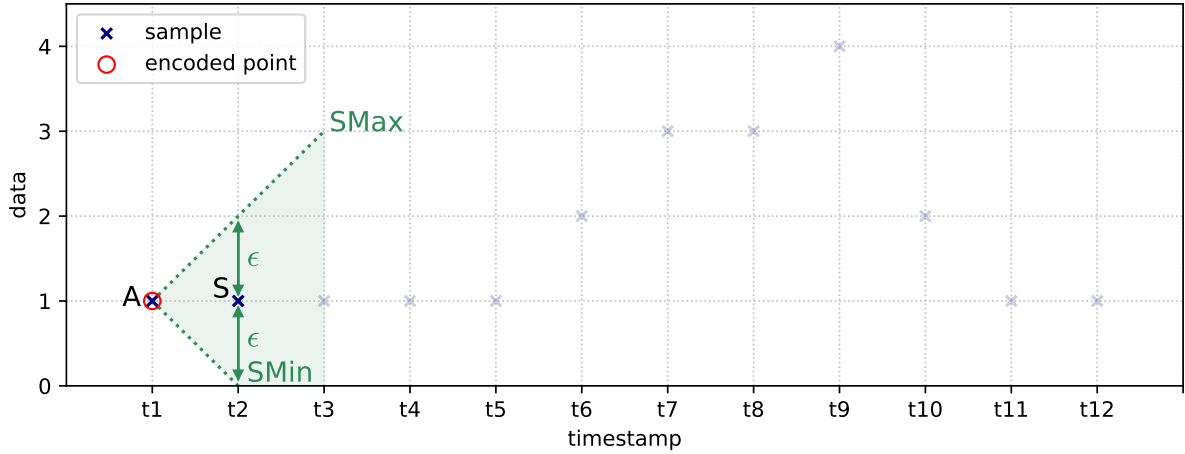


FIGURE 3.41: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 1.

In the third iteration, another sample value equal to 1 is processed. In Figure 3.42, point $E3$, as well as segment $(A, E3)$, defined in Line 11, is shown.

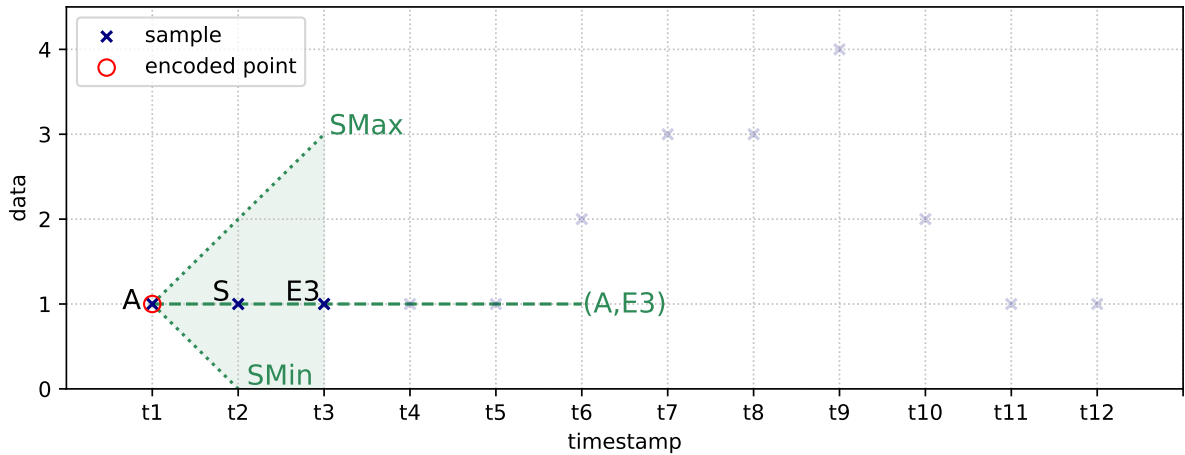


FIGURE 3.42: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 2.

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 $E3$, the current sample is added to win , and $SMin$ and $SMax$ are updated (lines 17-20). Figure 3.43 shows the information after the third iteration is completed.

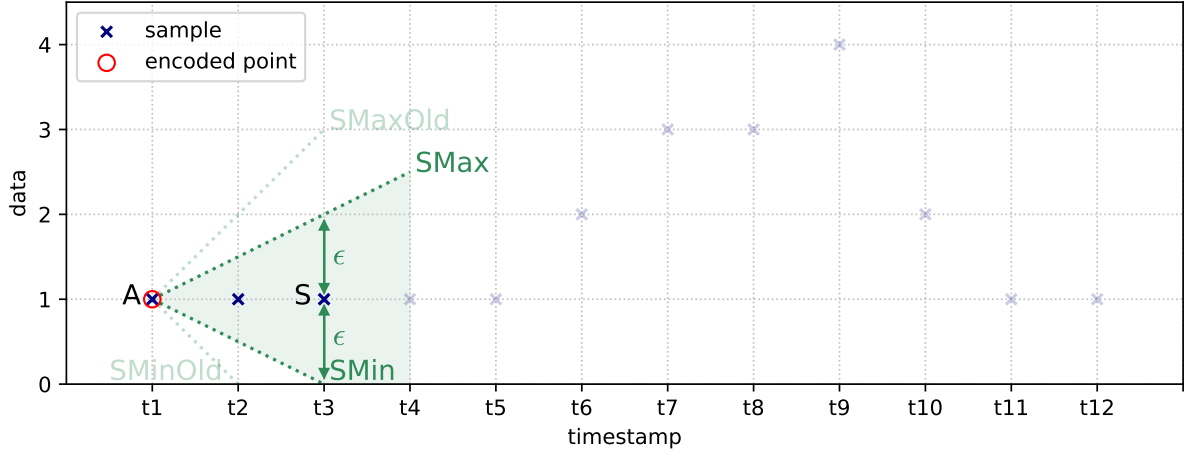


FIGURE 3.43: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 3.

The following two iterations are similar to the last one. In each iteration, the sample values are added to the window, and both rays, $SMin$ and $SMax$, are updated, reducing the slope of the cone. Figure 3.44 shows the information after the 5th iteration is completed.

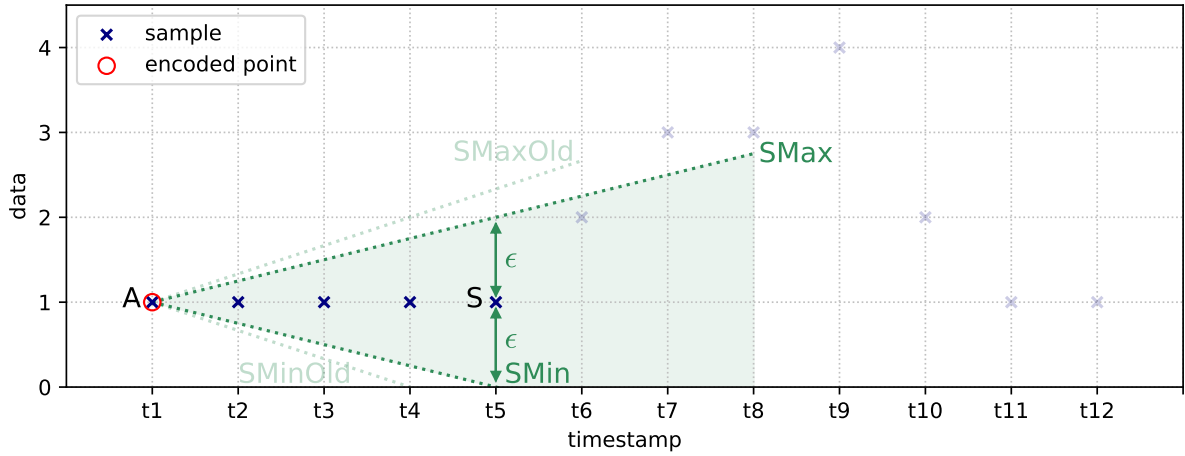


FIGURE 3.44: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 4.

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

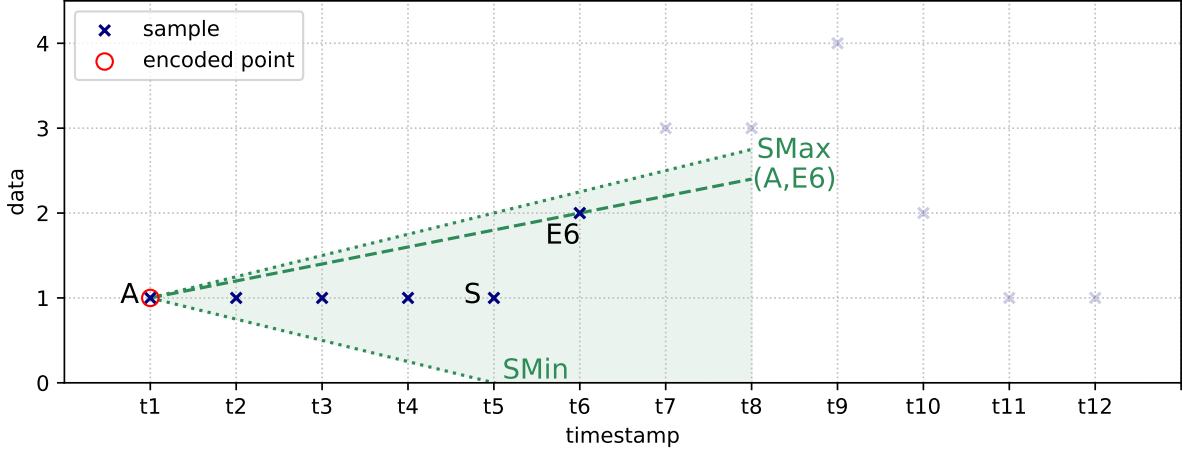


FIGURE 3.45: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 5.

$E6$ lies within the cone, so the condition in Line 12 evaluates to false. Therefore, S is made equal to $E6$, and the current sample is added to win . In this case, $SMin$ is updated, while $SMax$ remains unchanged. Figure 3.46 shows the information after the 6th iteration is completed.

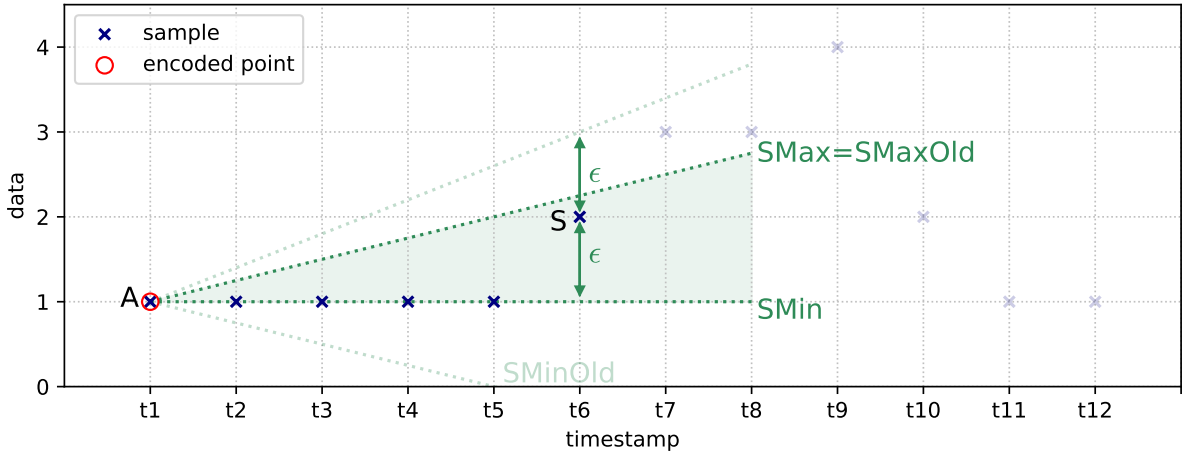


FIGURE 3.46: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 6.

