Universidad de la República

Tesis de Maestría

Coding of Multichannel Signals with Irregular Sampling

Autor:
Pablo Cerveñansky

Supervisores:
Álvaro Martín
Gadiel Seroussi

Núcleo de Teoría de la Información
Facultad de Ingeniería

November 9, 2020

Abstract

TODO

Table of Contents

List of Figures

Índice de tablas

Cambios de la versión anterior (9/10/2020) a esta versión (9/10/2020)

- Terminé el Chapter 2 (Datasets) y el Chapter 3 (Algorithms), ya estarían prontos para leer.
- En Chapter 3 (Algorithms), en varios lugares quedó la referencia a *column*.total_bits. *column*.total_bits representa el número fijo de bits, que depende del tipo de datos de la columna, que utilizo para codificar un sample. Debería cambiarlo por alguna otra cosa.
- En Chapter 4 (Experimental Results), tengo que hacer algunos cambios en las gráficas porque ahora el algoritmo SF soporta el parámetro del tamaño de ventana. Además, las gráficas de CR van a quedar mejores porque el algoritmo SF ahora codifica floats en vez de doubles, así que todas las gráficas van a quedar más a escala (antes los resultados de CR del algoritmo SF siempre eran mucho peores).

Chapter 1

Introduction

Reutilizar lo que se pueda de las presentaciones del Pedeciba:

2018

https://docs.google.com/presentation/d/1EtYbM5shn685DfP9qLd2E89LBBwyIevcM6oUQuE8RJA/edit
https://docs.google.com/document/d/1rBx11Ka9GhvohEkdwrMEuGeWtiitKOnbLrZ7yoOjjuE/edit

2019

https://docs.google.com/presentation/d/19glwhXE3IjQgQr-LBK5XV1lECWNcqBoEMdC9AlLMSaE/edit
https://docs.google.com/document/d/1c8W0dungTl2JVxHxCAv0cXXZ-uuMtzEaMVbQssSzw6M/edit

Chapter 2

Datasets

In this chapter we present all the datasets considered in the project. Every dataset consists of signals with either one or both of the characteristics we are interested in, namely, irregular sampling and data gaps. In Chapter 4, we present our experimental results, which make use of these datasets to analyze the performance of various compression algorithms, which are presented in Chapter 3.

The datasets come from multiple sources [1–6], each representing the data with different formats. We transformed all the data to a consistent, homogeneous format, which can be easily adapted to represent different kinds of datasets. In Section 2.1 we describe said format, including an example file that shows how the data is represented. In the following eight sections we present every dataset, namely, IRKIS, SST, ADCP, ElNino, Solar, Hail, Tornado, and Wind, laying out the source, characteristics and relevant statistics of every signal. In Section 2.10 we summarize information regarding each of the datasets presented in this chapter, describing their characteristics in terms of the amount of gaps, and the number of files and data types that each have.

2.1 Data format

The data from every dataset [1–6] was transformed to a consistent format, which can be adapted to represent different types of datasets. In Figure 2.1 we present an example of a 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 unique name, in this example ElNino, the measure unit for signal timestamps, and the timestamp value for the first sample. The fourth row has the labels of the data columns, and the remaining rows consist of the actual signal data.

DATASET:	ElNino					
TIME UNIT:	hours					
FIRST TIMESTAMP:	1980-03-07 00:00:00					
Time Delta	Lat	Long	ZonWinds	MerWinds	AirTemp	SST
0	-2	-10946	-68	7	2614	2624
24	-2	-10946	-49	11	N	N
24	-2	-10946	-45	22	N	N
48	-1	-10946	-38	19	N	2431
24	-2	-10946	-42	15	2557	2319
48	-2	-10946	-44	3	2472	2364
24	-2	-10946	-32	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 but the first timestamp is represented as an integer value, which is an increment with respect to the previous timestamp, measured in the unit specified in the second row. As mentioned, the value of the first timestamp is given in the third row. In the example presented in Figure 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 irregular sampling (recall that this is one of the characteristics we are interested in).

Besides the timestamp column, the csv presented in Figure 2.1 consists of six additional data columns, each representing a different data type. The first two columns represent the latitude and longitude coordinates, respectively, of a buoy floating in the ocean, while the last four columns represent readings of various physical magnitudes (wind velocity, air temperature, and sea temperature), made by sensors set in the buoy. In general, the timestamp column can only consist of integer values, while the rest of the columns can have both integer values and the character "N". An integer value represents an actual data sample, whose range depends on the range and accuracy of the sampling instrument used for acquiring and storing the data, 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 Figure 2.1, there are some gaps in the last two data columns (recall that this is another of the characteristics we are interested in).

2.2 Dataset IRKIS

Dataset IRKIS [1] 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 with the snowpack runoff and catchment discharge in high alpine terrain [2].

The dataset corresponds to seven stations, labeled 1202, 1203, 1204, 1205, 222, 333, and SLF2, which can be located in the map presented in [1]. 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.

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

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: Statistics of dataset IRKIS for each station. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

2.3 Dataset SST

Dataset SST [3] consists of *Sea Surface Temperature (SST)* measurements from buoys floating in the Pacific Ocean. This dataset is collected by the Tropical Atmosphere Ocean project (TAO), which was established in 1985 to study annual climate variations nears the equator [7].

The dataset consists of readings from 55 buoys, which can be located in the map presented in [3]. 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 degree Celsius (°C), with a precision of three significant figures. Every sample value is transformed to the integer domain by multiplying it by 10^3.

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

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: Statistics of dataset SST for each month. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

2.4 Dataset ADCP

Dataset ADCP [3] consists of water current velocity measurements from moorings in the Pacific Ocean. This dataset is collected by the Tropical Atmosphere Ocean project (TAO), which was established in 1985 to study annual climate variations nears the equator [7].

The dataset consists of readings from 3 moorings, which can be located in the map presented in [3]. 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, with a precision of three significant figures. Every sample value is transformed to the integer domain by multiplying it by 10^3.

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

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: Statistics of dataset ADCP for each month. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

2.5 Dataset ElNino

Dataset ElNino [4] 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 project (TAO), which was established in 1985 to study annual climate variations nears the equator [7].

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, and in each buoy the following measurements are taken: speed of the zonal and meridional winds (m/s), relative humidity (%), and air temperature and SST (°C). Therefore, a total of 546 (78×7) data samples are stored for each timestamp.

In Table 2.5 we present relevant statistics of dataset ElNino. The dataset consists of a single file, and each row in the table contains statistics concerning a different data type. The first column specifies the number of gaps, and the percentage over the 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.

Data Type	**#Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
Lat	318,858 (64.2)	-881	905	1	47.3	458.2
Long	318,858 (64.2)	-18,000	17,108	-11,126	-5,402.5	13,536.4
Zon.Wind	344,021 (69.2)	-124	143	-40	-33.0	33.7
Mer.Wind	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.	337,095 (67.8)	1,705	3,166	2,734	2,688.8	181.6
SST	335,865 (67.6)	1,735	3,126	2,829	2,771.5	205.7

Table 2.5: Statistics of dataset ElNino. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

2.6 Dataset Solar

Dataset Solar [5] 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, with a precision of zero significant figures. Therefore, the sample values are in the integer domain and it is not necessary to transform them.

In tables 2.6, 2.7, 2.8, and 2.9, we present relevant 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 concerning a different data type. The first column specifies the number of gaps, and the percentage over the number of entries. The rest of the columns show the minimum, maximum, median, mean, and standard deviation, of the sample values.

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: Statistics of dataset Solar for year 2011, for each data type. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

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: Statistics of dataset Solar for year 2012, for each data type. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

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: Statistics of dataset Solar for year 2013, for each data type. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

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: Statistics of dataset Solar for year 2014, for each data type. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

2.7 Dataset Hail

Dataset Hail [6] consists of hail size measurements from several stations around the US 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. The hail diameter is measured in 1/100 of an inch, with a precision of zero significant figures. Thus, the sample values are in the integer domain and it is not necessary to transform them.

In Table 2.10 we present relevant statistics of dataset Hail. The dataset consists of a single file, and each row in the table contains statistics concerning a different data type. The first column specifies the number of gaps, and the percentage over the 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.

Data Type	**#Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
Lat	0 (0.0)	2,570	4,932	3,845	3,854.7	475.7
Long	0 (0.0)	-12,442	-6,783	-9,676	-9,487.4	841.4
Size	0 (0.0)	100	600	100	136.2	51.8

Table 2.10: Statistics of dataset Hail for each data type. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

2.8 Dataset Tornado

Dataset Tornado [6] consists of tornadoes location coordinates from several stations around the US between 2015 and 2017. For each timestamp, the location hit by a tornado is specified via its latitude and longitude. 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.

In Table 2.11 we present relevant statistics of dataset Tornado. The dataset consists of a single file, and each row in the table contains statistics concerning a different data type. The first column specifies the number of gaps, and the percentage over the 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.

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: Statistics of dataset Tornado for each data type. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

2.9 Dataset Wind

Dataset Wind [6] consists of window velocity measurements from several stations around the US 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 several 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. The wind velocity is measured in mph, with a precision of one significant figure. Every sample value is transformed to the integer domain by multiplying it by 10.

In Table 2.12 we present relevant statistics of dataset Hail. The dataset consists of a single file, and each row in the table contains statistics concerning a different data type. The first column specifies the number of gaps, and the percentage over the 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.

Data Type	**#Gaps (%)**	**Min**	**Max**	**Mdn**	**Mean**	**SD**
Lat	0 (0.0)	0	4,899	3,776	3,778.4	460.8
Long	0 (0.0)	-12,432	-8	-8,732	-8,870.1	968.4
Speed	0 (0.0)	0	2,359	1,810	1,350.1	875.2

TABLE 2.12: Statistics of dataset Wind for each data type. The gaps are ignored when calculating the median, mean and standard deviation of the sample values.

2.10 Summary

In Figure 2.13 we summarize information regarding each of the eight datasets presented in the previous sections of this chapter. The second column indicates the characteristic of each dataset, in terms of the amount of gaps. The third column shows the number of files in each dataset. The last two columns show the number of data types in each dataset, and their names, respectively. In many datasets there are multiple data columns for a single data type. Additional details of every dataset are presented in the remaining sections of this chapter.

Dataset	**Dataset Characteristic**	**#Files**	**#Types**	**Data Types**
IRKIS [1, 2]	Many gaps	7	1	VWC
SST [3]	Many gaps	3	1	SST
ADCP [3]	Many gaps	3	1	Vel
ElNino [4]	Many gaps	1	7	Lat, Long, Zon.Wind, Mer.Wind, Humidity, Air Temp., SST
Solar [5]	Few gaps	4	3	GHI, DNI, DHI
Hail [6]	No gaps	1	3	Lat, Long, Size
Tornado [6]	No gaps	1	2	Lat, Long
Wind [6]	No gaps	1	3	Lat, Long, Speed

Table 2.13: Datasets overview. The second column indicates the characteristic of each dataset, in terms of the amount of gaps. The third column shows the number of files. The fourth and fifth columns show the number of data types and their names, respectively.

Chapter 3

Algorithms

In this chapter we present the coding algorithms implemented and evaluated in the project. In Section 3.1 we give an overview of the different algorithms, including their parameters and masking variants. In Section 3.2 we describe the general encoding scheme used for every algorithm, while in Section 3.3 we explain the approach taken to compress the gaps, using arithmetic coding with KT probability assignment. 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 and 3.6 we present algorithms PCA and APCA, respectively, which are the Constant model algorithms that we implemented. In Section 3.7 we describe the encoding scheme used by the Linear model algorithms, including the description of a routine used by every decoder. In sections 3.8, 3.9, 3.10, and 3.11, we present algorithms PWLH and PWLHInt, CA, SF, and FR, respectively, which are the Linear model algorithms that we implemented. In Section 3.12 we present GAMPS, a Correlation model algorithm. For every one of the algorithms, we show its coding routine, and describe an example that shows the encoding process step by step.

3.1 Introduction

The state-of-the-art algorithms used for sensor data compression reported in the literature [8, 9] assume, in general, that the signals have regular sampling and that there are no gaps in the data. However, this is often 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 signals.

We focus on algorithms that support near-lossless compression. Near-lossless compression guarantees a bounded per-sample error between the decompressed and the original signals. The error threshold can be specified by the user via a parameter, denoted ϵ. Observe that, when ϵ is equal to 0, 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. [8]. 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 complex nonlinear functions, but known algorithms that follow this technique do not support near-lossless compression and yield poor compression results [8].

For most algorithms we propose two variants, masking (*M*) and non-masking (*NM*), which differ in the encoding of the gaps in the data. The *M* variant of an algorithm first encodes the position of all the gaps, and then proceeds to encode the data values separately. On the other hand, the *NM* variant encodes the gaps and the data values simultaneously. 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 algorithm that supports both.

Most of the algorithms support a window size parameter, denoted w, which defines the size of the blocks into which the data are divided for encoding. In algorithm PCA, 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 model type, 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**	**Model**	***M***	***NM***	*w*
Base	x		Constant		x	
PCA [10]	x	x	Constant	x	x	x
APCA [11]	x	x	Constant	x	x	x
PWLH [12]/ PWLHInt	x	x	Linear	x	x	x
CA [13]	x	x	Linear	x	x	x
SF [14]	x	x	Linear	x		
FR [15]	x	x	Linear	x		x
GAMPS [16]	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 model type (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 every algorithm implemented in the project. 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 detail 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)
        ϵ: 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 number of rows and columns in the input file
5  Encode the timestamps column using a lossless code
6  if v == M then
7      Encode gap locations
8  end
9  if it is a Constant or Linear model algorithm then
10     Encode each signal column of the input file separately, using a coding routine for a
         specific algorithm (i.e. Base, PCA, APCA, PWLH, PWLHInt, CA, SF, FR)
11 else if it is a Correlation model algorithm then
12     Group the signal columns of the input file into disjoint subsets, then encode each
         subset independently, using a coding routine for a specific algorithm (i.e. GAMPS)
13 end
```

Figure 3.1: General encoding coding scheme for every algorithm.

The timestamps 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 5). This is done using a lossless code in which every integer is encoded independently, using a fixed number of bits. 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, in line 7; 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 (line 7 in Figure 3.1). We describe the position of the gaps by encoding a sequence of binary symbols, $x_1...x_n$, each symbol x_i indicating the presence (0) or absence (1) of a sample in the i-th timestamp in chronological order. To this end we use an arithmetic coder (AC) [17–19], which, sequentially provided with a probability assignment $p(x_i|x_1...x_{i-1})$, for each symbol x_i given the past symbols $x_1...x_{i-1}$, $1 \leq i \leq n$, generates a lossless encoding bit stream for $x_1...x_n$, of length $-\log P(x_1...x_n) + O(1)$, where $P(x_1...x_n) = \prod_{i=1}^{n} p(x_i|x_1...x_{i-1})$. This code length is optimal up to an additive constant.

For a sequence x of independent and identically distributed random binary symbols (with unknown probability distribution), the Krichevsky–Trofimov probability assignment [20], which we define next, yields an asymptotically optimal code length.

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

$$p(0|x_1...x_{i-1}) = \frac{n_0 + 1/2}{i}, \qquad p(1|x_1...x_{i-1}) = \frac{n_1 + 1/2}{i}, \tag{3.1}$$

where n_0 and n_1 denote the number of occurrences of 0 and 1 in $x_1...x_{i-1}$, 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 arbitrary 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 (3.1) 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.

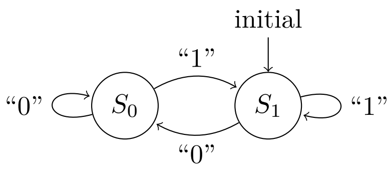

FIGURE 3.2: Markov process diagram

As we recall from Chapter 2, the positions of the gaps may follow different patterns for different datasets, but, in general, the gaps occur in bursts, and the amount of gaps is considerably less than the amount 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.

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 column entry of a csv data file. Since algorithm Base only supports the *NM* variant, 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 and using a fixed number of bits, which depends on the data type and the dataset. In practice, the number of bits used for encoding a data value ultimately depends on the range and accuracy of the sampling instrument used for acquiring and storing the data. 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 a (column-specific) fixed number of bits
8 end
```

FIGURE 3.3: Coding routine for algorithm Base.

3.5 Algorithm PCA

Algorithm PCA [10], also known as Piecewise Constant Approximation, is a Constant model algorithm that supports lossless and near-lossless compression. It has a window size parameter (w) that establishes a fixed block size in which the data are separately processed and encoded. For PCA we define both variants, *M* and *NM*.

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

```
input  : column: column of the csv data file to be encoded
         out: binary file encoded with algorithm 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 min and max be the minimum and maximum sample values in win, respectively
4      if $|max - min| \leq 2 * \epsilon$ then
5          Output bit 0 to out
6          $mid\_range = (min + max)/2$
7          Encode mid_range using a (column-specific) fixed number of bits
8      else
9          Output bit 1 to out
10         foreach sample in win, value, do
11             Encode value using a (column-specific) fixed number of bits
12         end
13     end
14 end
```

FIGURE 3.4: Coding routine for variant *M* of algorithm PCA.

A window can be encoded in two different ways. If the absolute difference between its maximum and minimum values is less than or equal to $2 * \epsilon$ (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 independently (lines 9-12).

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  : in: binary file coded with algorithm PCA
         out: decoded 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 a (column-specific) fixed number of bits
 7         Output size copies of mid_range to out
 8     else
 9         repeat size times
10             Decode value using a (column-specific) fixed number of 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 12 samples illustrated in Figure 3.6. Notice that the specific timestamp values are irrelevant for this algorithm. In this example we use algorithm PCA with an error threshold parameter (ϵ) equal to 1, and a fixed window size (w) equal to 4.

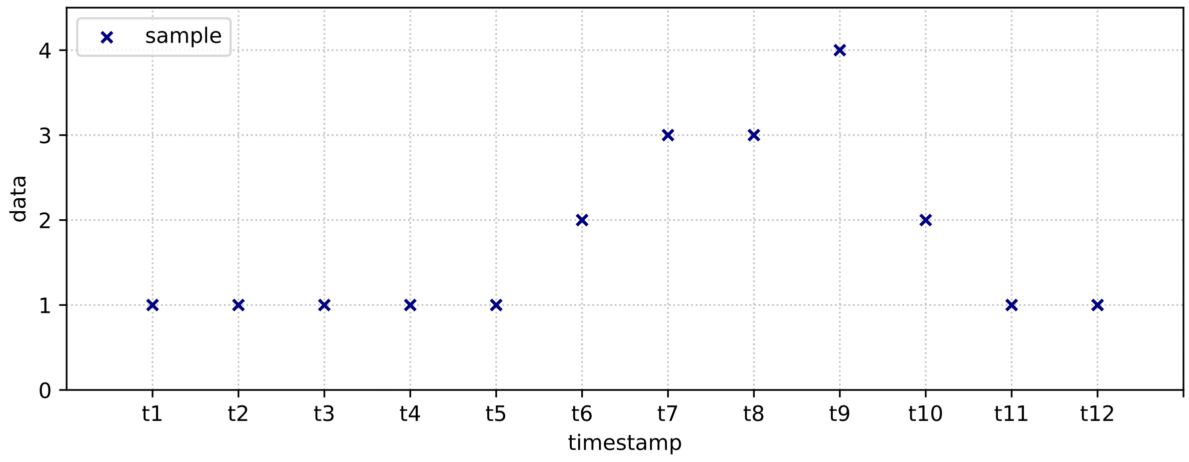

FIGURE 3.6

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.

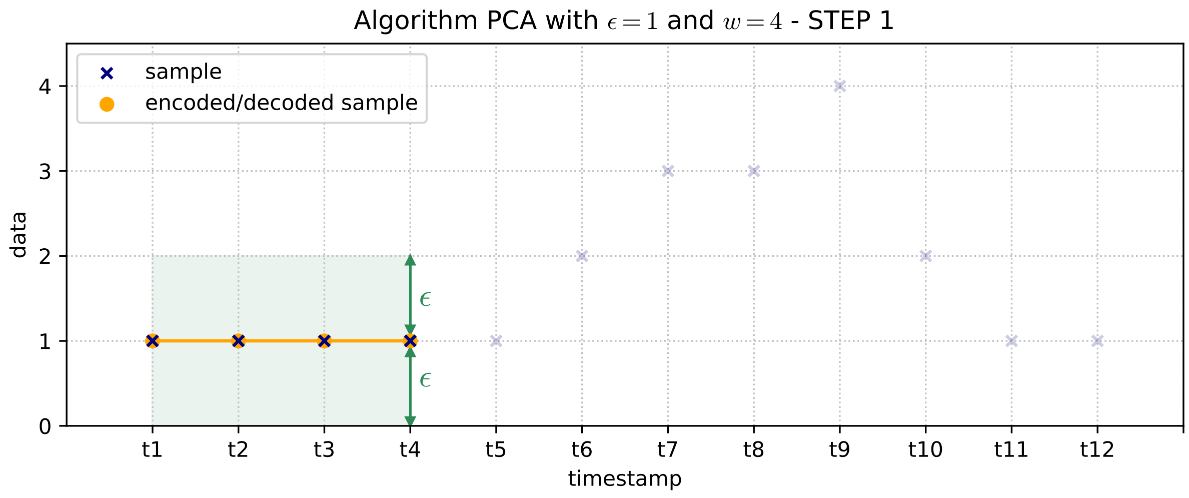

FIGURE 3.7

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 values are encoded as 2. This step is shown in Figure 3.8.

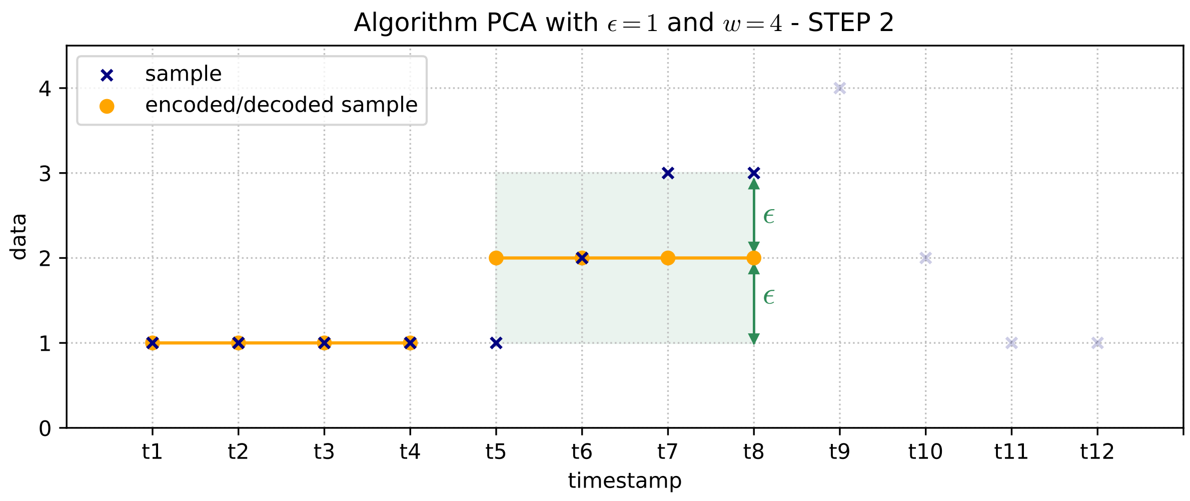

FIGURE 3.8

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 independently. This last step is shown in Figure 3.9.

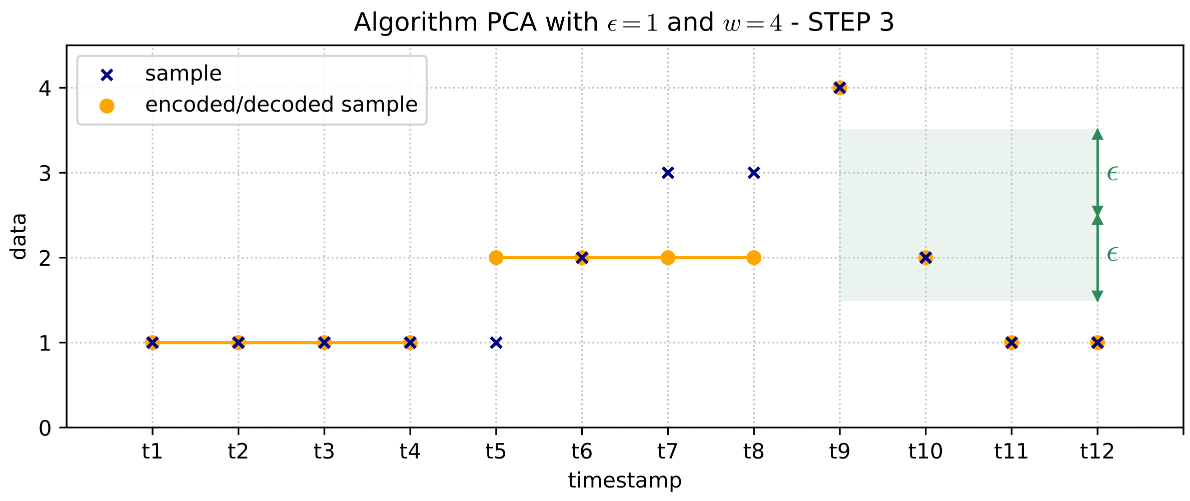

FIGURE 3.9

This simple example fairly represents every scenario that might 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 $1 +$ *column*.total_bits. On the other hand, since the threshold condition does not hold for the last window, it is encoded with $1 + w *$ *column*.total_bits 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 an integer representing a sample value, but also the character "N" representing 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 a (column-specific) fixed number of bits
6      else
7          Let min and max be the minimum and maximum sample values in win, resp.
8          if every entry in win is different from "N" and $|max - min| \leq 2 * \epsilon$ then
9              Output bit 0 to out
10             $mid\_range = (min + max)/2$
11             Encode mid_range using a (column-specific) fixed number of 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 a (column-specific) fixed number of 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 absolute difference between its maximum and minimum values is less than or equal to $2 * \epsilon$ (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 independently (lines 13-21), using *NO_DATA* for encoding gaps. Notice that in the first two cases the window is encoded with the same number of bits, i.e., $1 + column.\text{total\_bits}$, while in the last case the window is encoded with $1 + w * column.\text{total\_bits}$ 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 APCA [11], also known as Adaptive Piecewise Constant Approximation, 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 processed and 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 3-9). If the new entry makes the window violate the error threshold constraint (i.e. the absolute difference between its maximum and minimum values is greater than $2 * \epsilon$), or the window size greater than w, then the window is encoded, and a new empty window is created (lines 6-7). In any case, the current entry is added to *win* (line 9), and is eventually encoded. In particular, if the loop ends and *win* is not empty, it is encoded in line 12.

```
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 min and max be the minimum and maximum sample values in win, respectively
4      Let $m = \min\{min, entry\}$ and $M = \max\{max, entry\}$
5      if $(M - m) > 2 * \epsilon$ or $|win| == w$ then
6          EncodeWindow(win, out, w) // routine shown in Figure 3.12
7          Create an empty window, win
8      end
9      Add entry to win
10 end
11 if win is not empty then
12     EncodeWindow(win, out, w) // routine shown in Figure 3.12
13 end
```

FIGURE 3.11: Coding routine for variant *M* of algorithm APCA.