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

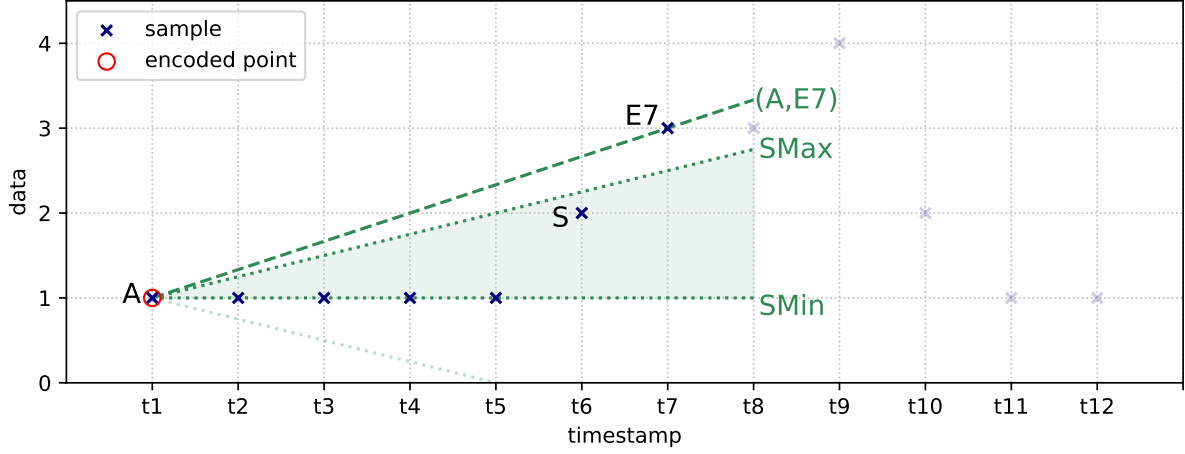


FIGURE 3.47: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 7.

For the first time, the incoming point, $E7$, lies outside of the cone, so $(A, E7)$ has a greater slope than $SMax$, and the condition in Line 12 becomes true. Therefore, auxiliary routine EncodeWinEnd is invoked, which uses $\lceil \log_2 w \rceil = 8$ bits for encoding the size of win (5), and B_c bits for encoding its last sample (2). Next, win is emptied, A is made equal to $E7$, and the current sample is encoded via the routine EncodeWinStart, using $\lceil \log_2 w \rceil = 8$ bits for encoding a 1, and B_c bits for encoding its value (3). In the following iteration, win is empty, so the condition in Line 7 is satisfied. Therefore, S is made equal to $E8$, the current sample (3) is added to win , and $SMin$ and $SMax$ are defined once more. Figure 3.48 shows the information after the 8th iteration is completed.

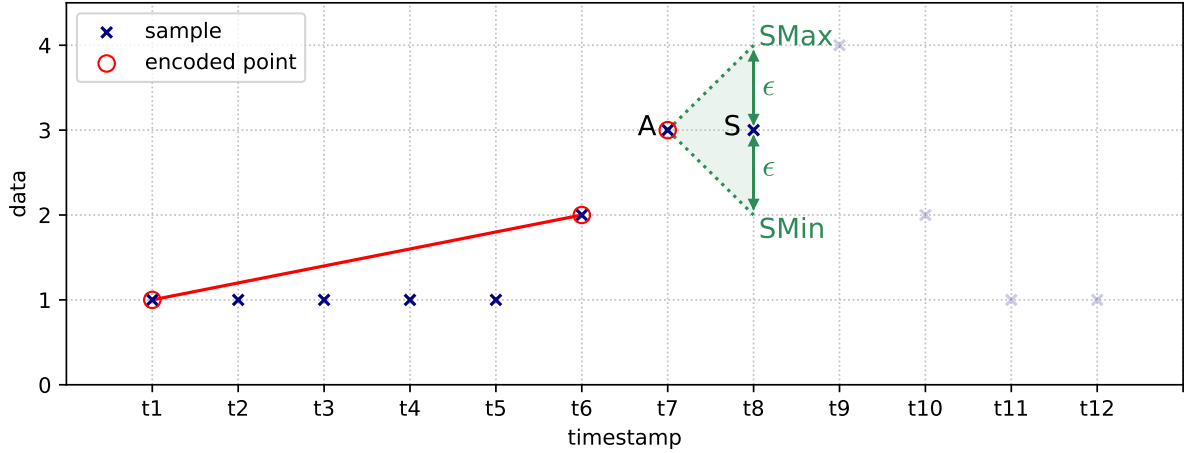


FIGURE 3.48: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 8.

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 EncodeWinEnd is invoked in Line 25. 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.

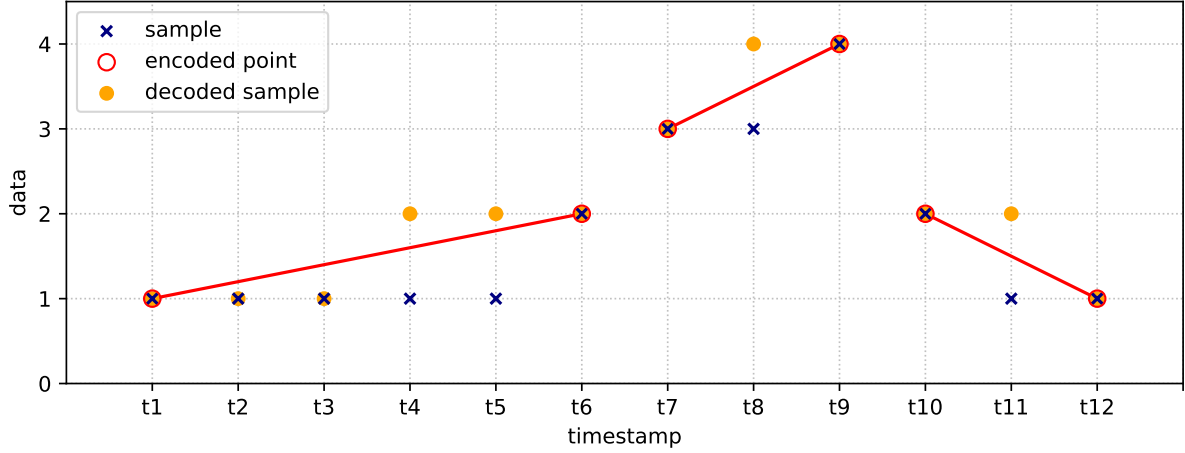


FIGURE 3.49: Example: Algorithm CA with $\epsilon = 1$ and $w = 256$. Step 9.

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 the incoming point lies outside of the cone determined by the two rays departing from point *A* (Line 12 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 values, or vice versa. We point out that, in the first scenario, it doesn’t make sense defining point *A*, so Line 15 is not executed in this case. For the same reason, Line 6 should not be executed if the first column entry is character “N”. Auxiliary routine EncodeWinEnd uses the same number of bits for encoding a window that consists of gaps and for encoding a window that consists of sample points, i.e. $\lceil \log_2 w \rceil + C$, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and C bits, where C is B_c bits, are used for encoding the special integer *NO_DATA*.

3.10 Algorithm SF

Algorithm *Slide Filter* (*SF*) [38] is a linear model algorithm that supports lossless and near-lossless compression. It has a window size parameter (w) that establishes a maximum block size for the algorithm. 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, 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 points representing entries in the same 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 that approximate the

points representing their respective entries, 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.

```

input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm SF
         $\epsilon$ : maximum error threshold
        w: maximum window size
        t_col: timestamp column
1  Create an empty window, win
2  foreach entry in column, entry, do
3      Obtain timestamp for entry,  $t_{entry}$ , from t_col
4      Let E be the point with coordinates ( $t_{entry}$ , entry)
5      if entry is the first entry in column then
6          Add entry to win, then continue
7      else if  $|win| == 1$  then
8          Let S be the point representing the single entry in win
9          Let SMin be the ray with initial point ( $S.x, S.y + \epsilon$ ), through point ( $E.x, E.y - \epsilon$ )
10         Let SMax be the ray with initial point ( $S.x, S.y - \epsilon$ ), through point ( $E.x, E.y + \epsilon$ )
11         Add entry to win, then continue
12     end
13     Let I be the point of intersection of SMin and SMax
14     Let (I, E) be the segment with initial point I that passes through point E
15     if not  $\text{slope}(SMin) \leq \text{slope}((I, E)) \leq \text{slope}(SMax)$  or  $|win| == w$  then
16         Let LS be the set of line segments such that  $s \in LS$  iff s has initial point I and
             $\text{slope}(SMin) \leq \text{slope}(s) \leq \text{slope}(SMax)$ 
17         if not segmentprev exists then
18             segment = EncodeWinStart(LS, win, ts_column, out) // rout. in Fig.3.52
19             Make segmentprev = segment
20         else
21             Let segmentconn  $\in LS$  be a line segment connected to segmentprev, which
                approximates the points representing entries in win
22             if segmentconn exists then
23                 EncodeWinEndStart(segmentprev, segmentconn, out) // rout. in Fig. 3.53
24                 Make segmentprev = segmentconn
25             else
26                 EncodeWinEnd(segmentprev, out) // routine shown in Figure 3.54
27                 segment = EncodeWinStart(LS, win, ts_column, out) // rout. in Fig.3.52
28                 Make segmentprev = segment
29             end
30         end
31         Set win to an empty window, then add entry to win
32     else
33         Add entry to win
34         Update SMin and SMax, considering the points representing entries in win
35     end
36 end

```

FIGURE 3.50: Coding routine for variant *M* of algorithm SF.

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. For the same reason, 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 SF
          $connected$ : boolean indicating if the segment is connected
1  Output bit  $connected$  to  $out$ 
2  Encode  $P.x$  as a float, using 32 bits
3  Encode  $P.y$  as a float, using 32 bits

```

FIGURE 3.51: Auxiliary routine EncodePoint for algorithm SF.

The coding routine operates by successively growing the current window, win , adding one entry at a time until it is complete. While win is being filled, the line segment that approximates the points representing the entries 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 representing the entries 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 22 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 that approximates the points representing the entries in the previous window, as well as the first endpoint of the segment that approximates the points representing the entries 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), and a new segment, $segment$, which approximates the points representing the entries 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 representing the entries in win , of which its slope is known and its first endpoint is encoded, is always saved as $segment_{prev}$ (lines 24 and 28), so that it is available in the following iterations, when the process of filling win , which is emptied and added the current sample in Line 31, continues.

```

input :  $LS$ : set of line segments
          $win$ : window to encode
          $t\_col$ : timestamp column
          $out$ : binary file encoded with algorithm SF
output:  $segment$ : line segment that approximates the points representing entries in  $win$ 
1  Let  $segment \in LS$  be the line segment that minimizes the MSE for the points
   representing entries in  $win$  (among the segments included in  $LS$ )
2  Let  $P$  be the first endpoint of  $segment$ 
3  EncodePoint( $P$ ,  $out$ , true) // routine shown in Figure 3.51
4  return  $segment$ 

```

FIGURE 3.52: Auxiliary routine EncodeWinStart for algorithm SF.

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

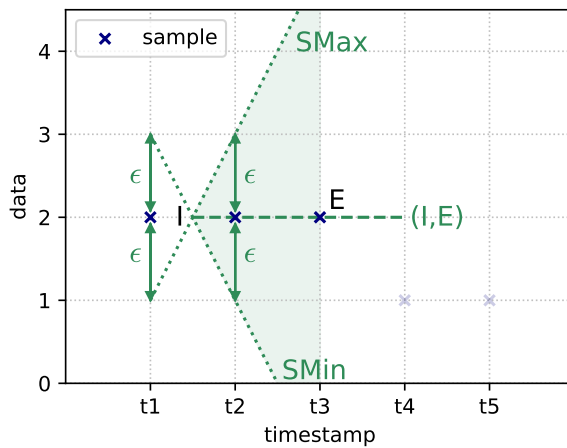
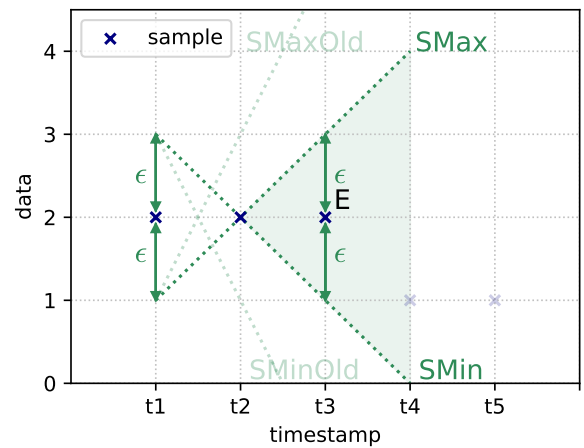
FIGURE 3.53: Auxiliary routine EncodeWinEndStart for algorithm SF.

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

FIGURE 3.54: Auxiliary routine EncodeWinEnd for algorithm SF.

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 point associated 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 the current column entry is added to the current window as long as point E lies within the cone. This cone, in turn, is defined so that if the current column entry is indeed added to the window, all the points representing the entries in the window are at a vertical distance of at most ϵ from the 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 first sample is added to win (Line 6), while in the second iteration the two rays are defined (lines 8-10), and the second sample is added to win (Line 11). In the third iteration, E lies within the cone determined by the two rays, so the condition in Line 15 evaluates to false (we assume that $|win| < w$), i.e. win is not complete, and lines 33-34 are executed: the third sample is added to win (Line 33), and $SMin$ and $SMax$ are updated (Line 34). The updated information is shown in Figure 3.56.

FIGURE 3.55: Example: key steps of the coding algorithm. Algorithm SF with $\epsilon = 1$. Step 1.FIGURE 3.56: Example: key steps of the coding algorithm. Algorithm SF with $\epsilon = 1$. Step 2.

In the successive iterations, the cone keeps reducing its slope, 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 representing entries in win [38].

Auxiliary routine EncodeWinStart, shown in Figure 3.52, is invoked, after win is complete, in two 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 17 is satisfied, and EncodeWinStart is invoked in Line 18. The second scenario is when a connected segment approximating the points representing the entries in win cannot be found. In this case, the condition in Line 22 evaluates to false, and auxiliary routines EncodeWinEnd and EncodeWinStart are invoked (lines 26-27). In both scenarios, the inputs are the same: LS is the set of line segments defined in Line 16, which consists of every segment with initial point E 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 representing the entries in win , and then encodes its first endpoint. $segment$ is the line segment that minimizes the MSE for the points representing entries in win , among the segments included in LS . $segment$ must belong to LS because this guarantees that every point representing an entry in win is at a vertical distance of at most ϵ from $segment$. The x and y-coordinates for the points are obtained from inputs t_col and win , respectively.

The decoding routine 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). 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). On the other hand, if the segment is not connected, the first endpoint of the subsequent encoded segment must be decoded (Line 16).