The routine called for encoding a window (lines 6 and 12), EncodeWindow, is shown in Figure 3.12. Observe that every window is encoded with the same number of bits, i.e. $\lceil \log_2 w \rceil +$ *column*.total_bits, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and *column*.total_bits 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 min and max be the minimum and maximum sample values in win, respectively
3  $mid\_range = (min + max)/2$
4  Encode mid_range using a (column-specific) fixed number of bits
```

FIGURE 3.12: EncodeWindow routine 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 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 2-3). 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 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 a (column-specific) fixed number of 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 12 samples illustrated in Figure 3.6. Notice that the specific timestamp values are irrelevant for this algorithm. In this example we use algorithm APCA with an error threshold parameter (ϵ) equal to 1, and a maximum window size (w) equal to 256.

The condition in line 5 of the coding routine evaluates to false in the first eight iterations, so those 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 *column*.total_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.

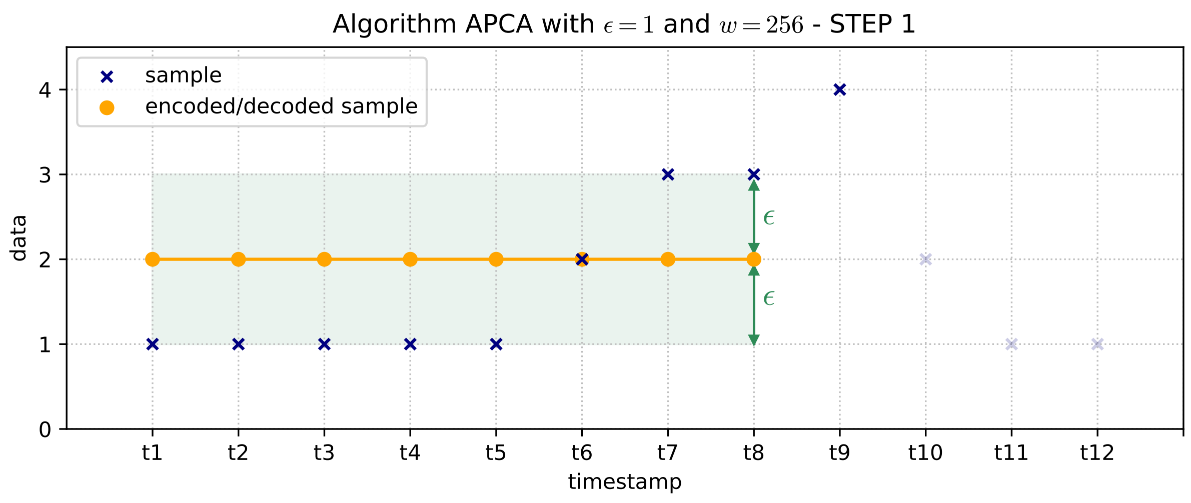

FIGURE 3.14

For the second window, the condition in line 5 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.

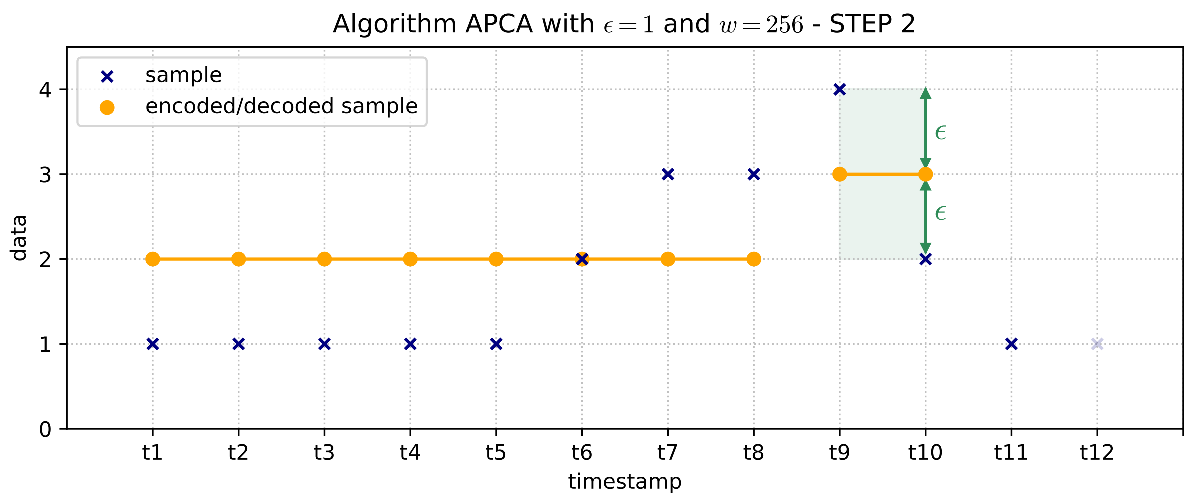

FIGURE 3.15

For the third window, the condition in line 5 evaluates to false in the 12th and last iteration. This window, which is equal to $[1, 1]$, is encoded after executing the last iteration, in line 12. In this case, the window size is 2 and its mid-range is 1. This last step is shown in Figure 3.16.

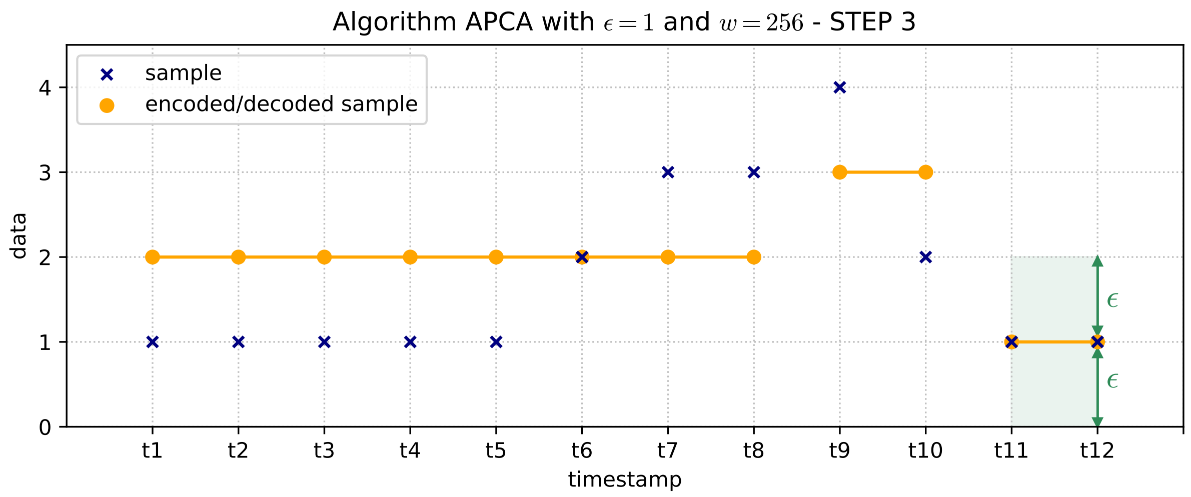

FIGURE 3.16

We point out that the three windows encoded in this example use exactly the same amount 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 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 would make it violate the error threshold constraint or the window size restriction (line 5 in Figure 3.11). In the coding routine for variant *NM*, a window must also be encoded if the newest 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. $\lceil \log_2 w \rceil$ + *column*.total_bits, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and *column*.total_bits 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 Linear model algorithms implemented and evaluated in the project. As we recall from Section 3.1, these type of algorithms approximate signals using linear, non-constant functions, so their implementations always require operating in the two-dimensional Euclidean space. Even though the encoding scheme varies between algorithms, it always involves encoding a sequence of line segments. Each line segment is encoded through its two endpoints, with their x-coordinates and y-coordinates corresponding to timestamps and sample values, respectively.

In Figure 3.17 we present the DecodeSegment routine, which is called by the decoding routine of every one of the Linear model algorithms. Its inputs are the timestamps column (recall that this column is encoded in line 5 of the routine shown in Figure 3.1), and a pair of timestamps and sample values, from which the coordinates of the encoded line segment's endpoints are obtained (line 1). The output is a list consisting of the sample values that are decoded from said segment, which the decoding routine must then write to the decoded csv data file.

```
input  : t_col: timestamps 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 when substituting the x-coordinate in the
      segment equation by $t_i$, and rounding the result to the nearest integer
5     Add $s_i$ to decoded_samples
6 end
```

FIGURE 3.17: DecodeSegment routine for Linear model algorithms.

We point out that the timestamps in the column are non-negative bounded integers (bounded in the sense that their range is known a priori by the coding routine, so they are always encoded using a fixed number of bits). For most algorithms, both the x-coordinates and the y-coordinates of the endpoints are encoded as bounded integers, the exceptions being algorithm PWLH, which encodes the y-coordinates as floats, and algorithm SF, which encodes the coordinates of both axes as floats.

Next, we present an example that details how the DecodeSegment routine works. The inputs are represented in Figure 3.18: the coordinates of the encoded segment's endpoints are $(t_5, 1)$ and $(t_9, 3.5)$, while the timestamps column is equal to $[t_1, ..., t_N]$, where $N \geq 9$. The segment defined in line 1 of the routine is colored red. After creating the empty list of decoded samples (line 2), a loop iterates over every timestamp t_i, $t_5 \leq t_i \leq t_9$, in the column, it decodes the corresponding sample value s_i (line 4), and adds it to the list (line 5). Given timestamp t_i, sample value s_i is obtained by taking the equation of the segment and substituting the x-coordinate by t_i, 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 DecodeSegment routine in this example.

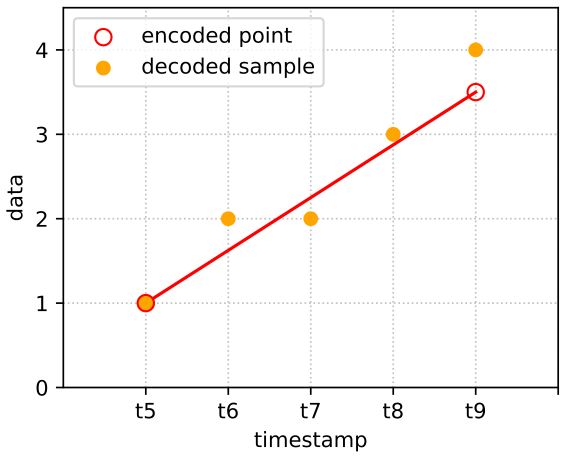

FIGURE 3.18: DecodeSegment routine example.

3.8 Algorithms PWLH and PWLHInt

Algorithm PWLH [12], also known as PieceWise Linear Histogram, 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 processed and encoded. For PWLH we define both variants, *M* and *NM*. Algorithm PWLHInt was implemented by introducing minor design changes to algorithm PWLH. Every comment in the current section pertains both algorithms, except for the specific differences that are pointed out in Subsection 3.8.2.

Since PWLH is a Linear model algorithm, from Section 3.7 we recall that its encoding process involves encoding a sequence of line segments. In Figure 3.19 we show the coding routine for variant *M*. It consists of a loop that iterates over all column entries, which are always integer values (the gaps are encoded separately). The algorithm maintains a window of consecutive points, *win*, which is initially empty (line 1). In each iteration, the addition of a new incoming point, *E*, to the window is considered (lines 3-10). The y-coordinate of the point is equal to the value of the column entry, while its x-coordinate is equal to the timestamp for said entry (lines 3-4). In line 5, a convex hull of the set that consists of every point in *win*, plus *E*, is obtained. If *E* makes the convex hull violate the error threshold constraint, or the window size is greater than w, then the window is encoded, and a new empty window is created (lines 8-9). In any case, *E* is added to *win* (line 11), and is eventually encoded. In particular, if the loop ends and *win* is not empty, it is encoded, whether it consists of a single (lines 14-15) or multiple values (line 17).

```
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
        $t\_col$: timestamps 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      Let $hull$ be the convex hull of the set that consists of every point in $win$, plus $E$
6      Let $valid\_hull$ be true iff there exists an edge in $hull$ for which the maximum
        distance from any of the points in $hull$ to said edge is less than or equal to $2 * \epsilon$
7      if not $valid\_hull$ or $|win| == w$ then
8          EncodeWindow($win$, $out$, $w$) // routine shown in Figure 3.20
9          Create an empty window, $win$
10     end
11     Add $E$ to $win$
12 end
13 if $|win| == 1$ then
14     Encode $|win|$ using $\lceil \log_2 w \rceil$ bits
15     Let $value$ be the y-coordinate of the single point in $win$
16     Encode $value$ using a (column-specific) fixed number of bits
17 else if $|win| > 1$ then
18     EncodeWindow($win$, $out$, $w$) // routine shown in Figure 3.20
19 end
```

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

The routine called for encoding a window (lines 8 and 17), EncodeWindow, is shown in Figure 3.20. The points in a window are modeled by a line segment that minimizes the mean square error for those points (line 1). For the operations in the two-dimensional Euclidean space, which involve calculating said segment, and computing the convex hull of the data points by applying Graham's Scan algorithm [21], we reuse part of the source code from the framework cited in [8]. Encoding a window involves encoding its size (line 3), together with the y-coordinates of both endpoints of the segment (lines 4-5). The window size is encoded using $\lceil \log_2 w \rceil$ bits, while each of the y-coordinates is encoded as a float, i.e. using 4 bytes, since this is the precision adopted in the method we reuse for calculating the segment. We point out that the values of the x-coordinates are encoded with the timestamp column (line 9 in Figure 3.1), so this routine must not encode them again.

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

FIGURE 3.20: EncodeWindow routine for algorithm PWLH.

The decoding routine for variant *M* is shown in Figure 3.21. 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 (line 5 in Figure 3.1), so this information is known by the decoding routine (input *t_col*). Each iteration of the loop starts with the decoding of the window size (line 3). If the window size is greater than 1, then the points in the window are modeled by a line segment, which means it was encoded via the EncodeWindow routine (recall Figure 3.20). In this case, the decoding routine decodes a pair of floats corresponding to the y-coordinates of the segment's endpoints (lines 5-6), obtains the timestamps corresponding to their x-coordinates (line 7), and calls the DecodeSegment routine with those inputs (line 8). As we recall from Figure 3.17, the DecodeSegment routine 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 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 csv data file
         w: maximum window size
         col_size: number of entries in the column
         t_col: timestamps 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 = DecodeSegment(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 a (column-specific) fixed number of bits
14         Output value to out
15     end
16     $n$ += size
17 end
```

FIGURE 3.21: Decoding routine for variant *M* of algorithm PWLH.

3.8.1 Example

Next we present an example of the encoding of 12 samples illustrated in Figure 3.6. Since PWLH is a Linear model algorithm, the specific timestamp values are required in the encoding routine (recall that the timestamps column is an input in Figure 3.19). We consider that the distance between any pair of adjacent timestamps is equal to 60. In this example we use algorithm PWLH with an error threshold parameter (ϵ) equal to 1, and a 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 violates the error threshold constraint in line 6 of the coding routine. In the first iteration, the window is empty, so the algorithm just adds the first sample point. Figure 3.22 shows this step in the graph. Observe that, besides the sample values, the convex hull for the current window is also shown.

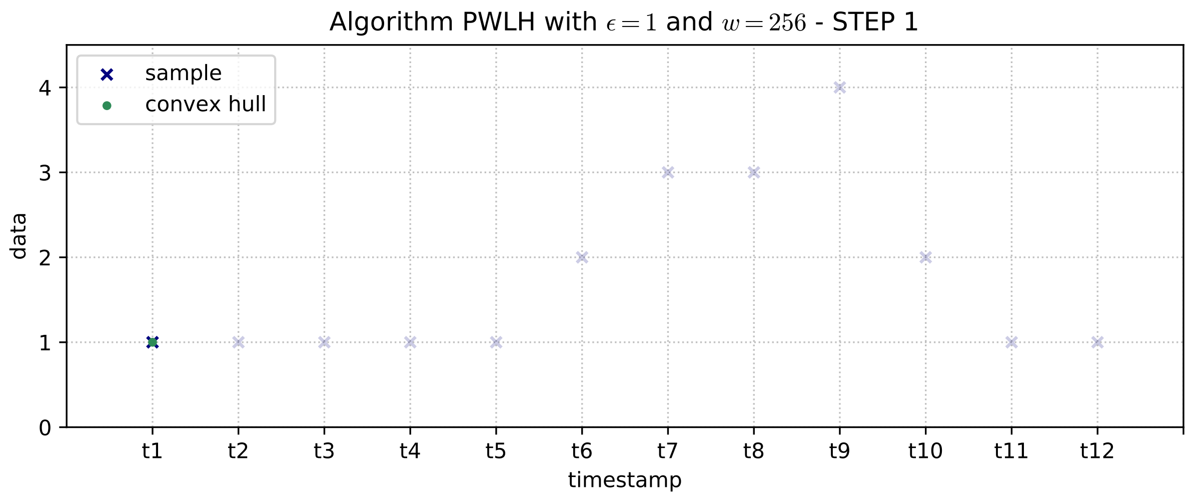

FIGURE 3.22

The convex hull that includes the second sample point consists of a single edge, and the maximum distance from either point to the edge is zero, i.e. $width = 0 \leq 2 * \epsilon = 2$, so the condition in line 6 is satisfied, and the second sample point is added to the window. This step is shown in Figure 3.23.

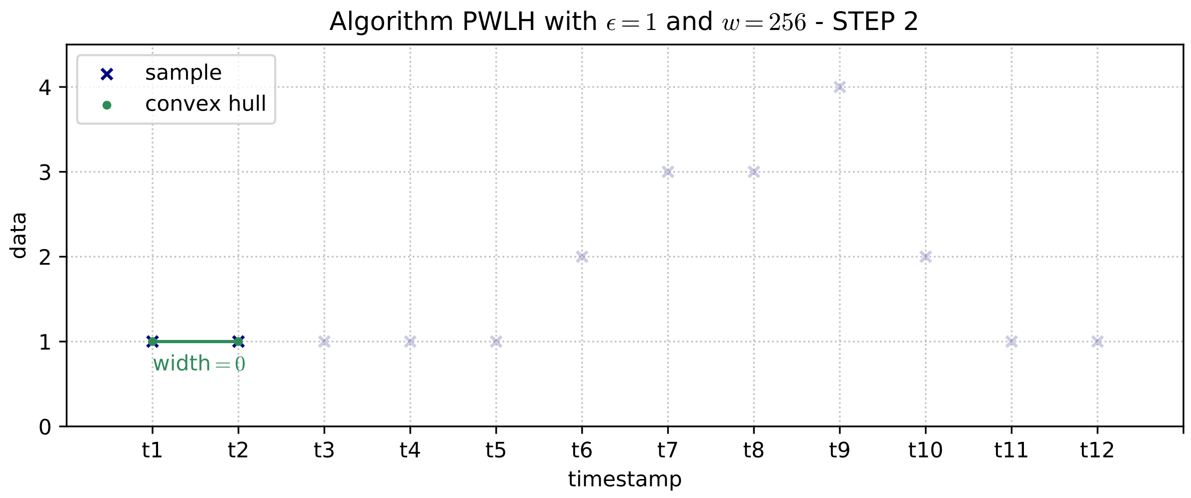

FIGURE 3.23

The sample values processed in the following three iterations are also equal to 1. The convex hull that includes those points still consists of a single edge, so *width* doesn't change and the threshold constrain is not violated in any case. Therefore, the three sample points are added to the window. This step is shown in Figure 3.24.