```

input : in: binary file coded with algorithm SF
        out: decoded column of CSV data file
        col_size: number of entries in the column
        t_col: timestamp column
1  n = 0
2  while n < col_size do
3      P, connected = DecodePoint() // routine shown in Figure 3.58
4      if n == 0 then
5          | Let Po = P
6          | continue
7      end
8      Let Pf = P
9      samples = FillSegment(t_col, Po.x, Pf.x, Po.y, Pf.y) // routine in Figure 3.17
10     foreach sample in samples, value, do
11         | Output value to out
12     end
13     if connected then
14         | Let Po = Pf
15     else
16         | Po, connected = DecodePoint() // routine shown in Figure 3.58
17     end
18     n += |samples|
19 end

```

FIGURE 3.57: Decoding routine for variant *M* of algorithm SF.

```

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

```

FIGURE 3.58: Auxiliary routine DecodePoint for algorithm SF.

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 distance between adjacent 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, the current column entry is added to the current window, win , as long as the incoming point (E) lies within the cone determined by the two rays, $SMin$ and $SMax$. In the first iteration, the condition in Line 5 is satisfied, so the first sample (1) is added to win (Line 6). In the second iteration, since win has a single element, the condition in Line 7 is satisfied, so using points S and $E2$, and parameter ϵ , rays $SMin$ and $SMax$ are defined (lines 8-10). Both rays are shown in Figure 3.59. The second sample (1) is also added to win (Line 11).

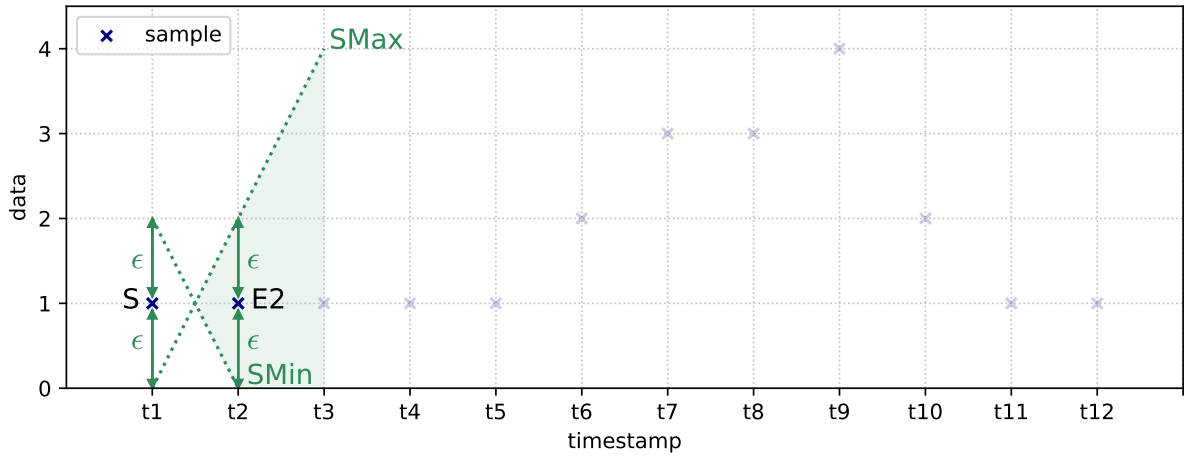


FIGURE 3.59: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 1.

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 13-14, is shown.

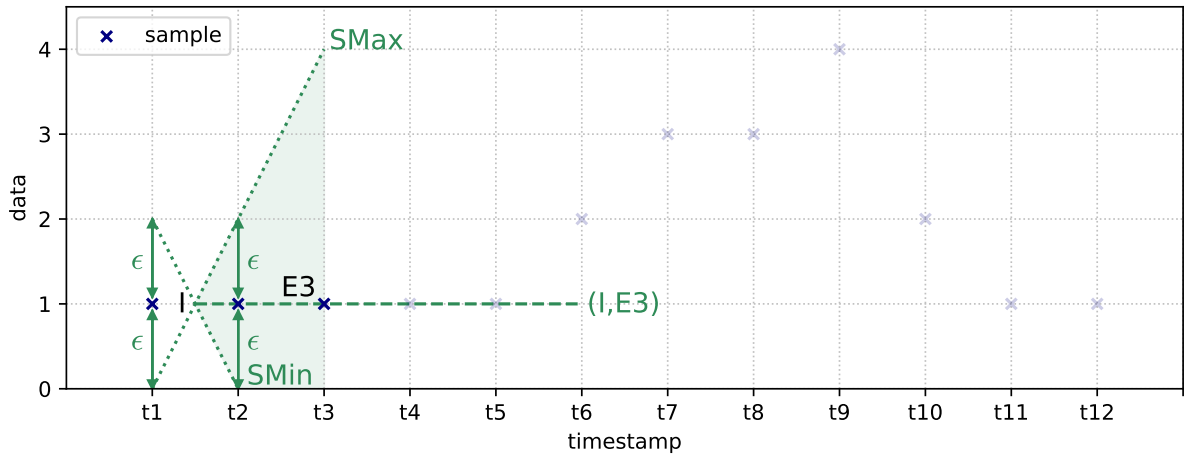


FIGURE 3.60: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 2.

Since $E3$ lies within the cone determined by the two rays, the condition in Line 15 evaluates to false. Therefore, the third sample is added to win (Line 33), and $SMin$ and $SMax$ are updated (Line 34). Figure 3.61 shows the information after the third iteration is completed.

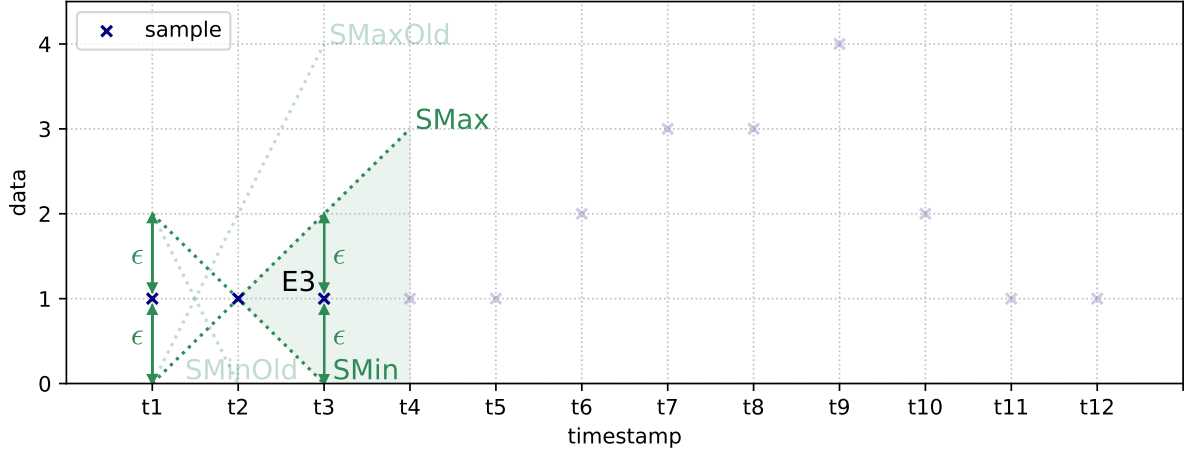


FIGURE 3.61: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 3.

The following six iterations are similar to the last one. In each iteration, the sample values are added to the window, and both rays are updated, reducing the slope of the cone. Figure 3.62 shows the information after the 9th iteration is completed. At this point, win includes the first 9 samples, and no bits have yet been written in the binary file.

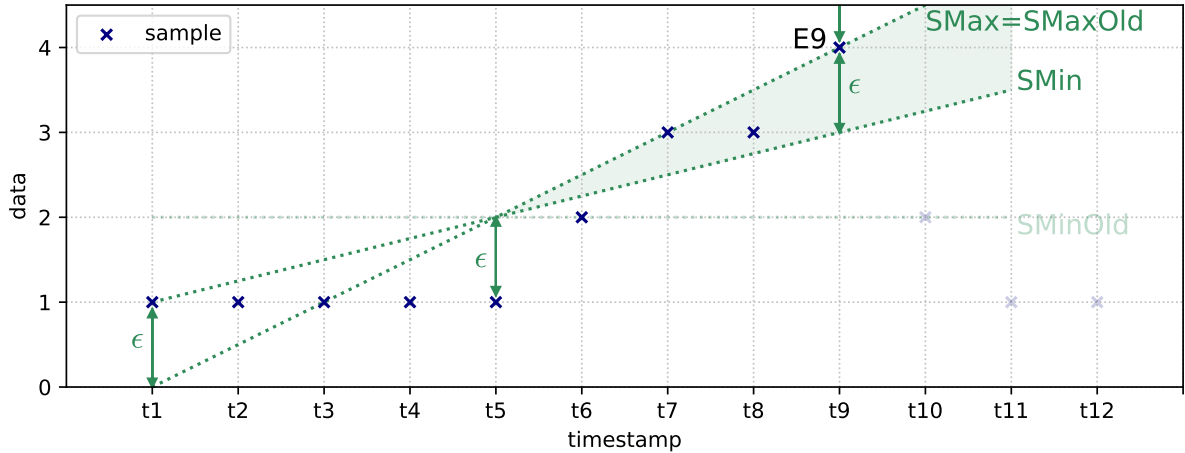


FIGURE 3.62: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 4.

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

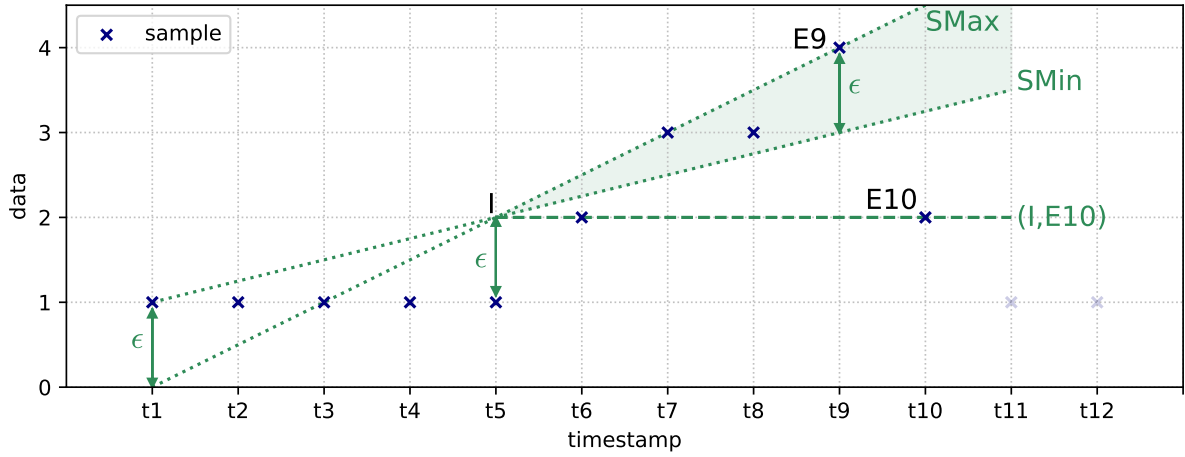


FIGURE 3.63: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 5.

For the first time, the incoming point, $E10$, lies outside of the cone, due to $(I, E10)$ having a smaller slope than $SMin$. Since no segment has yet been encoded, the condition in Line 17 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 16. Routine `EncodeWinStart` finds the segment included in set LS that minimizes the MSE for the 9 points representing entries in win (among the segments included in LS), and encodes its first endpoint. 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 19), and w is emptied and added the current sample (Line 31).

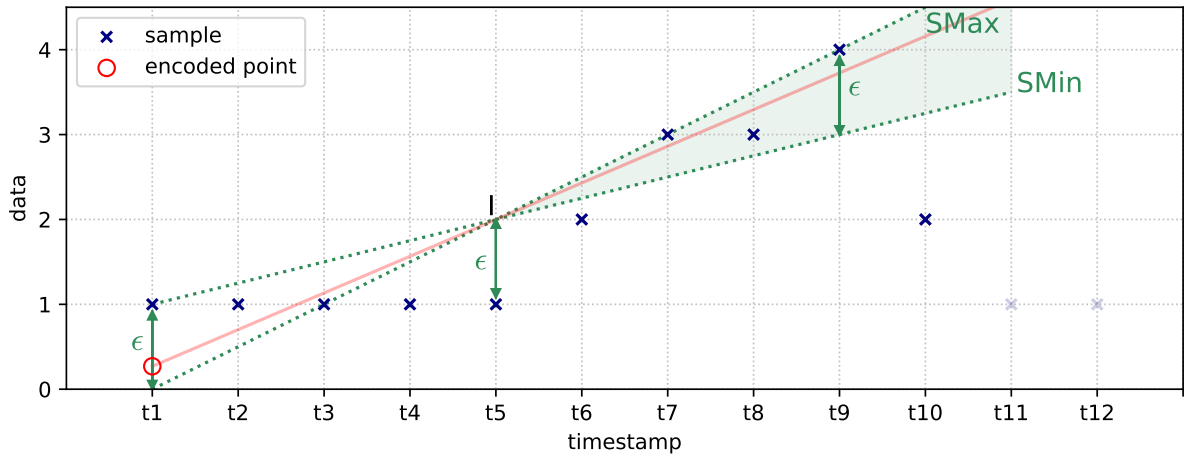


FIGURE 3.64: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 6.

In the 11th iteration, since win has a single element, the condition in Line 7 is satisfied, so using points S and $E11$, and parameter ϵ , rays $SMin$ and $SMax$ are defined (lines 8-10). Both rays are shown in Figure 3.65. The 11th sample (1) is also added to win (Line 11).

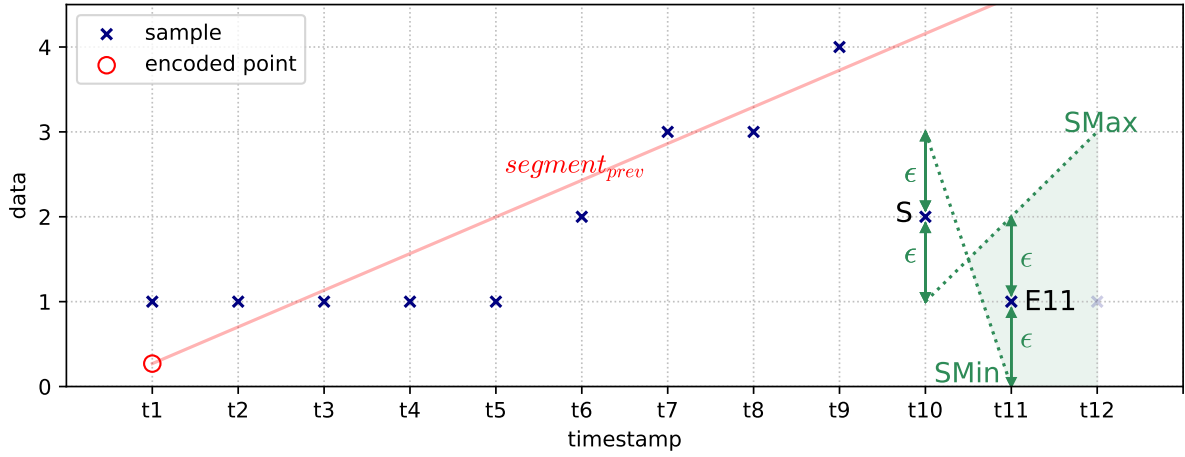


FIGURE 3.65: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 7.

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 13-14, is shown.

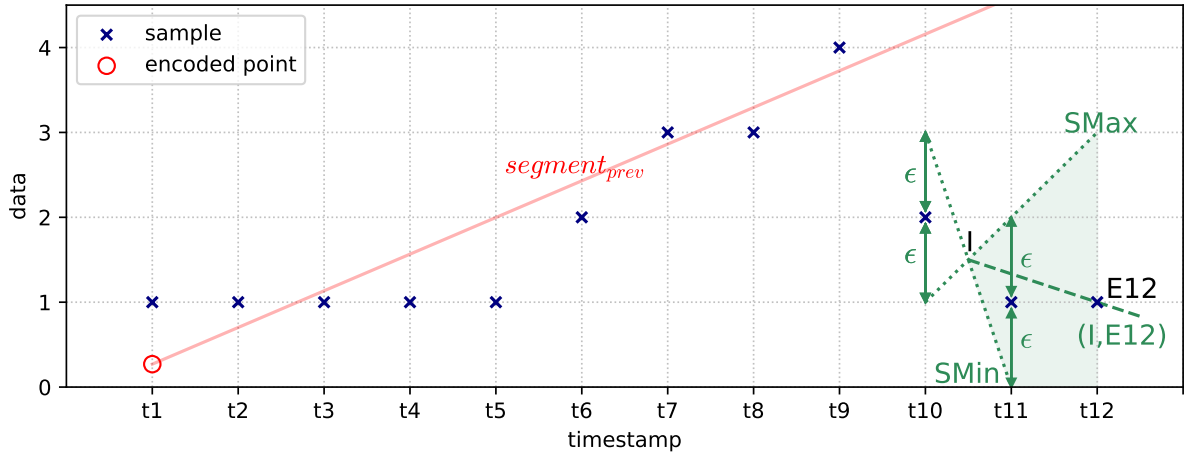


FIGURE 3.66: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 8.

Since $E12$ lies within the cone determined by the two rays, the condition in Line 15 evaluates to false. Therefore, the 12th sample is added to win (Line 33), and $SMin$ and $SMax$ are updated (Line 34). Figure 3.67 shows the information after the 12th iteration is completed. At this point, win includes the last three samples.

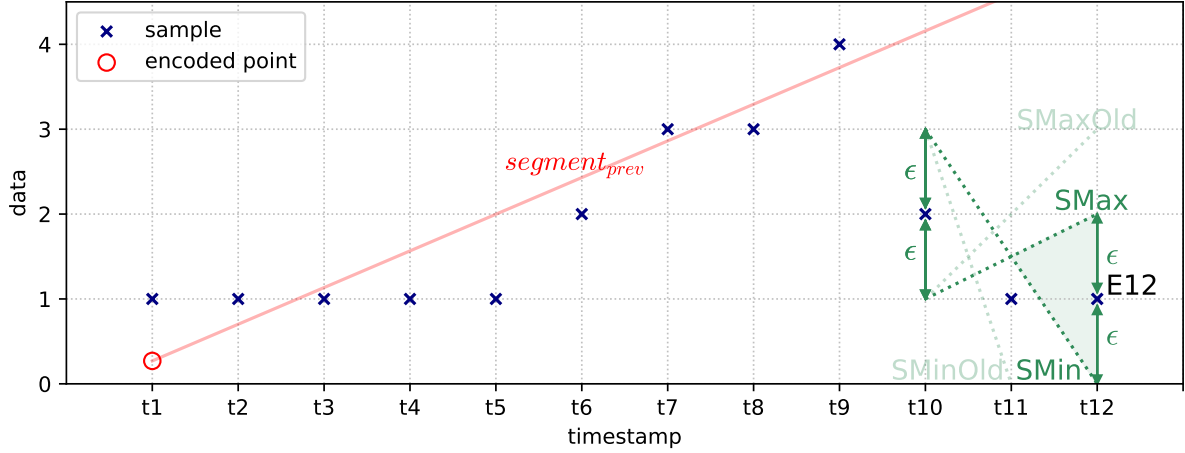


FIGURE 3.67: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 9.

After executing the last iteration of the loop, win is not empty, so a segment that approximates the three points representing entries in win must be encoded, executing the same code as in lines 16-30 (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 22 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. 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.

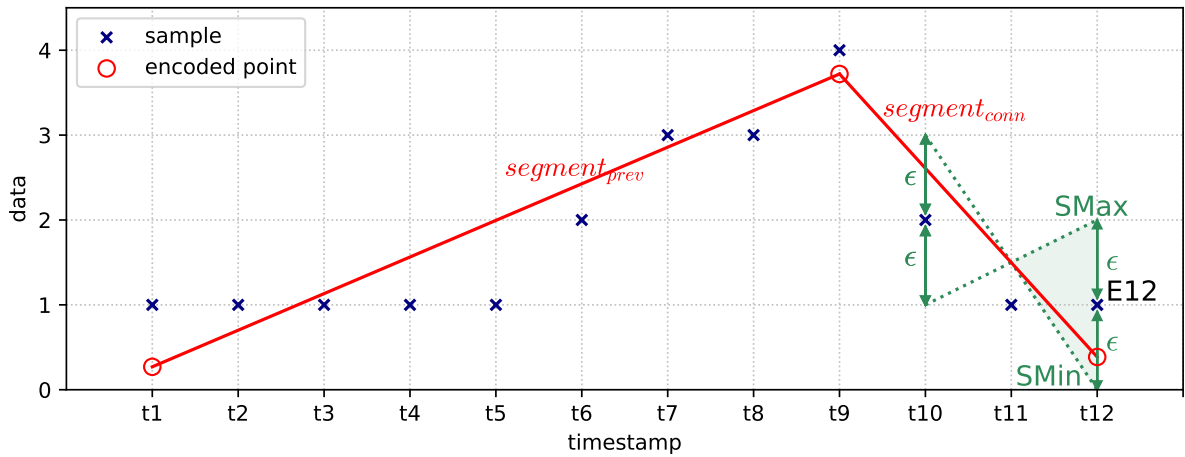


FIGURE 3.68: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 10.

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.

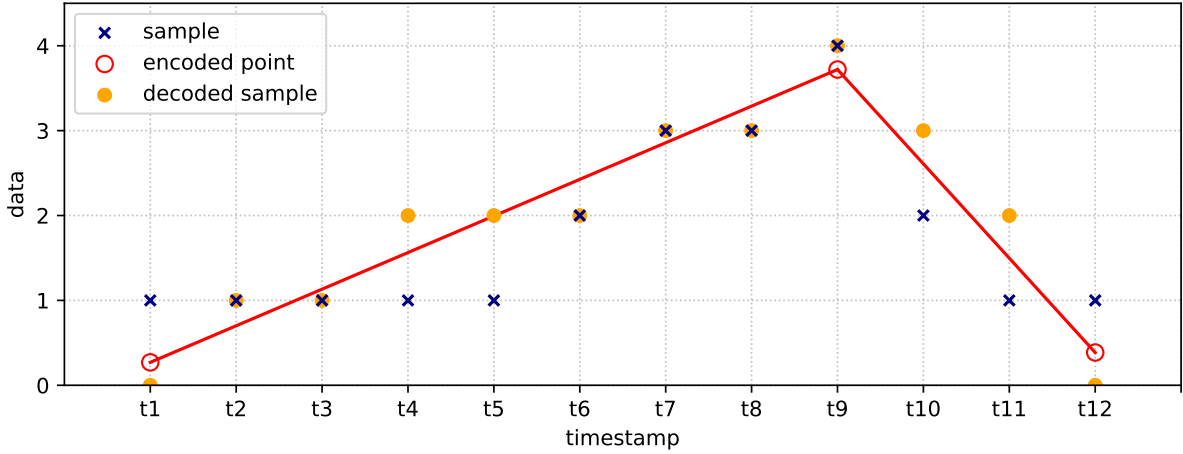


FIGURE 3.69: Example: Algorithm SF with $\epsilon = 1$ and $w = 256$. Step 11.

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. It has a window size parameter (w) that establishes the maximum block size in which the data are separately encoded. For FR we define a single variant, M .

FR is a linear model algorithm, so from Section 3.7 we recall that its encoding process involves encoding a sequence of line segments. In Figure 3.70 we show the coding routine, 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 points representing the entries in each window, win , 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)*.

Since w is always greater than 1, the condition in Line 3 might only be satisfied in the last iteration, i.e. the iteration corresponding to the last window. In the rest of the cases, an empty list, *displaced_points*, is created (Line 8), and passed to a recursive auxiliary routine *GetDisPointsMDP* (Line 9). This routine fills the list with the indexes (relative to win , in ascending order) corresponding to the respective y-coordinates of the displaced points obtained for win (their x-coordinates are obtained from the timestamp column, t_col). The first and last entries in *displaced_points* are always 0 and $w - 1$, respectively, since the first endpoint of the first segment always corresponds to the first entry in win , and the last endpoint of the last segment always corresponds to the last entry in win . In lines 10-13, each of the displaced points for win is encoded, using $\lceil \log_2 w \rceil$ bits for encoding their index (relative to win), *index*, and B_c bits for encoding their y-coordinate, $win[index]$.