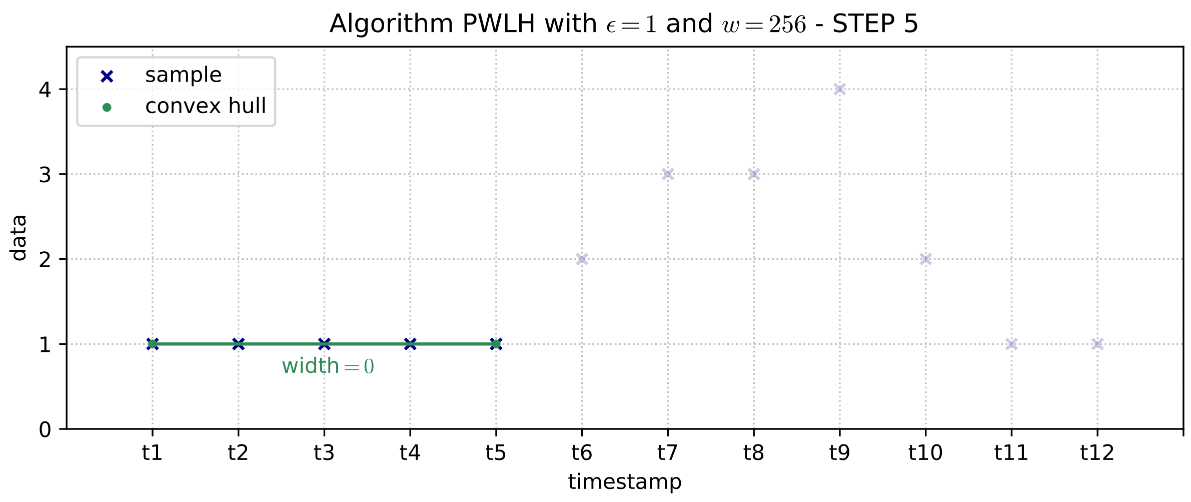

FIGURE 3.24

In the next iteration, sample value 2 is considered. The updated convex hull, which now consists of three edges, is shown in Figure 3.25. In this case, the maximum distance between its upper edge and any of its points is approximately 0.8. Therefore, $width \approx 0.8 \leq 2$, so the condition in line 6 is still satisfied, and the sample point is added to the window.

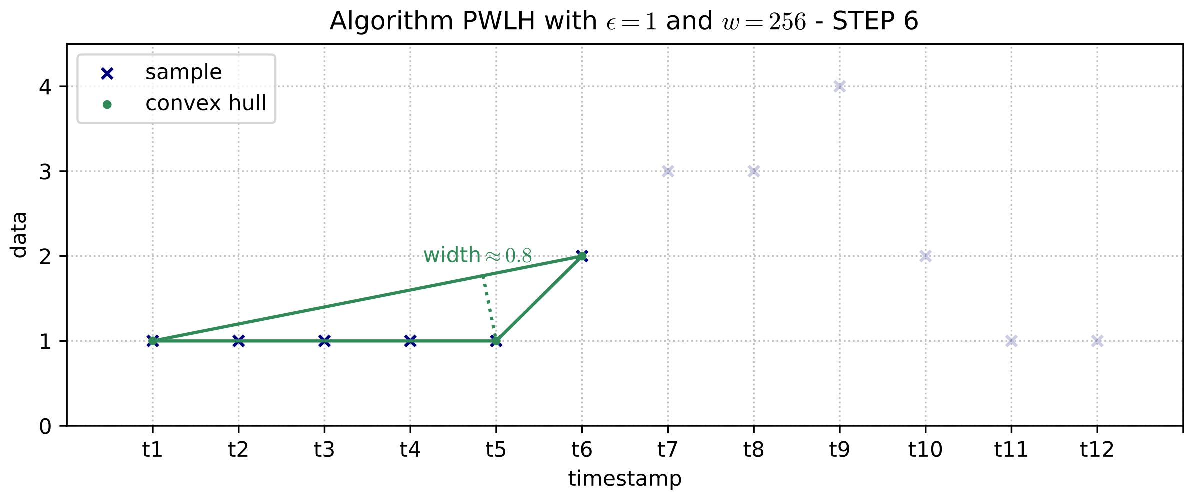

FIGURE 3.25

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 its upper edge and any of its points increases, it is never larger than 2, so the threshold constraint is never violated and the three sample points are added to the window. These steps are shown in figures 3.26, 3.27 and 3.28.

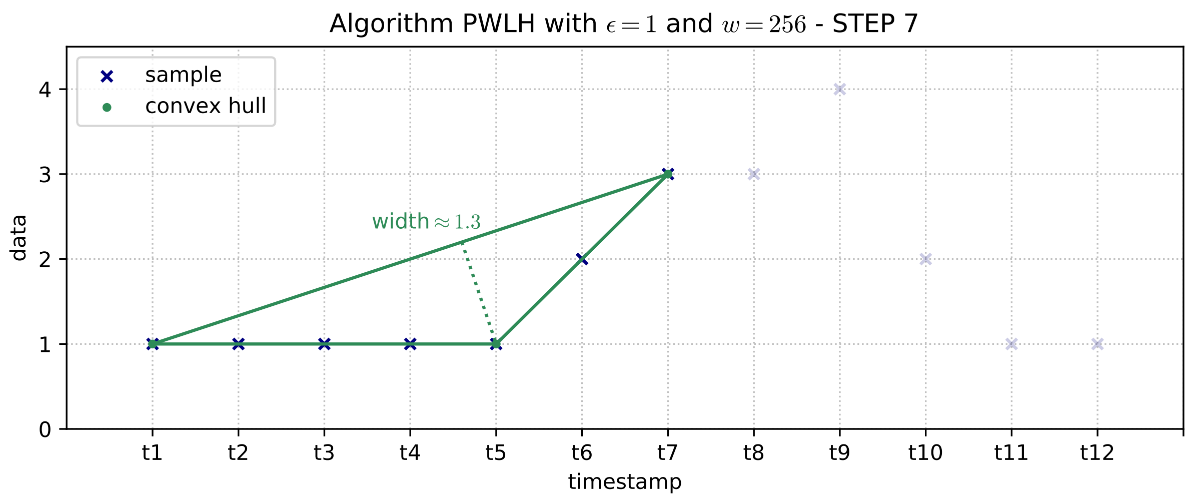

FIGURE 3.26

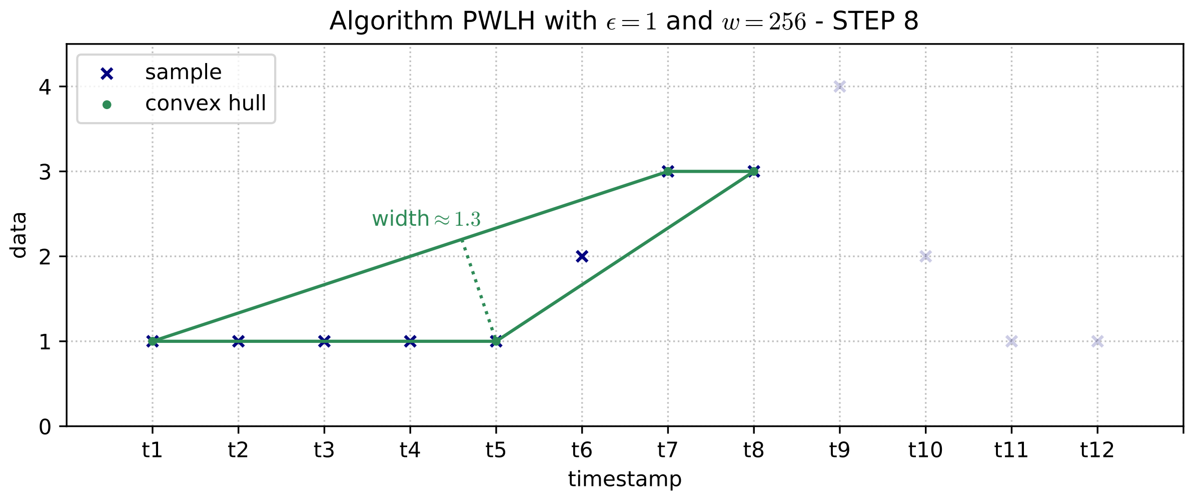

FIGURE 3.27

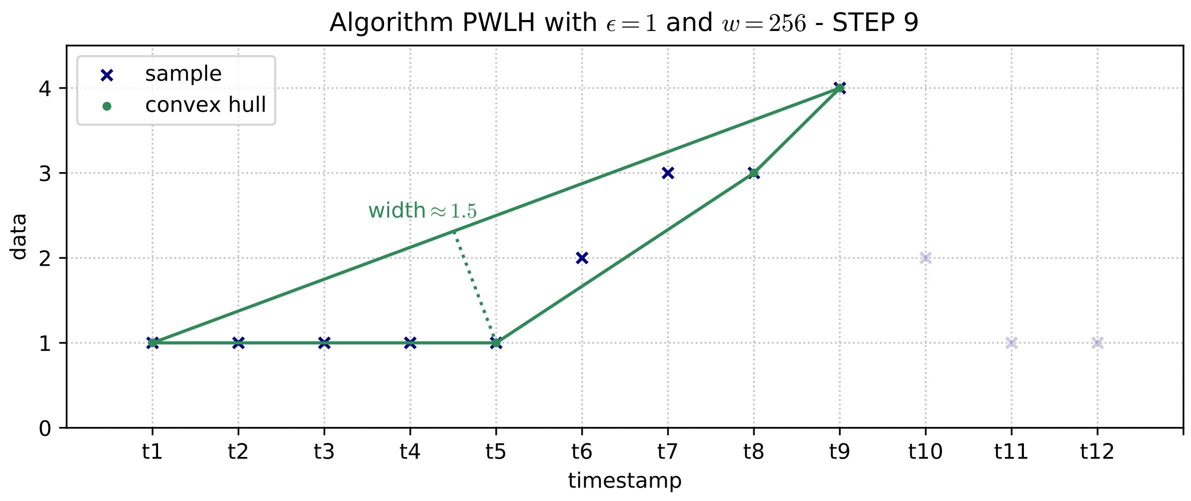

FIGURE 3.28

Eventually, in the 10th iteration, sample value 2 is considered. The updated convex hull, which is shown in Figure 3.29, violates the error threshold constraint in line 6 for the first time. Observe that, for every one of the four edges in the convex hull, there exists a point in the hull such that its distance to the edge is larger than 2.

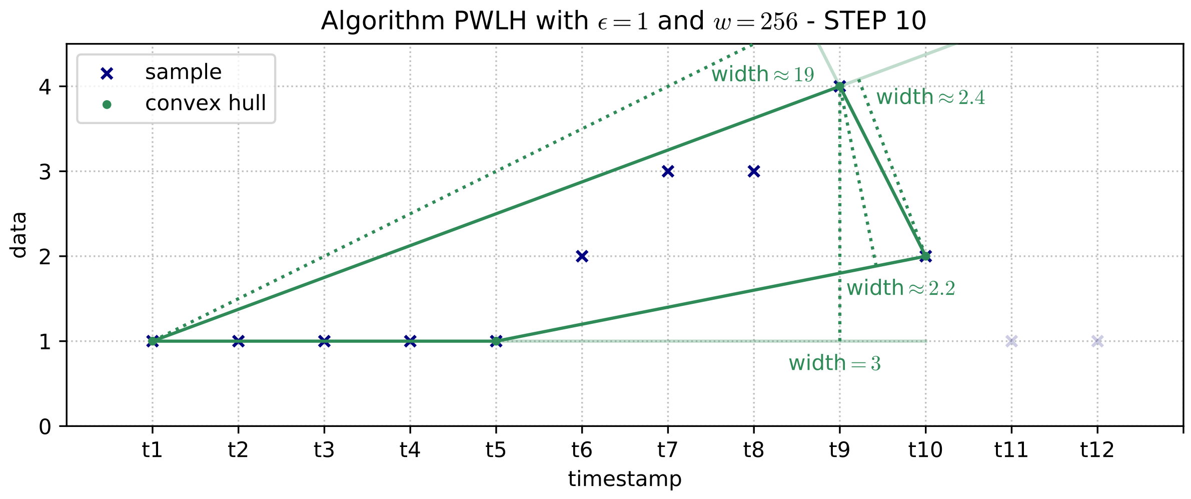

FIGURE 3.29

Since the condition in line 7 becomes true, the window is encoded via the EncodeWindow routine (line 8), and the point for sample value 2 is added to a new empty window (lines 9 and 11). The EncodeWindow routine (recall Figure 3.20) models the points in the window through the line segment that minimizes the mean square error for those points. This segment and its two endpoints are shown in Figure 3.30. 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.

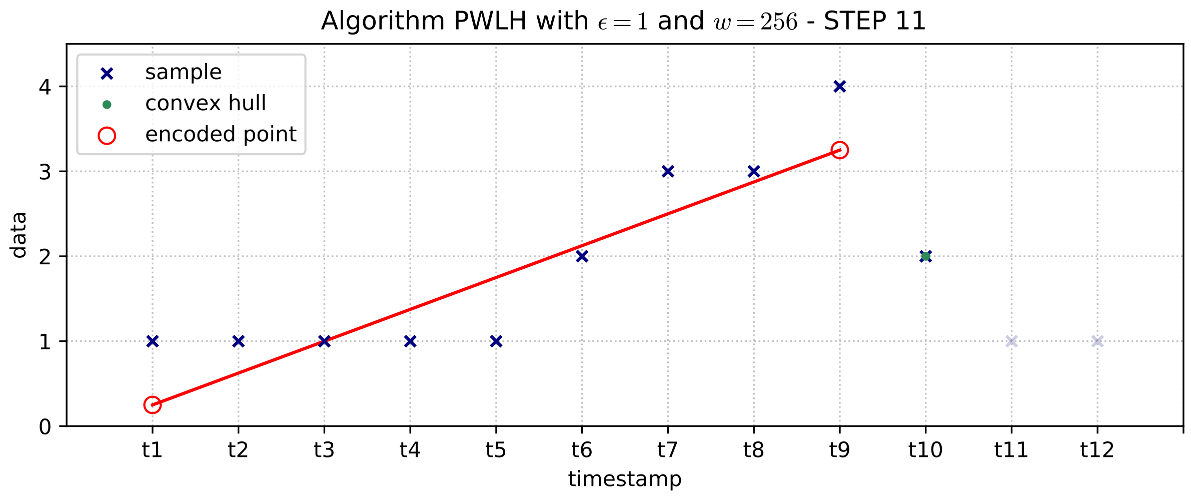

FIGURE 3.30

In the last two iterations of the coding routine, which correspond to the last two samples, the threshold constraint is not violated. Therefore, after executing the last iteration, *win* includes three points, so it is once again encoded via the EncodeWindow routine (line 17). The associated segment and its two endpoints are shown in Figure 3.31. In this figure we also display the values of the decoded samples, which are the values that the decoding routine, shown in Figure 3.21, would write to the decoded csv data file.

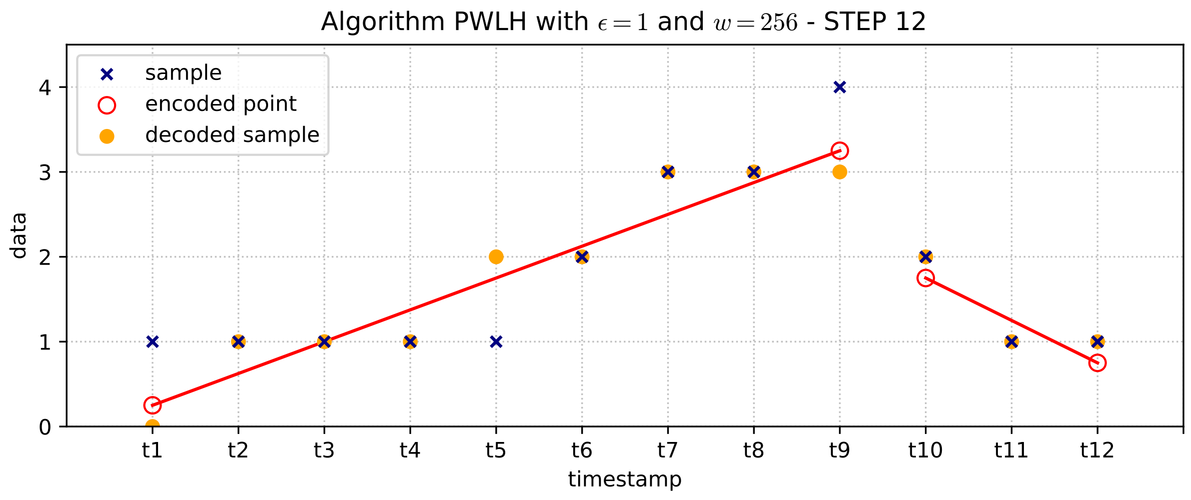

FIGURE 3.31

3.8.2 Differences between algorithms PWLH and PWLHInt

Recall, from Section 2.1, that the signals we are interested in compressing consist entirely of integer data samples. However, algorithm PWLH encodes the y-coordinates of a line segment's endpoints as floats, using 4 bytes (recall lines 4-5 in the EncodeWindow routine, shown in Figure 3.20). Since those y-coordinates correspond to the data samples domain, we realized that the compression performance of algorithm PWLH could be improved if they were instead encoded as bounded integers, using a fixed number of bits that depends on the data type and the dataset, which is the scheme adopted by most of the algorithms implemented in the project. That is precisely the idea behind the design of algorithm PWLHInt.

Algorithm PWLHInt was implemented by applying three changes to algorithm PWLH. First, we made the adjustment mentioned above, and changed lines 4-5 in the EncodeWindow routine, so that both y-coordinates are rounded to the nearest integer, then encoded using a (column-specific) fixed number of bits. Lines 5-6 were modified accordingly in the decoding routine (this is the only change required in the decoding routine). Secondly, we had to add a new constraint to the condition in line 7 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. Therefore, the coding routine of algorithm PWLHInt must operate with a lower maximum error threshold, i.e. $\epsilon' = \epsilon - 0.5$, to make sure that error threshold constraint holds, and the per-sample 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 implemented in the project, the experimental results suggest that the compression performance of algorithm PWLHInt is superior to that of algorithm PWLH. For instance, compare the plots in Figure 4.5, and the data in Table 4.4.

3.8.3 Non-masking (*NM*) variant

The coding and decoding routines for variant *NM* of algorithms PWLH and PWLHInt are similar to their 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 points or of gaps, but it cannot include both. Therefore, a new constraint is added to the condition in line 7 of the coding routine, shown in Figure 3.19, so that a window is also encoded if the newest entry is character "N" (gap in the data) and the other entries in the window are points, or vice versa. To encode a window that consists of gaps, algorithm PWLH uses $\lceil \log_2 w \rceil$ bits for encoding its size, and 4 bytes 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 *column*.total_bits bits for encoding the special integer *NO_DATA*.

3.9 Algorithm CA

Algorithm CA [13], also known as Critical Aperture, 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*.

In Figure 3.32 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 routine maintains a window of consecutive points, *win*, which is initially empty (line 1). The algorithm is based upon three points: archived (A) is the point most recently encoded (lines 6 and 15); snapshot (S) is the point most recently added to *win* (lines 8 and 17); incoming (E) is the point associated to the column entry for the current iteration (lines 3-4). The determination of whether a point must be encoded or not is based upon calculating slopes using parameter ϵ and these three points.

```
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: timestamps 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 EncodePoint(A, out, w) // routine shown in Figure 3.33
7      else if win is empty then
8          Make $S = E$, then add S 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 SE be the ray with initial point A that passes through point E
12         if not slope(SMin) $\leq$ slope(SE) $\leq$ slope(SMax) or |win| == w then
13             EncodeWindow(win, out, w) // routine shown in Figure 3.34
14             Create an empty window, win
15             Make $A = E$, then EncodePoint(A, out, w) // routine shown in Figure 3.33
16         else
17             Make $S = E$, then add S 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     EncodeWindow(win, out, w) // routine shown in Figure 3.34
26 end
```

FIGURE 3.32: Coding routine for variant *M* of algorithm CA.

The condition in line 5 is only satisfied in the first iteration. In that case, point E is saved as A, and encoded via the EncodePoint routine, which is shown in Figure 3.33. The condition in line 7 is satisfied when *win* is empty. This only occurs if point A was encoded in the previous iteration. In that case, point E is saved as S, and added to the window, and two rays, *SMin* and *SMax*, are defined from parameter ϵ, and points A and S.

If both conditions (in lines 5 and 7) evaluate to false, then, another ray, *SE*, is defined from points A and E (line 11). If *SE* is not between *SMin* and *SMax*, then point E must not be added to *win*, because doing so would violate the error threshold constraint. It also must not be added if the window has reached the maximum size allowed (w). In any of those cases, the window is encoded via the EncodeWindow routine, which is shown in Figure 3.34, a new one is created, and point E is saved as A, and encoded (lines 13-15). On the other hand, if the condition in line 12 is not satisfied, then point E is saved as S, and added to the window (line 17), and *SMin* and *SMax* are updated considering the new point S (lines 18-20).

```
input  : $P$: point to encode
         $out$: binary file encoded with algorithm CA
         $w$: maximum window size
1 Encode 1 using $\lceil \log_2 w \rceil$ bits
2 Let $value$ be the y-coordinate of point $P$
3 Encode $value$ using a (column-specific) fixed number of bits
```