```

input : column: column of the CSV data file to be encoded
        out: binary file encoded with algorithm FR
         $\epsilon$ : maximum error threshold
        w: maximum window size
        t_col: timestamp column
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  $|win| == 1$  then
4          Let value be the single entry in win
5          Encode value using  $B_c$  bits
6          return
7      end
8      Create an empty list, displaced_points
9      GetDisPointsMDP(win, t_col,  $\epsilon$ , displaced_points, 0,  $|win| - 1$ ) // routine in Fig. 3.71
10     foreach entry in displaced_points, index, do
11         Encode index using  $\lceil \log_2 w \rceil$  bits
12         Encode win[index] using  $B_c$  bits
13     end
14 end

```

FIGURE 3.70: Coding routine for variant *M* of algorithm FR.

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 the points representing the entries in *win* that are between said endpoints. If the segment satisfies the *valid segment condition*, defined in Line 5, it is guaranteed that the encoded samples satisfy the per-sample absolute error constraint, which is specified via the maximum error threshold parameter (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 representing the entries 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
        t_col: timestamp column
         $\epsilon$ : maximum error threshold
        displaced_points: list including the indexes (relative to win, in ascending order) of
the respective y-coordinates of the displaced points
         $i_o, i_f$ : indexes (relative to win) of the y-coordinates 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  $P_o$  and  $P_f$  be the points whose y-coordinates are win[ $i_o$ ] and win[ $i_f$ ], and whose
   x-coordinates are obtained from t_col
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, t_col,  $\epsilon$ , displaced_points,  $i_o$ , half)
9      Recursively call GetDisPointsMDP(win, t_col,  $\epsilon$ , displaced_points, half,  $i_f$ )
10 end

```

FIGURE 3.71: Auxiliary routine GetDisPointsMDP for algorithm FR.

The decoding routine 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 3-7 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 values are decoded by means of obtaining the endpoints of the one or more line segments that model the points representing the entries in the 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 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 representing the entries 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 the last endpoint of a segment always coincides with the first endpoint of the subsequent segment.

```

input : in: binary file coded with algorithm FR
         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 variant M of algorithm FR.

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 distance between adjacent 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 samples, is created in Line 1 of the coding routine. In this case, since $|win|$ is greater than 1, the window is encoded in lines 8-13.

In Line 9, the auxiliary routine `GetDisPointsMDP`, presented in Figure 3.71, is invoked with the following inputs: win , t_col , $\epsilon = 1$, $displaced_points = []$, $i_o = 0$, and $i_f = 11$. Figure 3.73 shows the information after lines 1-5 in said routine are executed. $displaced_points$ is equal to $[0, 11]$, 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 representing the samples in win , with indexes between 0 and 11, 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).

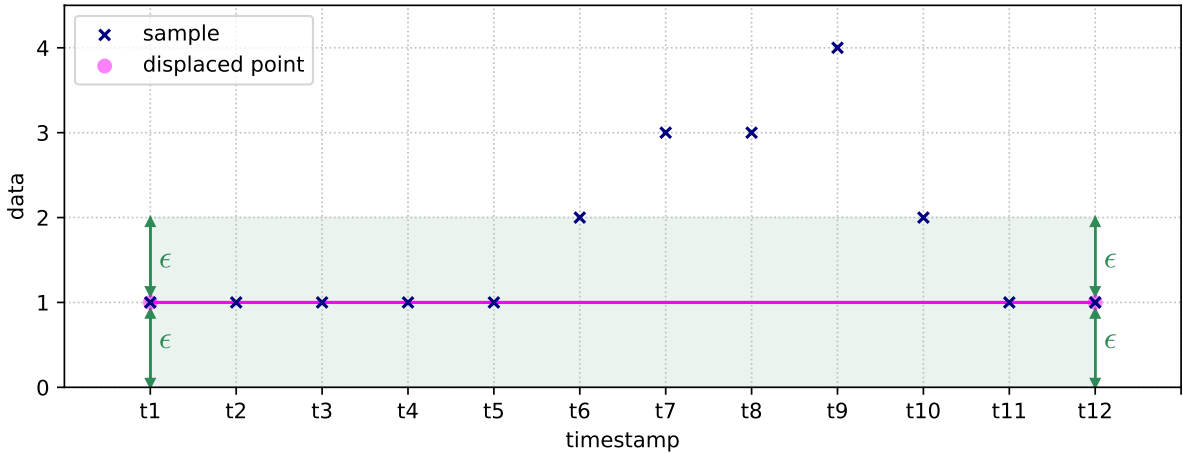
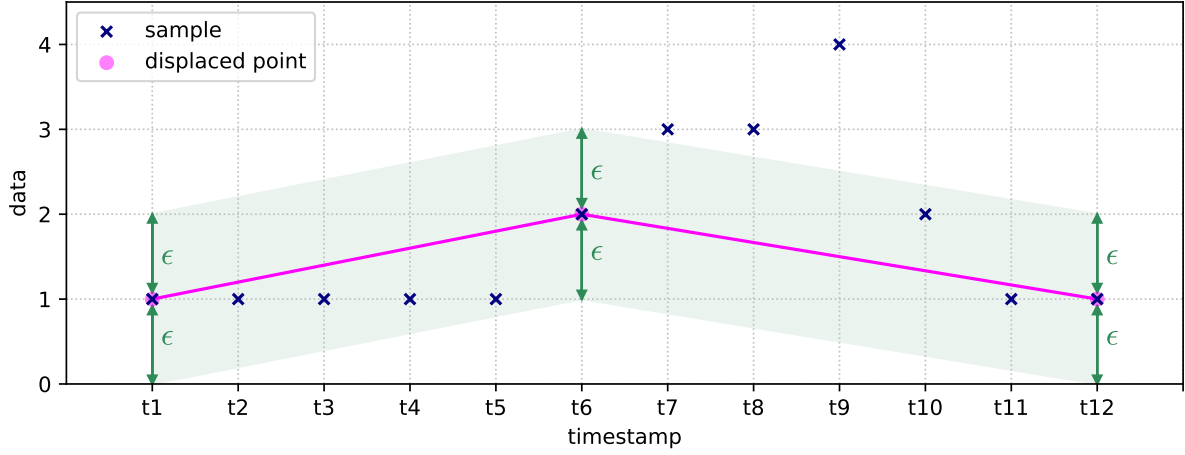


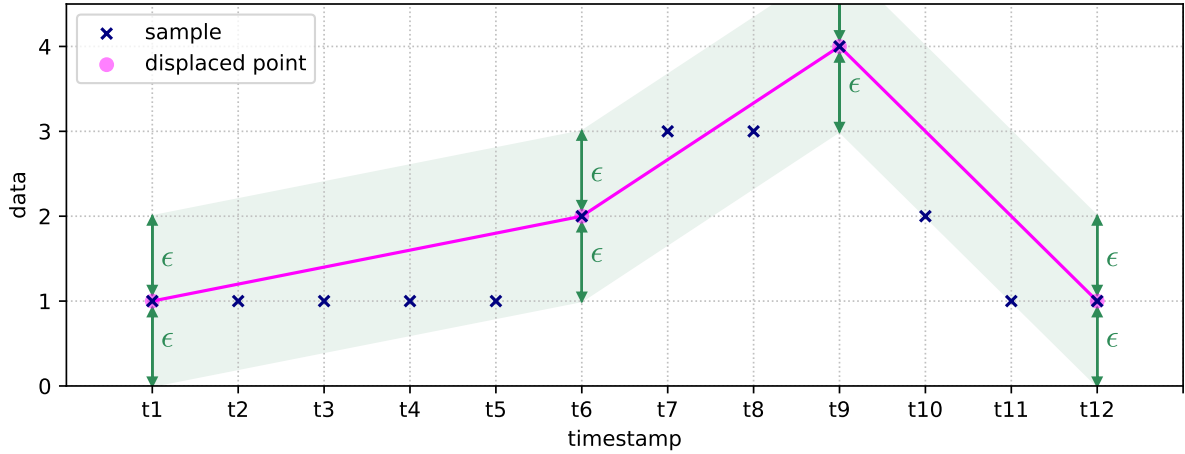
FIGURE 3.73: Example: Algorithm FR with $\epsilon = 1$ and $w = 256$. Step 1.

The first recursive call has inputs $displaced_points = [0, 11]$, $i_o = 0$ and $i_f = 5$, and it adds a single index to $displaced_points$, making it equal to $[0, 5, 11]$. 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 representing the samples 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 FR 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 $displaced_points = [0, 5, 11]$, $i_o = 5$ and $i_f = 11$. Here, Figure 3.74 shows that the valid segment condition is violated by three points representing the entries in the window. 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, $half = 8$ 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 9 in the coding routine, shown in Figure 3.70), list $displaced_points$ is equal to $[0, 5, 8, 11]$. This information is shown in Figure 3.75. Observe that, for every point corresponding to an entry in win , its vertical distance to the correspondent segment is at most ϵ .

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

In this example, the data points representing the samples 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 10-13 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. 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.

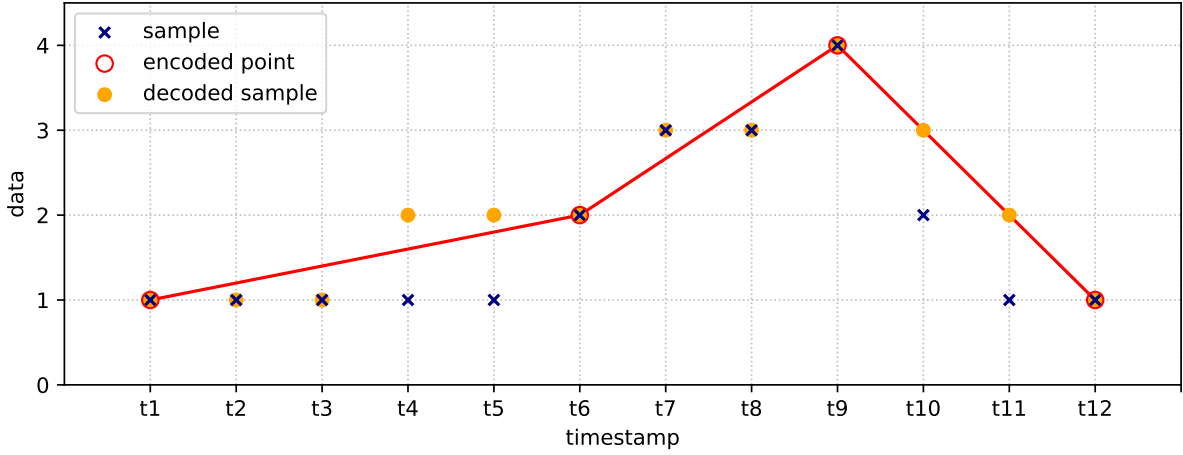


FIGURE 3.76: Example: Algorithm FR with $\epsilon = 1$ and $w = 256$. Step 4.

3.12 Algorithm GAMPS

Algorithm *Grouping and AMplitude Scaling (GAMPS)* [40] is a correlation model algorithm that supports lossless and near-lossless compression. It has a window size parameter (w) that establishes the maximum block size in which the data are separately encoded. For GAMPS we define both variants, M and NM .

Algorithm GAMPS follows the general schema 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*. The base column is encoded using algorithm APCA_F , which is quite similar to algorithm APCA, presented in Chapter 3.6, the only difference being that in the auxiliary routine `EncodeWindow`, shown in Figure 3.12, in the APCA_F case the *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 base column's samples are offset), then encoded, which is also done using APCA_F . The original ratio signals may be rough, but, since base and ratio signals in the same subset are spatially correlated, the transformed ratio signals are expected to be slowly varying. As we recall from Section 3.6, algorithm APCA achieves better compression performances on slowly varying signals rather than rough signals, and this is also the case for algorithm APCA_F . Therefore, the key idea behind algorithm GAMPS is that, under certain circumstances, it may be more convenient to exploit the spatial correlation between the signals, and encode, for each subset, a single rough base signal and a handful of slowly varying 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 GAMPS
          $\epsilon$ : 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  $\epsilon_b$  and  $\epsilon_r$ , for base and ratio columns,
   respectively.
2  Encode the number of subsets using  $\lceil \log_2 |\text{columns}| \rceil$  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  $\lceil \log_2 |\text{columns}| \rceil$  bits in every case
7      Encode base_col by calling the coding routine for variant  $M$  of algorithm  $\text{APCA}_F$ 
       with parameters base_col, out,  $\epsilon_b$ ,  $w$ 
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
            $\text{APCA}_F$  with parameters trans_ratio_col, out,  $\epsilon_r$ ,  $w$ 
11     end
12 end

```

FIGURE 3.77: Coding routine for variant M of algorithm GAMPS.