Figure 3.33: EncodePoint routine 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 y-coordinate of the last point in $win$
3 Encode $value$ using a (column-specific) fixed number of bits
```

Figure 3.34: EncodeWindow routine for algorithm CA.

The decoding routine for variant M is shown in Figure 3.35. 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 timestamps column (line 5). If *size* is equal to 1, *value* was encoded by the EncodePoint routine (recall Figure 3.33), so it corresponds to the y-coordinate of an archived point (A). In this case, *value* is written to the decoded csv data file and point A is saved (lines 7-8). On the other hand, if *size* is greater than one, *value* was encoded by the EncodeWindow routine (recall Figure 3.34), so it corresponds to the y-coordinate of the last snapshot point (S) added to the window. In this case, the points in the window are modeled by a line segment, whose endpoints are A and S, so the DecodeSegment routine is called with their coordinates as inputs (line 11). The DecodeSegment routine (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 csv data file
         w: maximum window size
         col_size: number of entries in the column
         t_col: timestamps column
1  $n = 0$
2  while $n < col\_size$ do
3      Decode size using $\lceil \log_2 w \rceil$ bits
4      Decode value using a (column-specific) fixed number of 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 = DecodeSegment(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 3.35: Decoding routine for variant M of algorithm CA.

3.9.1 Example

Next we present an example of the encoding of 12 samples illustrated in Figure 3.6. Recall from Example 3.8.1, that since CA is a Linear model algorithm, the specific timestamp values are required in the encoding routine, and we consider that the distance between any pair of adjacent timestamps is equal to 60. In this example we use algorithm CA with an error threshold parameter (ϵ) equal to 1, and a maximum window size (w) equal to 256.

This example involves encoding 12 samples, so no window can reach the maximum size (256). Therefore, a window is only encoded when an incoming point (E) violates the error threshold constraint in line 12 of the coding routine.

In the first iteration, the condition in line 5 is satisfied, so the first incoming point is saved as the archived point (A), and it is encoded via the EncodePoint routine, using $\lceil \log_2 w \rceil = \log_2 256 = 8$ bits for encoding a 1, and *column*.total_bits bits for encoding its y-coordinate, also 1. In the second iteration, *win* is empty, so the condition in line 7 is satisfied. Therefore, the second incoming point is saved as the snapshot point (S), it is added to *win*, and two rays, *SMin* and *SMax*, are defined. Figure 3.36 shows both saved points, A and S, as well as both rays, *SMin* and *SMax*, after the second iteration is completed.

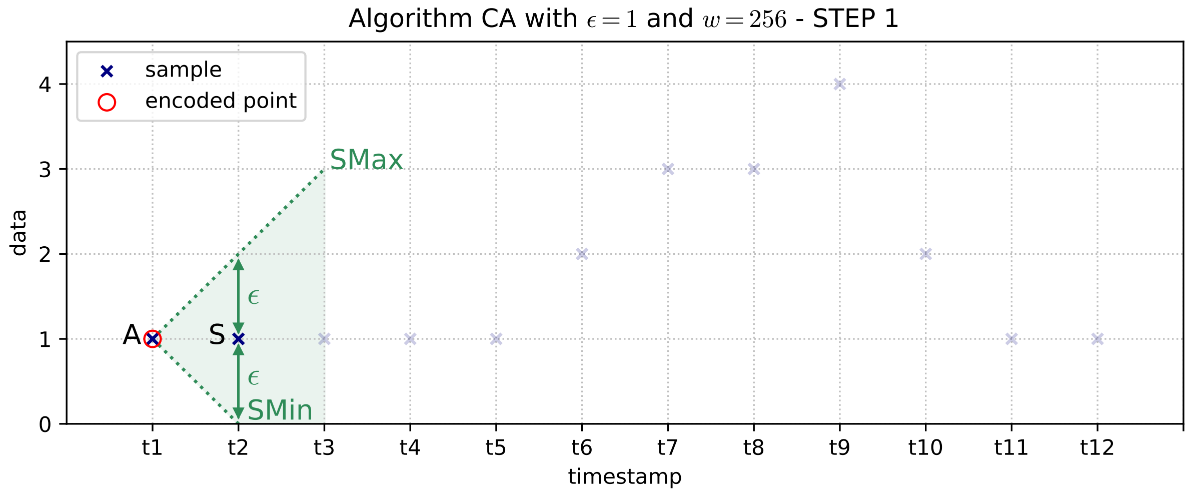

FIGURE 3.36

In the third iteration, another sample value equal to 1 is processed. In Figure 3.37, both *E3*, the incoming point, and *SE3*, which is the ray with initial point A that passes through *E3*, are shown.

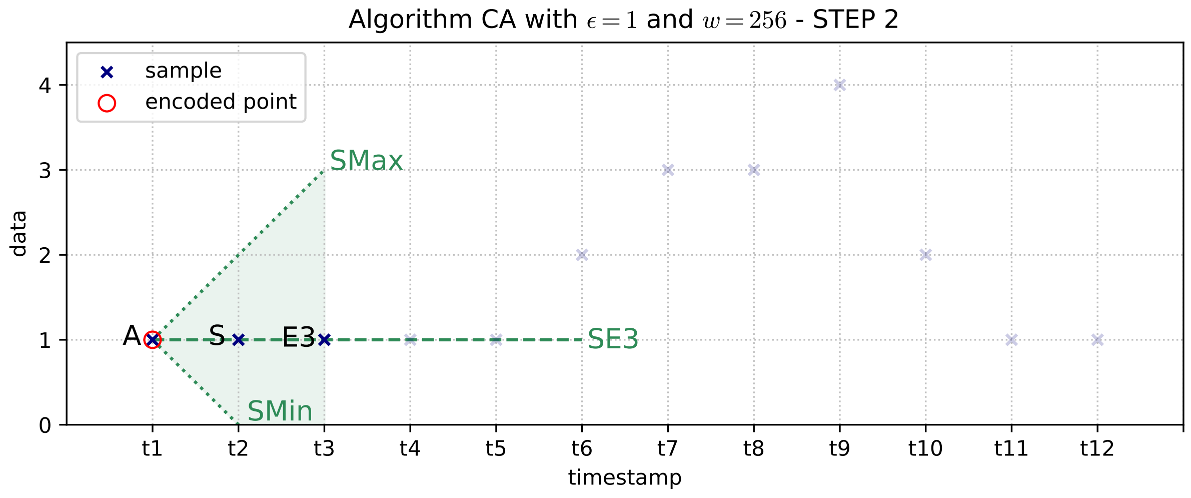

FIGURE 3.37

SE3 is between *SMin* and *SMax*, so the threshold constraint is not violated, and the condition in line 12 evaluates to false. Therefore, *E3* is saved as *S*, and added to *win*, and *SMin* and *SMax* are updated. Figure 3.38 shows the information after the third iteration is completed.

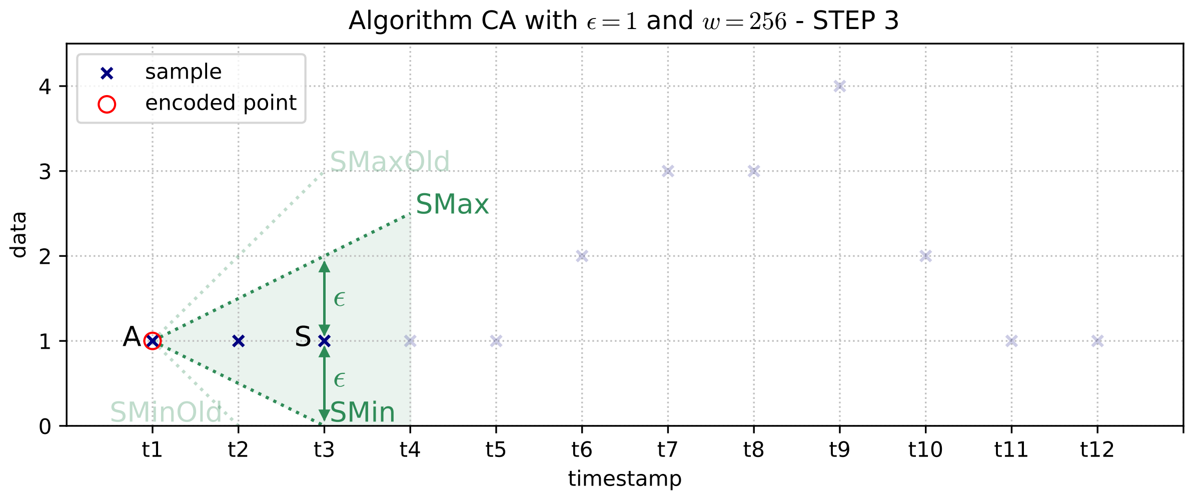

FIGURE 3.38

The following two iterations are similar to the last one. In each iteration, the incoming points are added to the window, and both rays, *SMin* and *SMax*, are updated. Figure 3.39 shows the information after the 5th iteration is completed.

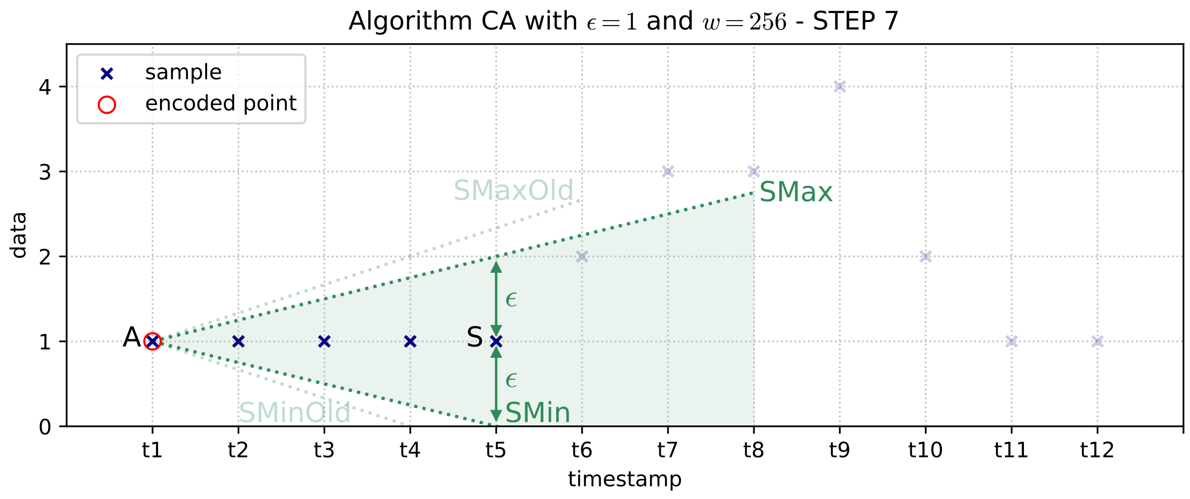

FIGURE 3.39

In the 6th iteration, sample value 2 is processed. In Figure 3.40, both incoming point *E6*, and ray *SE6* are shown.

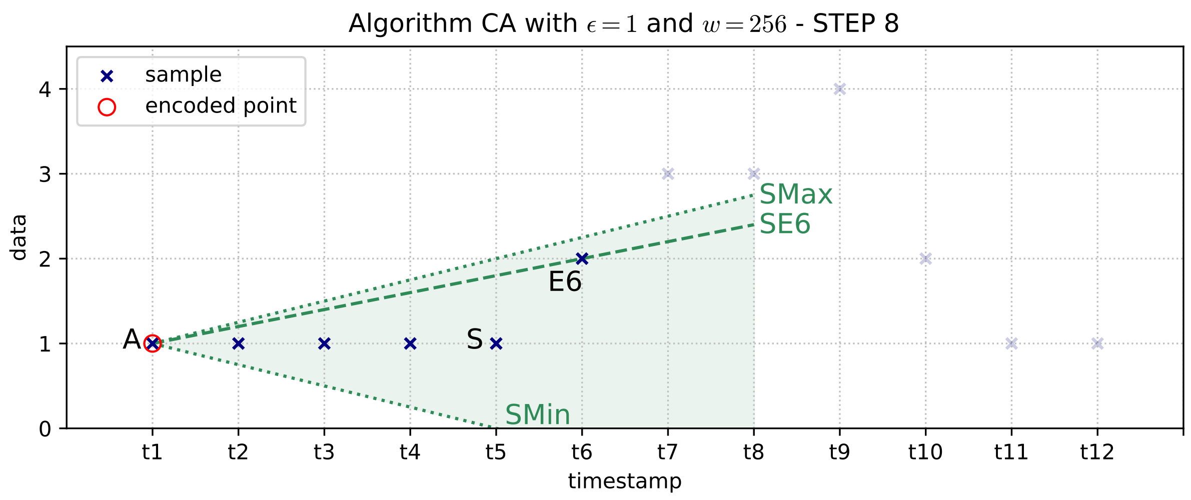

FIGURE 3.40

SE6 is between *SMin* and *SMax*, so the condition in line 12 evaluates to false. Therefore, *E6* is saved as *S*, and added to *win*. In this case, *SMin* is updated, while *SMax* remains unchanged. Figure 3.41 shows the information after the 6th iteration is completed.

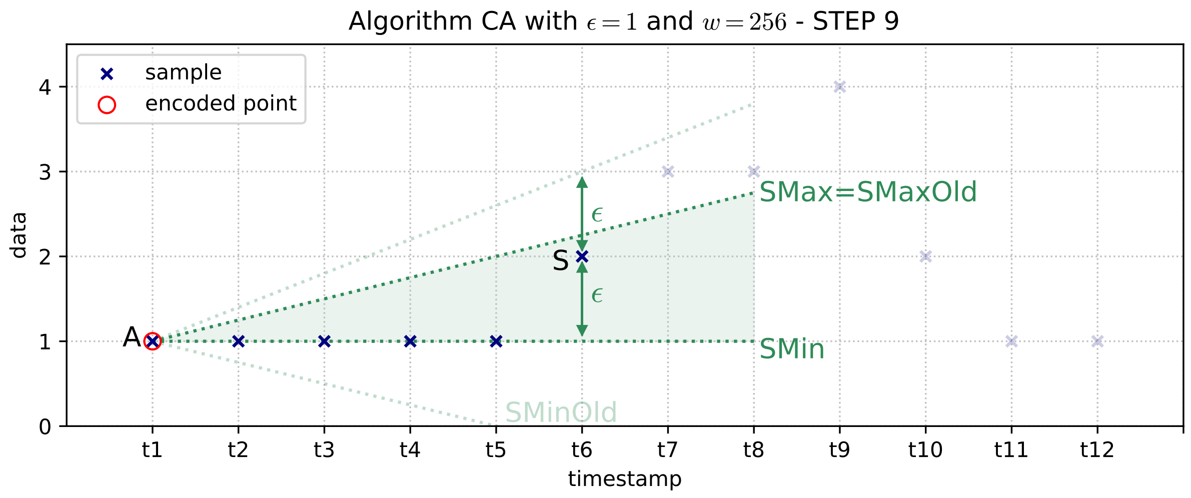

FIGURE 3.41

In the 7th iteration, sample value 3 is processed. In Figure 3.42, both incoming point *E7*, and ray *SE7* are shown.

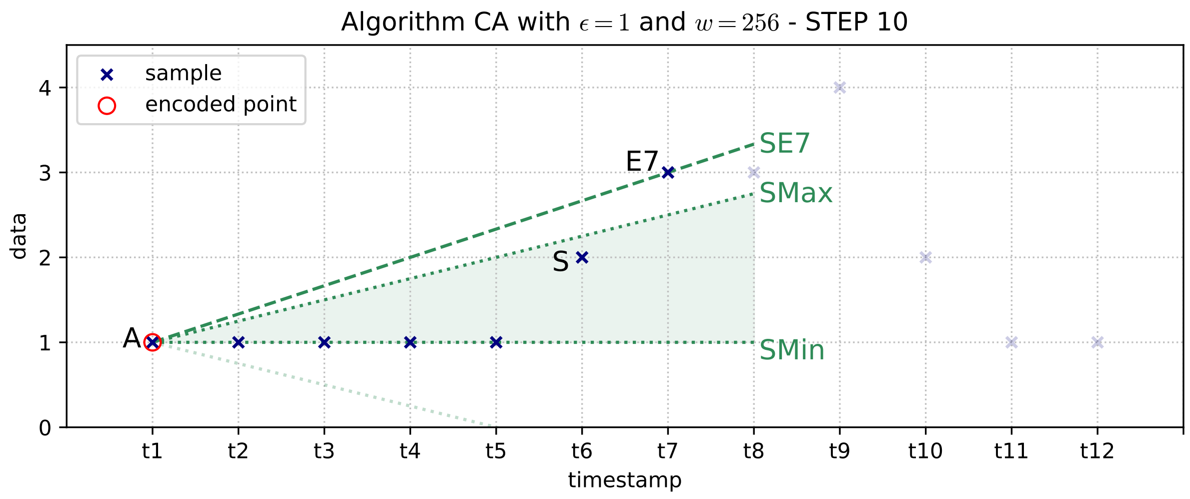

FIGURE 3.42

For the first time, the threshold constraint is violated, since *SE7* is greater than *SMax*, so the condition in line 12 becomes true. *win* is encoded via the EncodeWindow routine, using $\lceil \log_2 w \rceil = 8$ bits for encoding its size (i.e. 5), and *column*.total_bits bits for encoding the y-coordinate of its last point (i.e. 2). Also, a new window is created, and *E7* is saved as *A*, and encoded via the EncodePoint routine, using $\lceil \log_2 w \rceil = 8$ bits for encoding a 1, and *column*.total_bits bits for encoding its y-coordinate, 3. In the following iteration, *win* is empty, so the condition in line 7 is satisfied. Therefore, incoming point *E8* is saved as *S*, it is added to *win*, and *SMin* and *SMax*are defined once more. Figure 3.43 shows the information after the 8th iteration is completed.

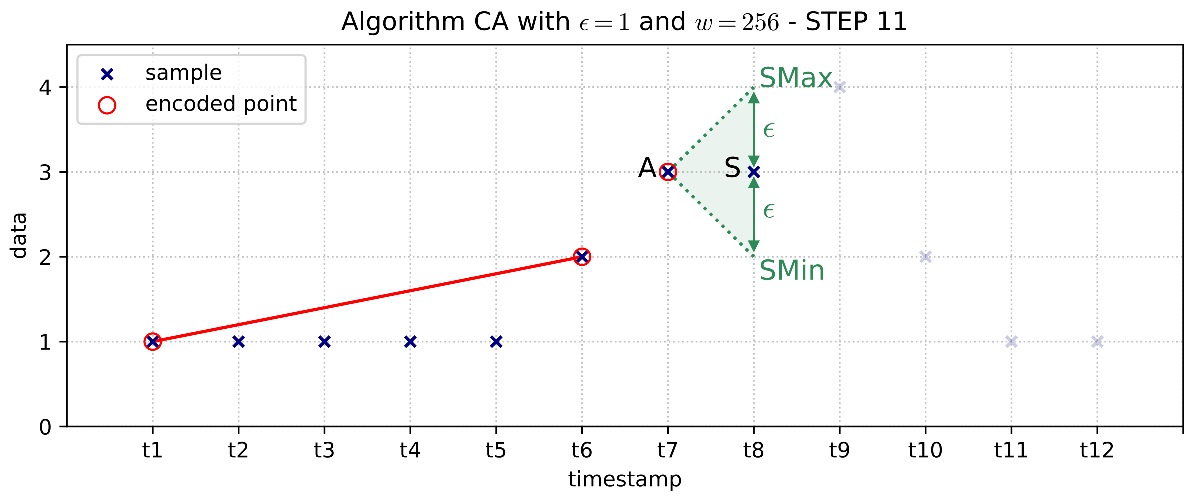

FIGURE 3.43

In Figure 3.44 we present the information after the coding routine has finished. After the last iteration in the loop is executed, the last window is not empty, so it is encoded calling the EncodeWindow routine in line 25. Besides showing the encoded points and their associated line segments, in Figure 3.44 we also display the values of the decoded samples, which are the values that the decoding routine, shown in Figure 3.35, would write to the decoded csv data file. We point out that, since the algorithm must support the scenario in which more than one sample exists for a single timestamp, the archived point must be encoded independently from the window. Otherwise, the algorithm could consider that the archived value was the last point of the most recently encoded window.

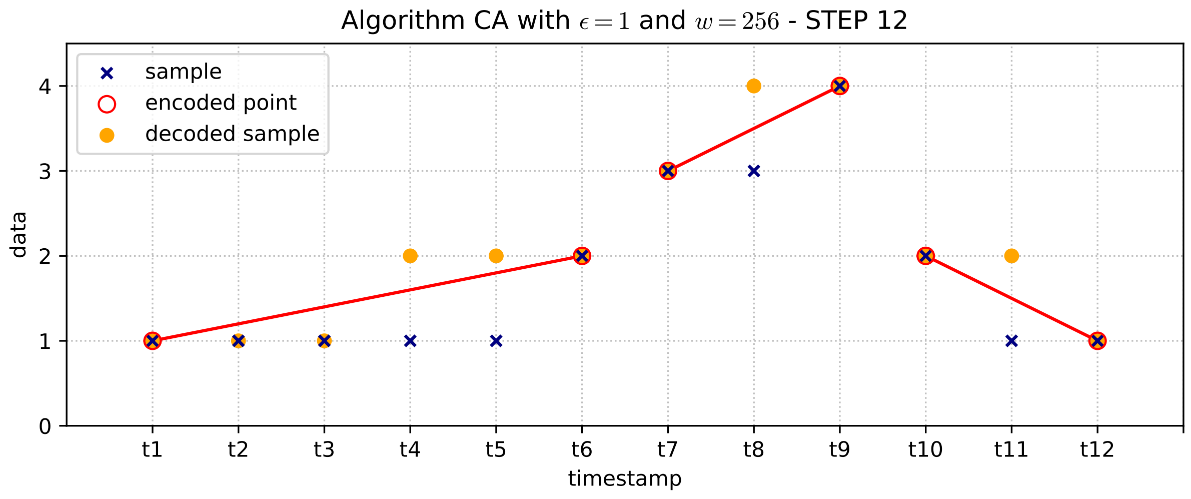

FIGURE 3.44

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 adding a new point would make it violate the error threshold constraint or the window size restriction (line 12 in Figure 3.32). The coding routine for variant *NM* must also encode a window if the newest column entry is character "N" (gap in the data) and the other entries in the window are sample points, or vice versa. We point out that, in the first scenario, line 15 is not executed, and neither is line 6 executed when "N" is the first column entry. A window that consists of gaps is encoded with the same number of bits as a window that consists of sample points, i.e. $\lceil \log_2 w \rceil$ + *column*.total_bits, where $\lceil \log_2 w \rceil$ bits are used for encoding its size, and *column*.total_bits bits are used for encoding the special integer *NO_DATA*.

3.10 Algorithm SF

Algorithm SF [14], also known as Slide Filter, 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 it is a Linear model algorithm, the encoding process of algorithm SF involves encoding a sequence of line segments. However, unlike the previously presented algorithms, SF accepts the mixture of connected and disconnected line segments, which can reduce the number of encoded data points, leading to better compression results. In Figure 3.45 we show the coding routine. It consists of a loop that iterates over all column entries, which are always integer values (the gaps are encoded separately). The algorithm maintains a window of consecutive points, *win*, which is initially empty (line 1). In each iteration, the addition of an incoming point, *E*, to the window is considered (lines 3-34). The y-coordinate of the point is equal to the value of the column entry, while its x-coordinate is equal to the timestamp for said entry (lines 3-4). *win* can only be empty in the first iteration, in which case *E* is added to *win* (line 5).

The process to establish whether the incoming point must be added to the window or not, is similar to the process described in Section 3.9 for algorithm CA, and is based upon calculating slopes using parameter ϵ and the points already added to the window. If *win* has a single point, *P*, then two rays, *SMin* and *SMax*, are defined from parameter ϵ, and points *P* and *E* (lines 8-10); next, *E* is added to *win* (line 11). Segment *SE* (lines 13-14) being included in set *LS* (line 15) means that the threshold condition is valid for point *E*. If that is the case, and the window has not reached the maximum size allowed (w), then condition in line 16 is satisfied, so *SMin* and *SMax* are updated, and *E* is added to *win* (lines 17-18). Like it occurs in algorithm CA, as new points are processed, the slope of *SMin* increases, and the slope of *SMax* decreases. However, the process for updating those rays is a bit more complex in algorithm SF, since the intersection of the rays is not fixed, and all the points in *win* must be considered each time [14]. To carry out this process, we reuse part of the source code from the framework cited in [8].

If the condition in line 16 of the coding routine evaluates to false, then, depending on the case, either the EncodePoint routine or the EncodeWindow routine are called. The EncodePoint routine, shown in Figure 3.46, outputs a bit that indicates whether the encoded point belongs to a connected or disconnected line segment (line 1), and the x and y-coordinates of said point are encoded as floats, i.e. using 4 bytes (lines 2-3). We point out that this is the only algorithm presented in our project for which the x-coordinate of an encoded point might not coincide with a timestamp (integer) value. This is the reason why it must be always encoded as a float. The EncodeWindow routine, shown in Figure 3.47, obtains the segment (from the input set *LS*) which minimizes the mean square error for the points in the input window (line 1), then encodes its first endpoint via the EncodePoint routine, passing the connected flag as true (lines 2-3). Finally, the segment is returned (line 4).

Back to the coding routine, shown in Figure 3.45, if no window has been encoded, then the EncodeWindow routine is called (line 21), the generated segment is saved as $segment_{prev}$ (line 33), and a new window that includes point *E* is created (line 34). Otherwise, if a window has already been encoded in a previous iteration, that implies that a segment has already been generated and saved as $segment_{prev}$, so the algorithm looks for a new segment that can approximate the data points in *win* without violating the threshold constraint, and which can be connected to $segment_{prev}$ (line 23). If said segment is found, then the intersection point between the connected segments is encoded via the EncodePoint routine, passing the connected flag as true (lines 25-26). Otherwise, the last point of $segment_{prev}$ is encoded via the EncodePoint routine, in this case passing the connected flag as false (lines 28-29), and the EncodeWindow routine is called (line 30), which generates a new segment and encodes its first endpoint. Notice that, if the connected segment is found, only one point is encoded, while two points are encoded if that is not the case. The connected segment might not be the best fitted segment to encode the data points in the window, but, unlike algorithm PWLH, algorithm SF prioritizes reducing the number of encoded points, through the use of connected segments, over finding unconnected segments which minimize the mean square error for the data points in each window. Depending on the case, in line 33 the segment defined in line 23 or in line 30 is saved, and a new window that includes point E is created (line 34)

```
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: timestamps 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 win is empty then
6          Add $E$ to win, then continue
7      else if $|win| == 1$ then
8          Let $P$ be the single point in win
9          Let SMin be the ray with initial point $(P.x, P.y + \epsilon)$, through point $(E.x,E.y - \epsilon)$
10         Let SMax be the ray with initial point $(P.x, P.y - \epsilon)$, through point $(E.x,E.y + \epsilon)$
11         Add $E$ to win, then continue
12     end
13     Let $S$ be the point of intersection of SMin and SMax
14     Let SE be a segment with endpoint $S$ that passes through point $E$
15     Let LS be the set that includes every line segment $s$ that passes through point $S$ and
         satisfies the following condition: slope(SMin) $\leq$ slope($s$) $\leq$ slope(SMax)
16     if $SE \in LS$ and $|win| < w$ then
17         Update SMin and SMax
18         Add $E$ to win, then continue
19     end
20     if no window has been encoded then
21         segment = EncodeWindow(win, LS) // routine shown in Figure 3.47
22     else
23         Let $segment \in LS$ be the line segment that approximates the data points in win,
             and can be connected to $segment_{prev}$
24         if segment exists then
25             Let $P$ be the point of intersection of $segment_{prev}$ and segment
26             EncodePoint(P, out, true) // routine shown in Figure 3.46
27         else
28             Let $P$ be the last endpoint of $segment_{prev}$
29             EncodePoint(P, out, false) // routine shown in Figure 3.46
30             segment = EncodeWindow(win, LS) // routine shown in Figure 3.47
31         end
32     end
33     Save segment as $segment_{prev}$
34     Create an empty window, win, then add $E$ to win
35 end
```

FIGURE 3.45: Coding routine for variant *M* of algorithm SF.

```
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 3.46: EncodePoint routine for algorithm SF.

```
input  : $win$: window to encode
         $LS$: set of line segments
output: $segment$: line segment that approximates all the points in $win$
1 Let $segment \in LS$ be the line segment that minimizes the MSE for the data points 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 3.46
4 return $segment$
```

FIGURE 3.47: EncodeWindow routine for algorithm SF.

The decoding routine is shown in Figure 3.48. We point out that this is the only decoding routine presented in our project 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. 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.

```
input  : $in$: binary file coded with algorithm SF
         $out$: decoded csv data file
         $col\_size$: number of entries in the column
         $t\_col$: timestamps column
1  $n = 0$
2  while $n < col\_size$ do
3      $P$, $connected$ = DecodePoint() // routine shown in Figure 3.49
4      if $n == 0$ then
5          Let $P_o = P$
6          continue
7      end
8      Let $P_f = P$
9      $samples$ = DecodeSegment($t\_col$, $P_o.x$, $P_f.x$, $P_o.y$, $P_f.y$) // 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 $P_o = P_f$
15     else
16         $P_o$, $connected$ = DecodePoint() // routine shown in Figure 3.49
17     end
18     $n$ += $|samples|$
19 end
```

FIGURE 3.48: Decoding routine for variant M of algorithm SF.