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 resolved [40]. Its inputs are the set of columns, *columns*, and the maximum error threshold, ϵ , and the output is the grouping and two maximum error threshold parameters, ϵ_b and ϵ_r , used for encoding the base and ratio columns, respectively. 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), and, 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_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, then encoded by invoking the APCA_F coding routine with an error threshold equal to ϵ_r .

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.

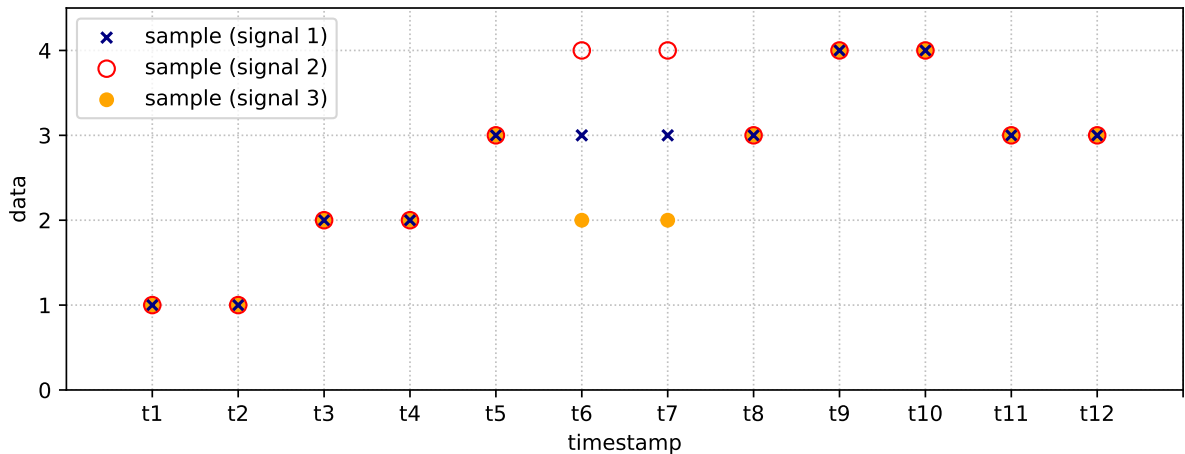


FIGURE 3.78: Example: Three signals consisting of 12 samples each.

There is a high degree of spatial correlation among 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 created. 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 encoded up to this point.

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), then encoded by calling the $APCA_F$ 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.

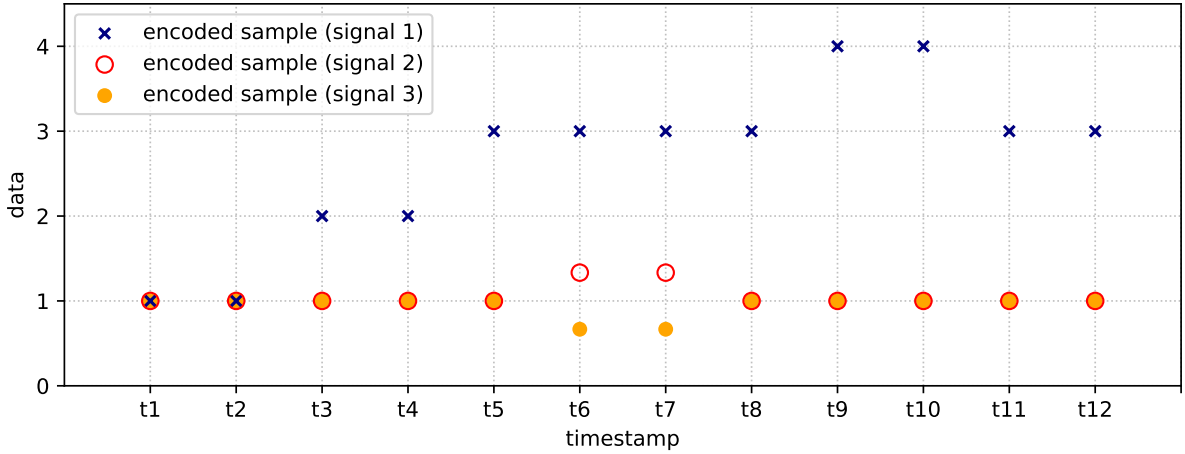


FIGURE 3.79: Example: Algorithm GAMPS with $\epsilon = 0$ and $w = 256$. Step 1.

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 a column-specific fixed number of bits is used for encoding its mid-range. Instead, the coding routine for algorithm $APCA_F$ 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 $APCA_F$, 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, encoding the base signal also involves encoding 5 windows, but 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. This is expected because, since there is a high degree of spatial correlation among the signals, the transformed ratio signals are slowly varying, compared to the original ratio signals, which are more rough.

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 algorithms presented in this chapter are implemented in C++. The implementation of algorithms PCA, APCA, CA and FR is completely ours, while algorithms PWLH, SF and GAMPS reuse part of the source code from the framework cited in [27]¹. We implemented algorithm PWLHInt by introducing the changes described in Section 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 consist of computing the convex hull of the data points by applying Graham's Scan algorithm [47] (Line 6 in Figure 3.19), and calculating the segment that minimizes the MSE for all the points representing entries in a window (Line 2 in Figure 3.20). In algorithm SF they consist of finding connected line segments (Line 21 in Figure 3.50), and calculating the segment that minimizes the MSE for all the points representing entries 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 [48, 49] of the AC, which is written in C.

¹The 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 performance of each of the coding algorithms presented in Chapter 3, by encoding the various datasets introduced in Chapter 2.

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 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 algorithms. 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 said file, with the results obtained when using the optimal window size for the whole dataset. 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.

In Section 4.4 we compare the performance of the different coding algorithms among each other, and with the general-purpose compression algorithm gzip. Among our implemented coding algorithms, for larger error thresholds APCA is the best algorithm for compressing every data type in our experimental datasets, while for lower thresholds the recommended algorithms are PCA, APCA and FR, depending on the data type. If we also consider algorithm gzip, for neither larger nor lower error thresholds there exists an algorithm that is better for compressing every data type. Depending on the data type, the recommended algorithms are APCA and gzip for larger error thresholds, and PCA, APCA, FR and gzip for lower thresholds.

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). There exist some $a \in A$ for which either a_M or a_{NM} is invalid (recall this information from Table 3.1). We denote by V the set of variants composed of every valid variant a_v for every algorithm $a \in A$. Also, we denote by A_M the subset of algorithms from A composed 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. 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 ϵ is calculated as a percentage fraction, given by parameter 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 $\epsilon = 2$, since 2 is the 10 percent of 20, so in this case the near-lossless compression allows for a maximal per-sample distortion of 2 sampling units. We test all the values of the error 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 $c_{\langle a_v, w, e \rangle}$ 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.,

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

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 a pair of 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

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

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

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

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

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

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

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

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

$$\text{OWS}(a_v, e, z, d) = \arg \min_{w \in W} \left\{ \text{CR}(c_{\langle a_v, w, e \rangle}, z, d) \right\}, \quad (4.6)$$

where we break ties in favor of the smallest window size.

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 $c_{\langle a_M, w_M^*, e \rangle}$ and $c_{\langle a_{NM}, w_{NM}^*, e \rangle}$, 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 $z = \text{"SST"}$ of the dataset $d = \text{SST}$, presented in Section 2.3, and Figure 4.2 shows the results for the data type $z = \text{"Longitude"}$ of the dataset $d = \text{Tornado}$, 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.

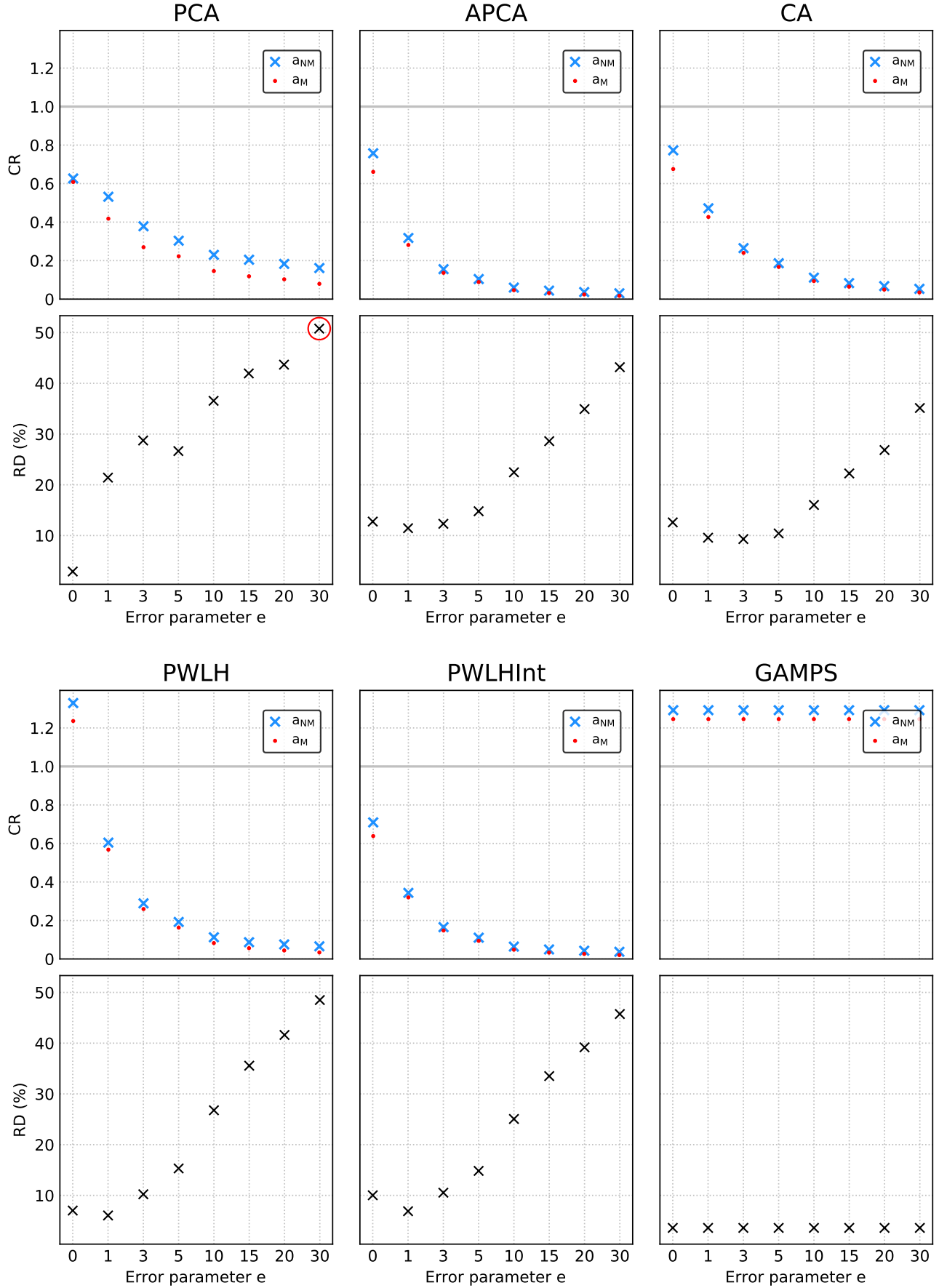


FIGURE 4.1: CR and RD plots for variants a_M and a_{NM} , for each algorithm $a \in A_M$, for the data type “SST” of the dataset SST. In the RD plot for algorithm PCA we highlight with a red circle the marker for the maximum value (50.78%) obtained for all the tested CAIs.

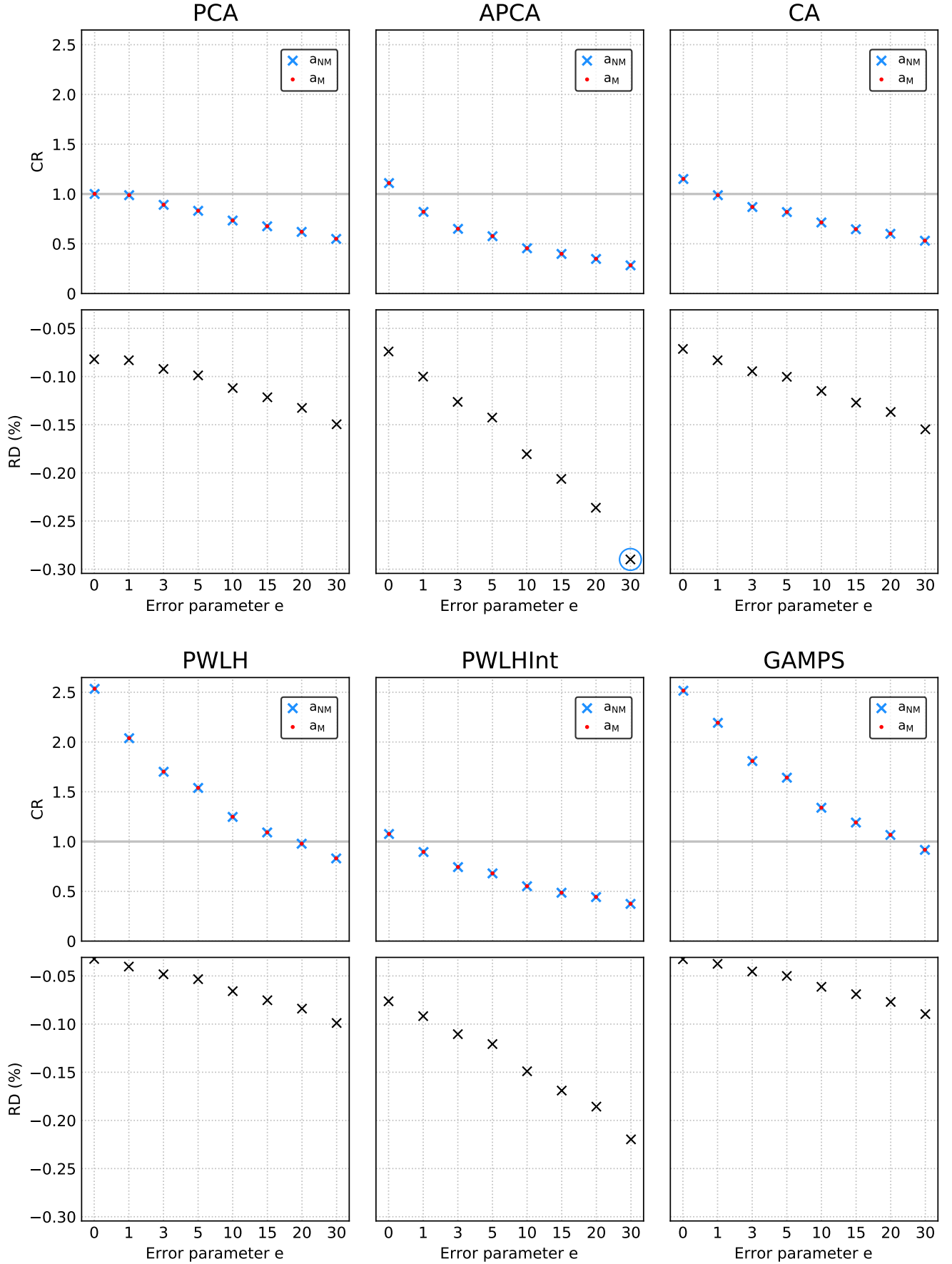


FIGURE 4.2: CR and RD plots for variants a_M and a_{NM} , for each algorithm $a \in A_M$, for the data type “Longitude” of the dataset Tornado. In the RD plot for algorithm APCA we highlight with a blue circle the marker for the minimum value (-0.29%) obtained for all the tested CAIs.

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 $c_{\langle a_M, w_{NM}^*, e \rangle}$ and $c_{\langle a_{NM}, w_{NM}^*, e \rangle}$, 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 algorithms. For these experiments we consider the set of algorithm variants 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

$$\text{LOWS}(a_v, e, z, f) = \arg \min_{w \in W} \left\{ \text{CR}(c_{\langle a_v, w, e \rangle}, z, f) \right\}, \quad (4.7)$$

where we break ties in favor of the smallest window size.

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 $c_{\langle a_v, w_{global}^*, e \rangle}$ and $c_{\langle a_v, w_{local}^*, e \rangle}$, as defined in (4.2), where $w_{global}^* = \text{OWS}(a_v, e, z, d)$ and $w_{local}^* = \text{LOWS}(a_v, e, z, f)$. 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 $c_{\langle a_v, w_{global}^*, e \rangle}$ and $c_{\langle a_v, w_{local}^*, e \rangle}$, as a function of the error parameter e , obtained for the data type $z = \text{"VWC"}$, for two different files of the dataset $d = \text{IRKIS}$, presented in Section 2.2. Figure 4.3 shows the results for the file $f = \text{"irkis-1202.csv"}$, and Figure 4.4 shows the results for $f = \text{"irkis-1203.csv"}$. 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 algorithm APCA 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.

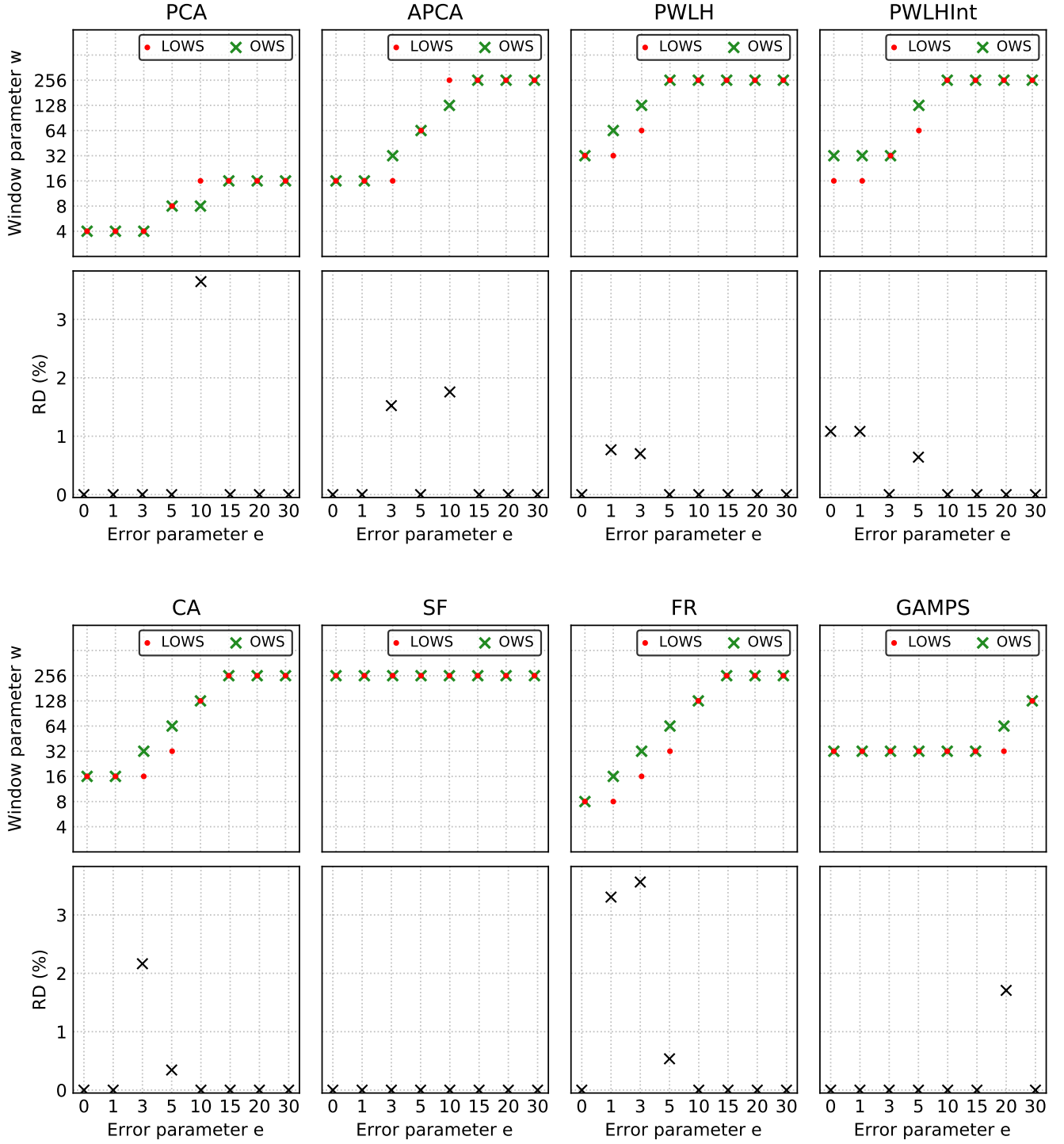


FIGURE 4.3: Plots of w_{global}^* , w_{local}^* , and the RD between $c_{<a_v, w_{global}^*, e>}$ and $c_{<a_v, w_{local}^*, e>}$, as a function of the error parameter e , obtained for the data type “VWC” of the file “irkis-1202.csv” of the dataset IRKIS.

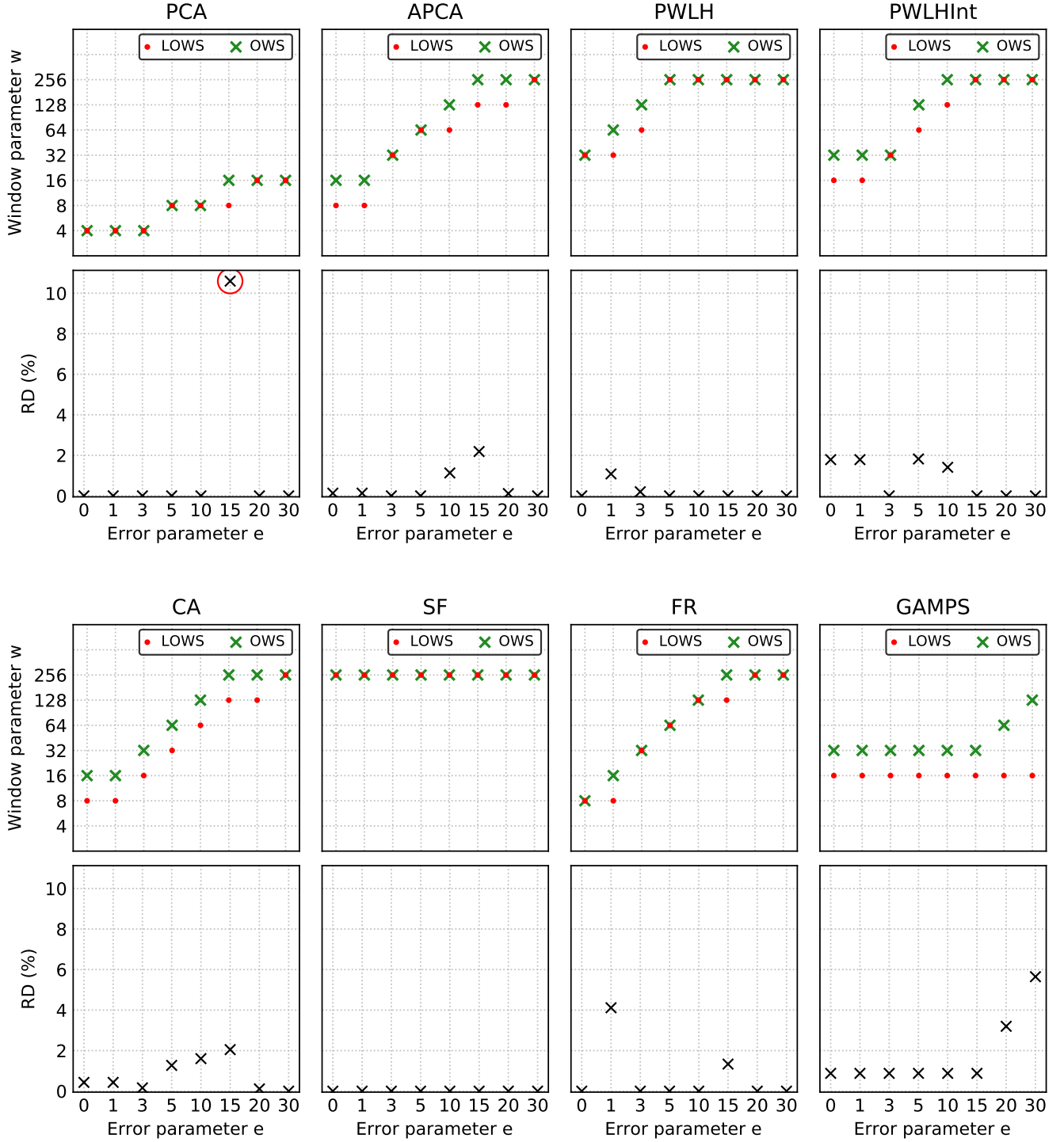


FIGURE 4.4: Plots of w_{global}^* , w_{local}^* , and the RD between $c_{<av, w_{global}^*, e>}$ and $c_{<av, w_{local}^*, e>}$, as a function of the error parameter e , obtained for the data type “VWC” of the file “irkis-1203.csv” of the dataset IRKIS. In the RD plot for algorithm PCA we highlight with 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.

Algorithm	RD (%) Range				
	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	172 (86%)	16 (8%)	6 (3%)	6 (3%)	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	1,433 (89.5%)	83 (5.2%)	48 (3%)	27 (1.7%)	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 172 (86%) of the 200 evaluated pairs of CAIs for that algorithm. The second column reveals that for 16 pairs of CAIs (8%), the RD takes values greater than 0 and less 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 89.5% 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 97.7% 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 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 SST, 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 Algorithms Performance

In this section, we compare the compression performance of the coding algorithms presented in Chapter 3, by encoding the various datasets introduced in Chapter 2. We begin by comparing the algorithms among each other and later we compare them with gzip, a popular general-purpose lossless compression algorithm. We analyze the performance of the algorithms 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 $c_{\langle a_v, w_{global}^*, e \rangle}$, 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.,

$$c^b(z, d, e) = \arg \min_{c \in CI} \left\{ \text{CR}(c_{\langle a_v, w_{global}^*, e \rangle}, z, d) \right\}. \quad (4.8)$$

When $c^b(z, d, e) = c_{\langle a_v^b, w_{global}^{b*}, e \rangle}$, we refer to a^b and w_{global}^{b*} as the *best coding algorithm* 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 $\text{CR}(c_{\langle a_v, w_{global}^*, e \rangle}, z, d)$ 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 $z = \text{"SST"}$ of the dataset $d = \text{ElNino}$, 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 algorithm). 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 algorithm, 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.

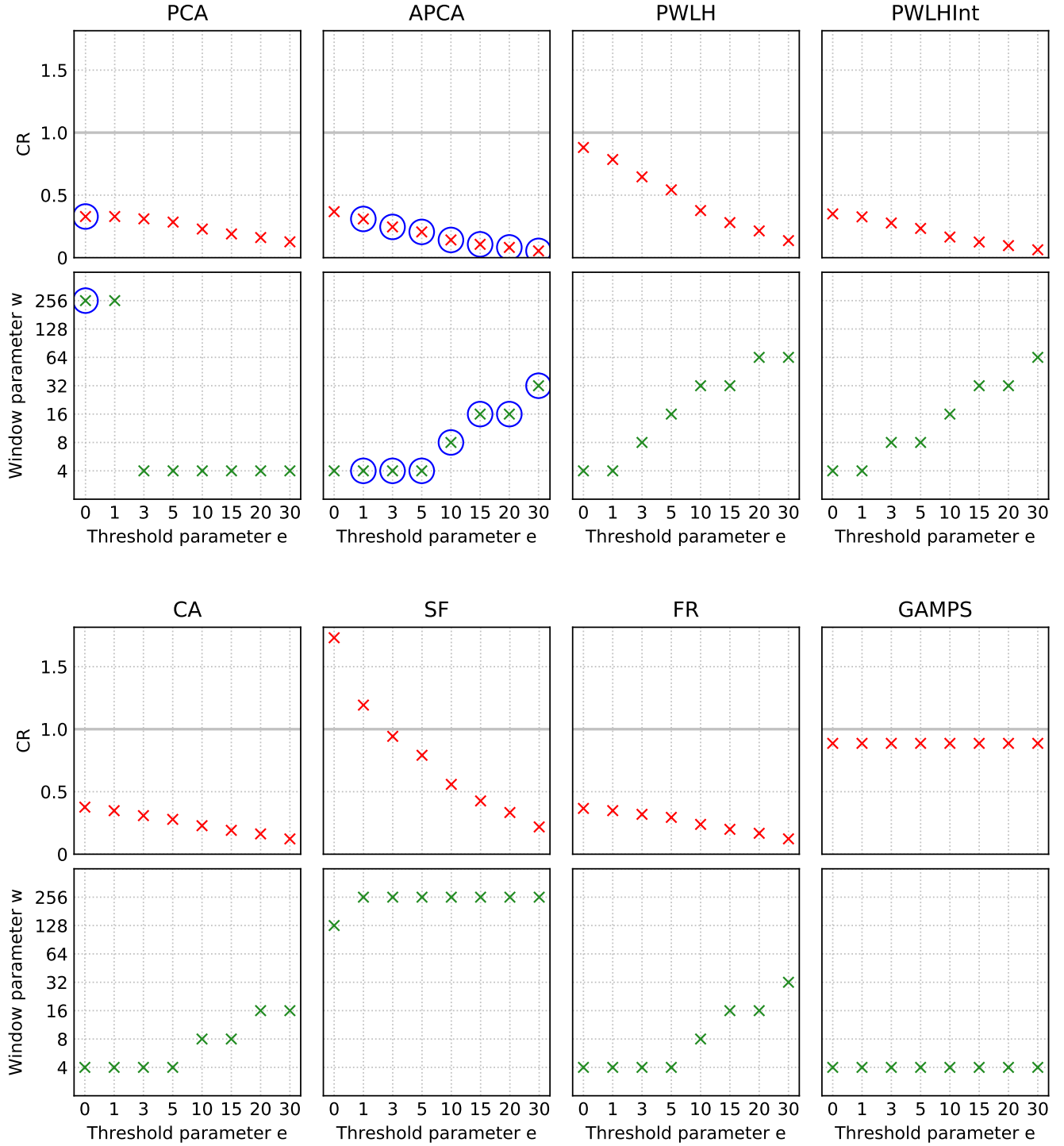


FIGURE 4.5: CR and window size parameter plots for every evaluated algorithm, for the data type “SST” of the dataset ElNino. For each error parameter $e \in E$, we use blue circles to highlight the markers for the minimum CR value and the best window size (in the respective plots corresponding to the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

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 “SST” 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		FR											
Dataset	Data Type	e = 0		e = 1		e = 3		e = 5		e = 10		e = 15		e = 20		e = 30	
		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	Vel	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	0.76	3	0.71	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 only three algorithms (PCA, APCA, 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 134 combinations (80%), 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, while FR is the best algorithm in only 3 combinations (2%), all of them for data type “Speed” of the dataset Wind.

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

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

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

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

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

Again, minmaxRD is 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.

Algorithm	maxRD (%)							
	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	38.28	38.28	54.68	63.12	65.44	72.94	77.21	81.84
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, PWLHInt is the minmax coding algorithm, with minmaxRD being equal to 29.72%. 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 $e = 0$, PWLHInt 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 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 [50].

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 results obtained by near-lossless algorithms.

We observe that algorithm gzip obtains the best compression results in 36 (21%) of the 168 possible data type and error parameter combinations. Algorithms APCA, PCA, and FR now obtain the best results in exactly 106 (63%), 23 (14%), and 3 (2%) of the total combinations, respectively. 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 . We also observe 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 only outperforms the rest of the algorithms in about a half (10 out of 21) of the data types.

		GZIP		PCA		APCA		FR									
Dataset	Data Type	e = 0		e = 1		e = 3		e = 5		e = 10		e = 15		e = 20		e = 30	
		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	Vel	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 $\text{maxRD}(a, e)$ 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 correspondent to the $\text{minmaxRD}(e)$ 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 be the obvious choice for compressing every data type when $e \geq 10$, as we had concluded in the previous section.

Algorithm	maxRD (%)							
	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	73.82	73.45	69.31	68.29	65.44	72.94	77.21	81.84
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: $\text{maxRD}(a, e)$ 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 $\text{minmaxRD}(a)$ value is highlighted.

Chapter 5

Conclusions and Future Work

TODO: REESCRIBIR...

In conclusion, our experimental results indicate that none of the implemented coding algorithms obtains a satisfactory compression performance in every scenario. This means that selection of the best algorithm is heavily dependent on the data type to be compressed, and the error threshold that is allowed. In addition, we have shown that, in some cases, even a general compression algorithm such as gzip can outperform our implemented algorithms. In general, according to our results, algorithms APCA and gzip achieve better compression results for larger error thresholds, while PCA, APCA, FR and gzip are preferred for lower thresholds. Therefore, if one wishes to compress certain data type, our recommended way for choosing the appropriate algorithm is to select the best algorithm for said data type according to Table 4.5.

In our research, we have also compared the compression performance of the masking and non-masking variants of each coding algorithm. The experimental results show that on datasets with few or no gaps both variants have a similar performance, while on datasets with many gaps the masking variant always performs better, sometimes achieving a significant difference. We concluded that the masking variant of a coding algorithm is preferred, since it is more robust and performs better in general.

In addition, we have studied the extent to which the window size parameter impacts the compression performance of the coding algorithms. We analyzed the compression results obtained when using optimal global and local window sizes. The experimental results reveal that the effect of using the optimal global window size, instead of the optimal local window size for each file, is rather small. Thus, we could fix the window size parameter in advance, for example by optimizing over a training set, without compromising the performance of the coding algorithm. This is relevant, since calculating the optimal local window for a file is, in general, computationally expensive.

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

- In Section 3.2 we mention that we focus our work on the compression of the sample columns of the datasets, and do not delve into the optimization of the timestamp compression. This is an interesting problem to investigate in the future.
- In our experimental analysis, which is presented in Chapter 4, we assess the compression performance of the implemented algorithms 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 algorithms obtains better compression results for each data type in the experimental datasets. It would be useful to analyze *why* is it the case that a certain algorithm achieves better results for a certain data type. In order to do this, we would need to examine the signals for each data type, analyze their characteristics (e.g. whether they are slowly varying or rough signals, the number of outliers, periodicity), and observe if there exists a relation between these characteristics and the algorithm that obtains the best compression performance. This would be useful to predict which algorithm 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 for the signal, this could prove to be beneficial for online compression, as it would allow us to select a different compression algorithm as the trends in the signal vary over time.

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 February 23, 2021].
- [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 February 23, 2021].
- [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 February 23, 2021].
- [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 February 23, 2021].
- [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 February 23, 2021].
- [10] GNU Gzip. <https://www.gnu.org/software/gzip/>, 2020. [Accessed February 23, 2021].
- [11] WinRAR at a glance. <https://www.win-rar.com/features.html>, 2021. [Accessed February 23, 2021].
- [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 February 23, 2021].
- [15] M. Tramz. Photographing Pluto: This Is How New Horizons Works. <https://time.com/3944157/new-horizons-pluto/>, July 14, 2015. [Accessed February 23, 2021].
- [16] K. Sohrawy, 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 February 23, 2021].
- [30] El Nino Data Set. <https://archive.ics.uci.edu/ml/datasets/El+Nino>, 1999. [Accessed February 23, 2021].

- [31] NOAA - TAO Data Download. https://tao.ndbc.noaa.gov/tao/data_download/search_map.shtml, 2016. [Accessed February 23, 2021].
- [32] SolarAnywhere - Data. <https://data.solaranywhere.com/Data>, 2020. [Accessed February 23, 2021].
- [33] Storm Prediction Center - Severe Weather Database Files (1950-2017). <https://www.spc.noaa.gov/wcm/#data>, 2016. [Accessed February 23, 2021].
- [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 February 23, 2021].
- [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 February 23, 2021].
- [46] J.J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.
- [47] R. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1:132–133, 1972.
- [48] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, 1987.

-
- [49] Data Compression With Arithmetic Coding. <https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html>, 2014. [Accessed February 23, 2021].
- [50] GNU Gzip Manual - Invoking gzip. https://www.gnu.org/software/gzip/manual/html_node/Invoking-gzip.html, 2018. [Accessed February 23, 2021].