Each iteration of the loop starts by calling the DecodePoint routine (line 3), shown in Figure 3.49, which returns a decoded point, P, and the *connected* flag, that 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 DecodeSegment routine is called with both endpoints of the segment as inputs (line 9). As we recall from Figure 3.17, the DecodeSegment routine 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). However, if the segment is not connected, the first endpoint of the subsequent encoded segment must be decoded (line 16).

```
output: connected: boolean indicating if the segment is connected
        P: decoded point
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 3.49: DecodePoint routine for algorithm SF.

3.10.1 Example

Next we present an example of the encoding of 12 samples illustrated in Figure 3.6. Since SF is a Linear model algorithm, the specific timestamp values are required in the encoding routine (recall that the timestamps column is an input in Figure 3.45). We consider that the distance between any pair of adjacent timestamps is equal to 60. In this example we use algorithm SF with an error threshold parameter (ϵ) equal to 1, and a 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 incoming point, E, violates the threshold condition. Recall from the coding routine, shown in Figure 3.45, that this occurs occurs when segment *SE* is not included in set *LS* (lines 13-16). In the first iteration, the first sample point is added to *win* (line 6). In the second iteration, two rays, *SMin* and *SMax*, are defined from parameter ϵ, and the first two sample points (lines 8-10). Both rays are shown in Figure 3.50. The second sample point is also added to *win* (line 11).

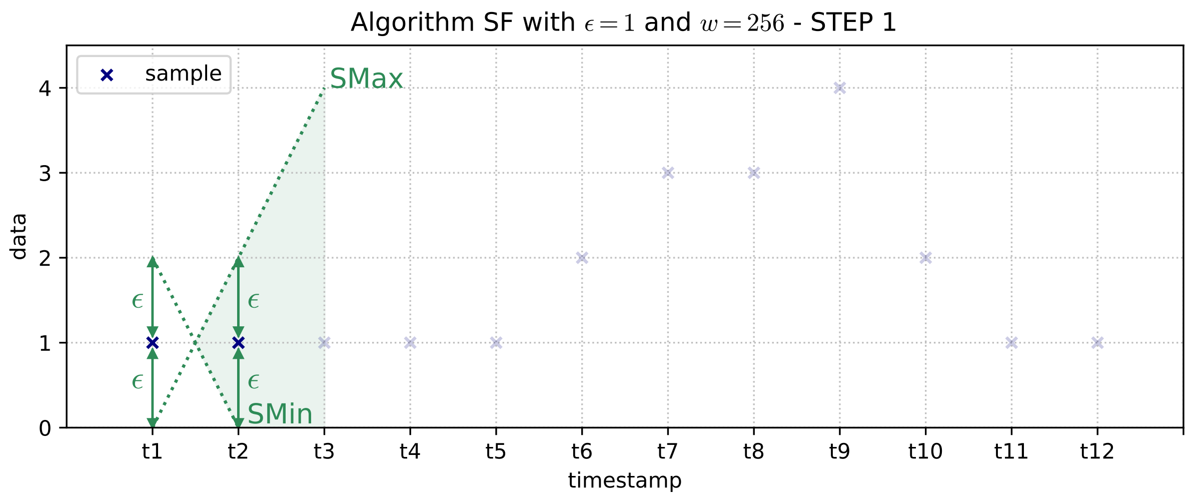

FIGURE 3.50

In the third iteration, sample point *E3* is processed. In Figure 3.51, both *E3* and segment *SE3*, defined in lines 13-14, are shown. *SE3* includes the point in which *SMin* and *SMax* intersect, and *E3*.

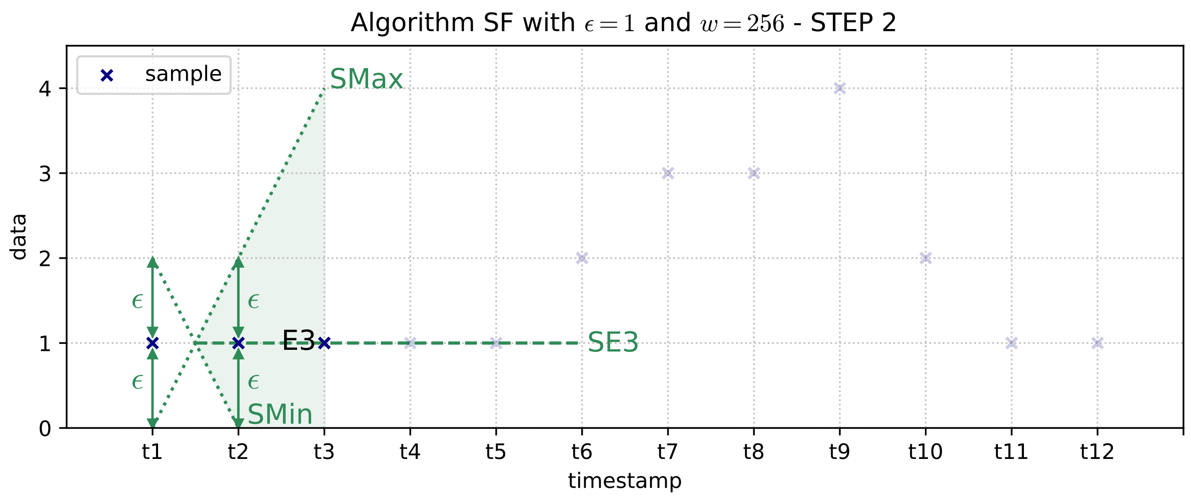

Figure 3.51

Since the slope of segment *SE3* is between the slopes of rays *SMin* and *SMax*, the condition in line 16 is satisfied. Therefore, *SMin* and *SMax* are updated, as shown in Figure 3.52, and *E3* is added to *win* (line 18).

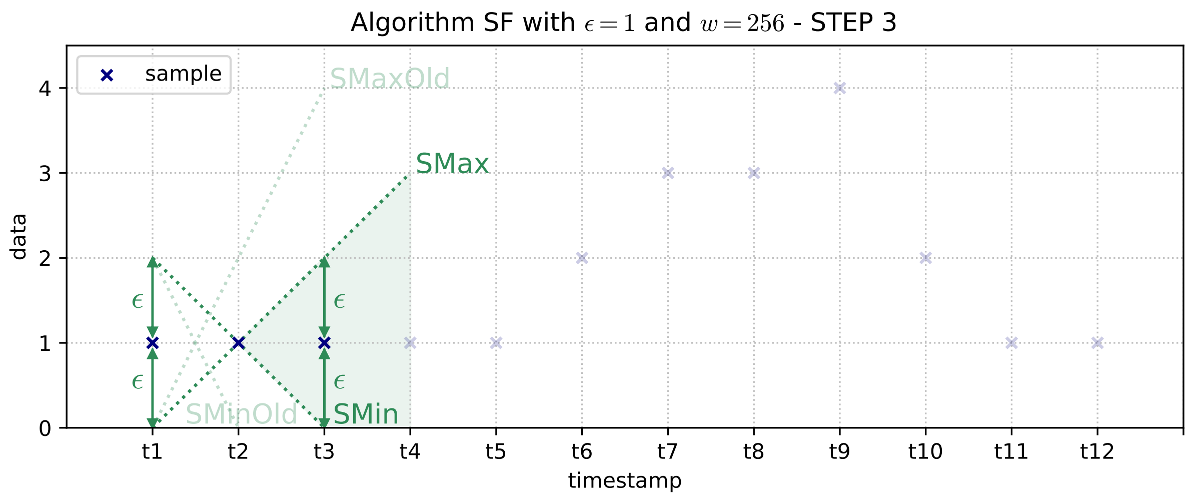

Figure 3.52

In the following six iterations, the lines executed in the coding routine are the same as in the third one. In each iteration, the incoming point is added to the window, and both rays are updated, with the slope of *SMin* increasing, and the slope of *SMax* decreasing. Figure 3.53 shows the information after the 9th iteration is completed. At this point, *win* includes the first 9 sample points, and nothing has yet been encoded in the binary file.

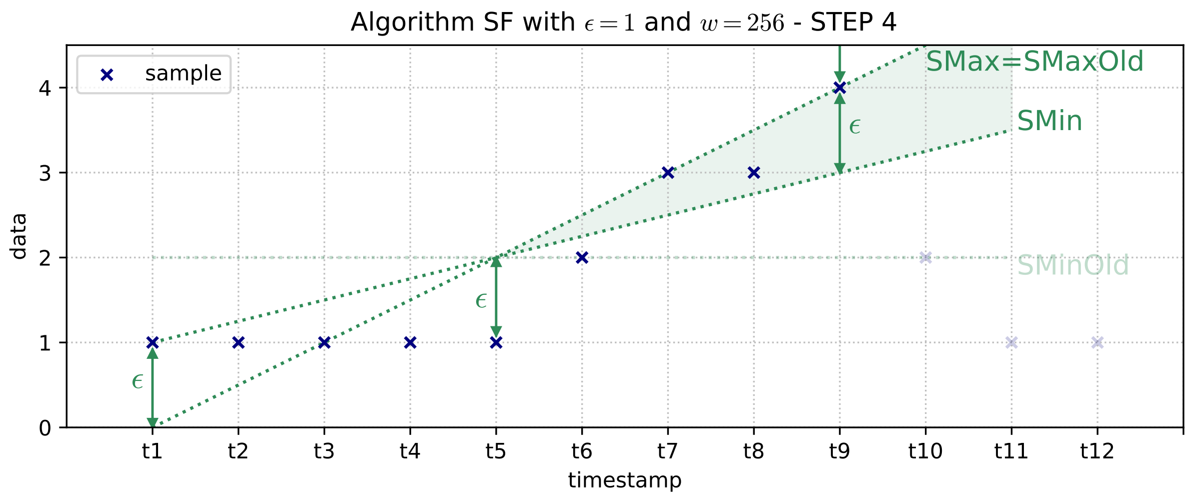

FIGURE 3.53

In the 10th iteration, sample point *E10* is processed. In Figure 3.54, both *E10* and segment *SE10* are shown. This is the first iteration where the condition in line 16 is not satisfied, since the slope of segment *SE10* is smaller than the slope of *SMin*. No window has yet been encoded, so the EncodeWindow routine is called, with parameters *win* and *LS* (line 21). The EncodeWindow routine finds the segment included in *LS* which minimizes the mean square error for the 9 sample points in *win* (among the segments included *LS*). Said segment, and its first endpoint, which is encoded via the EncodePoint routine, are also shown in Figure 3.54. The next encoded point, which is determined in a future iteration, must also belong to that segment. Finally, lines 33 and 34 are executed, the segment is saved, and *E10* is added to a new window.

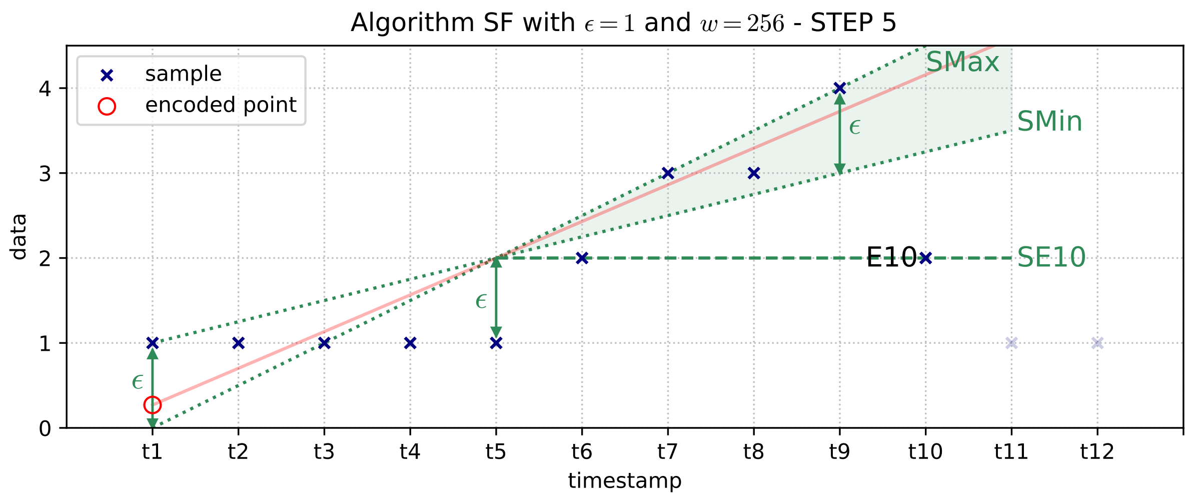

FIGURE 3.54

In the 11th iteration, *win* has a single element, so the two rays are defined from parameter ϵ, and the 10th and 11th sample points (lines 8-10). Both rays are shown in Figure 3.55. The 11th sample point is also added to *win* (line 11).

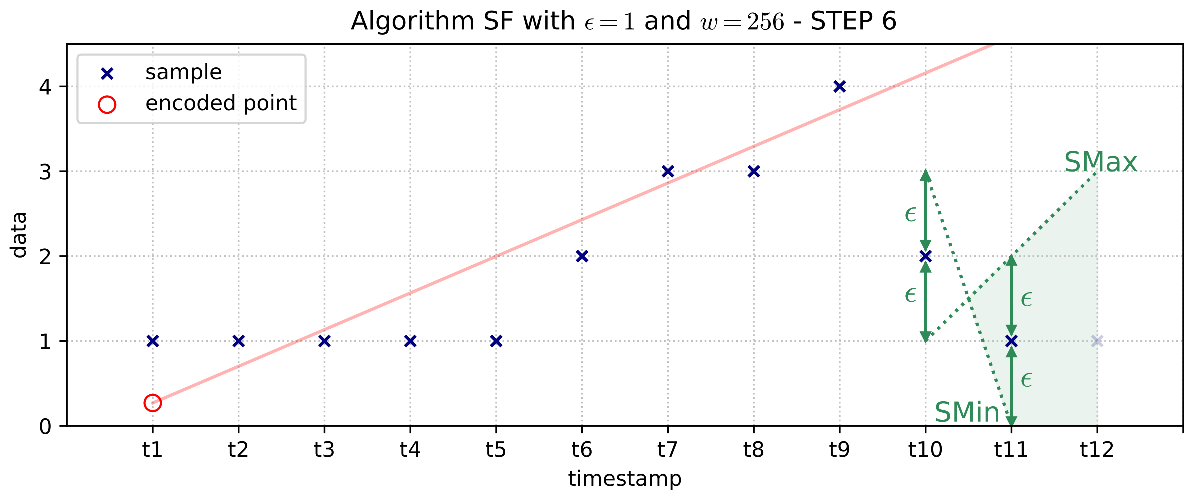

FIGURE 3.55

In the 12th and last iteration, sample point *E12* is processed. In Figure 3.56, both *E12* and segment *SE12*, defined in lines 13-14, are shown. *SE12* includes the point in which *SMin* and *SMax* intersect, and *E12*.

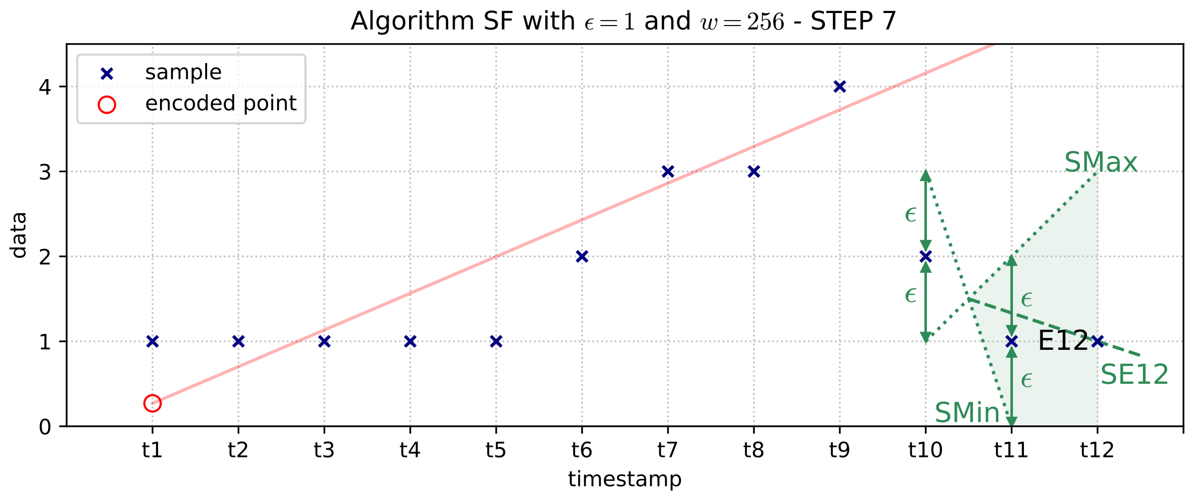

FIGURE 3.56

Since the slope of segment *SE12* is between the slopes of rays *SMin* and *SMax*, the condition in line 16 is satisfied. Therefore, *SMin* and *SMax* are updated, as shown in Figure 3.57, and *E12* is added to *win* (line 18).

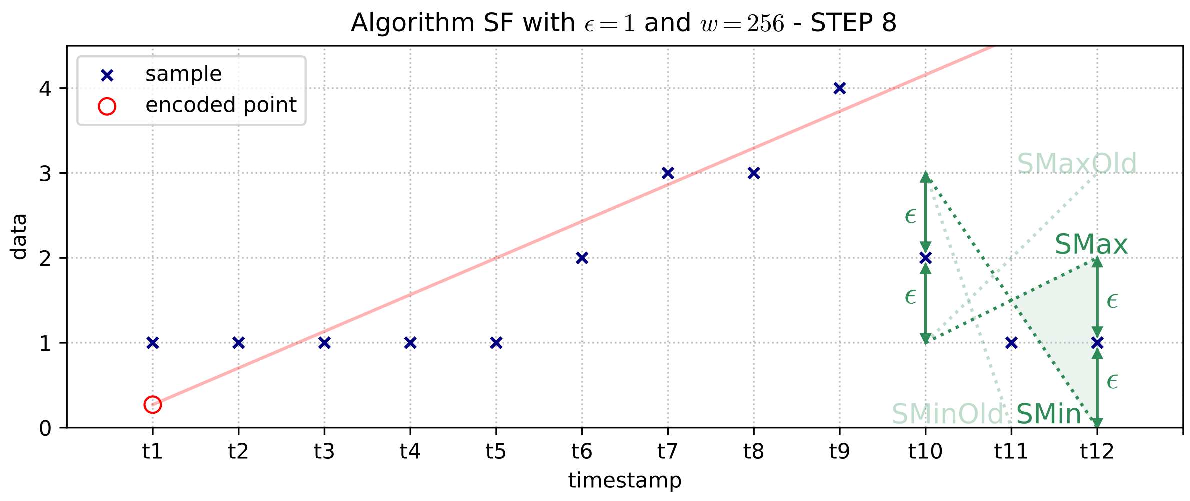

FIGURE 3.57

After the last iteration in the loop finishes, *win* is not empty, so its points are encoded executing the same code as in lines 23-31 (this was left out of the coding routine in Figure 3.45 for clarity). In this case, a connected segment that approximates the sample points in *win* exists, so the intersection point between said segment and the previous segment (the one that was saved, and whose first endpoint was encoded in the 10th iteration), is encoded via the EncodePoint routine (line 26). Since the last sample point has already been processed, the second endpoint of the current segment must also be encoded, through the EncodePoint routine (this was also left out of Figure 3.45 for clarity). In Figure 3.58, the three encoded points and the two associated segments are shown. Notice that the last segment passes through the point in which rays *SMin* and *SMax* intersect, and its slope is between the slopes of those two rays. We point out that, even though in this example the x-coordinate of the intersection point between the segments coincides with a timestamp (integer) value, this is not always necessary the case.

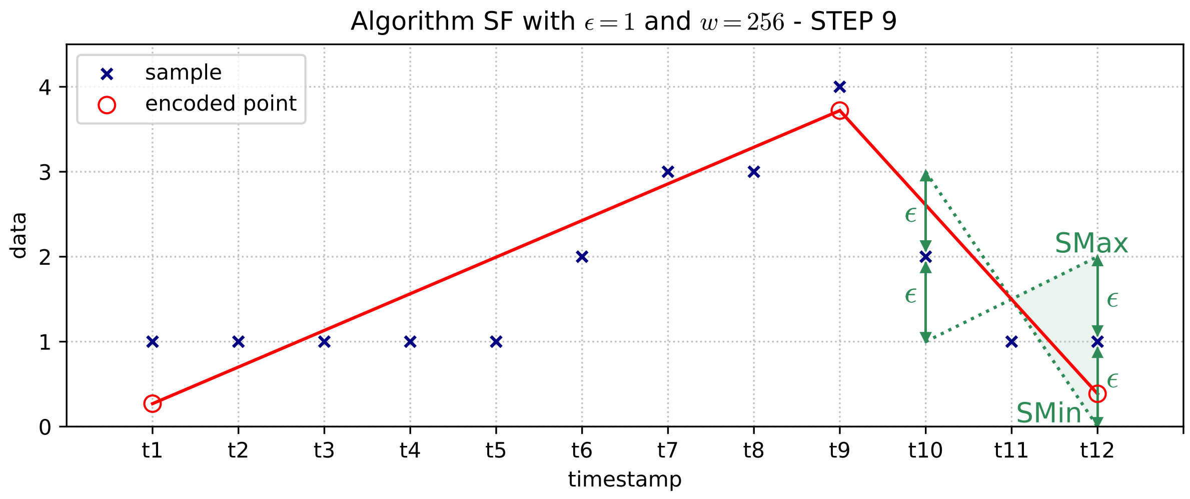

FIGURE 3.58

Finally, in Figure 3.59 we also display the values of the decoded samples, which are the values that the decoding routine, shown in Figure 3.48, would write to the decoded csv data file.

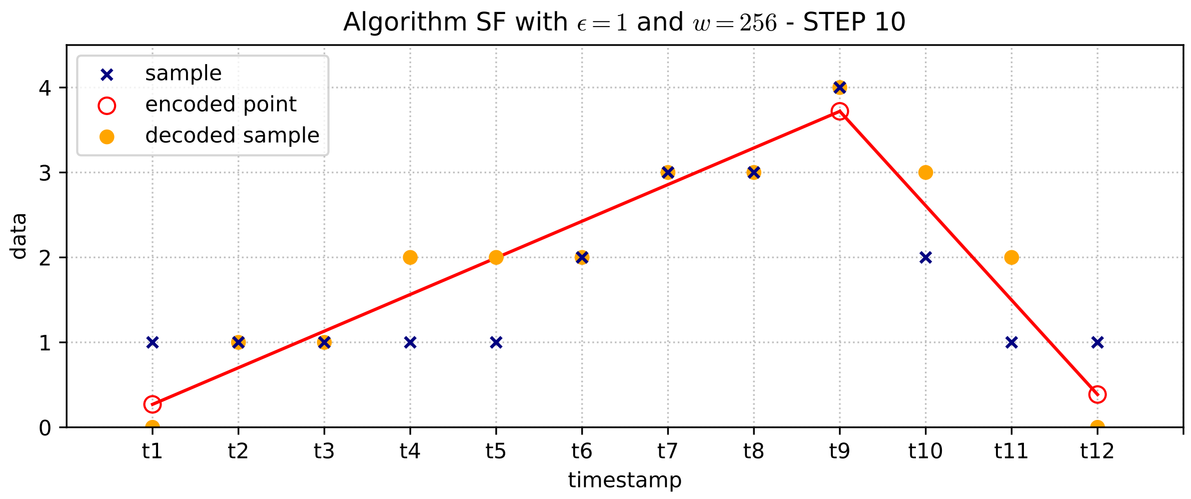

FIGURE 3.59

3.11 Algorithm FR

Algorithm FR [15], also known as Fractal Resampling, 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 processed and encoded. For FR we define a single variant, *M*.

Since FR is a Linear model algorithm, from Section 3.7 we recall that its encoding process involves encoding a sequence of line segments. In particular, the data points in each window are modeled by one or more line segments that are selected using a simple recursive technique called mid-point displacement. In Figure 3.60 we show the coding routine, in which all the column entries are integer values (the gaps are encoded separately). The points obtained from parsing the x-coordinates and y-coordinates from the timestamps and the data column, respectively, are added into consecutive non-overlapping windows of size w (line 1), and each of these windows is encoded independently (lines 3-14).

Since w is always greater than 1, the condition in line 3 may only be satisfied in the last iteration. In the rest of the cases, an array, *displaced_points*, is created (line 9), and passed to the recursive GetDisplacedPoints routine (line 10), which fills it with the *win* indexes of the displaced points. These points correspond to the endpoints of the one or more line segments that model all the points in the window. *displaced_points* always includes the first and last index of *win*, since the first line segment must always begin in the first point of the window, and the last segment must always end in the last point of the window. In lines 11-15, every one of the endpoints is encoded, using $\lceil \log_2 w \rceil$ bits for encoding their *win* index, and *column*.total_bits bits for encoding their y-coordinate.

```
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
        $t\_col$: timestamps column
1  Add the points obtained from parsing the x-coordinates and y-coordinates from $t\_col$
     and $column$, respectively, into consecutive non-overlapping windows of size $w$, except
     possibly for the last window that may consist of fewer points
2  foreach window in the parsing, $win$, do
3      if $|win| == 1$ then
4          Let $value$ be the y-coordinate of the single point in $win$
5          Encode $value$ using a (column-specific) fixed number of bits
6          return
7      end
8      Create an empty array, $displaced\_points$
9      GetDisplacedPoints($win, \epsilon, displaced\_points, 0, |win| - 1$) // routine in Figure 3.61
10     foreach $index$ in $displaced\_points$ do
11         Encode $index$ using $\lceil \log_2 w \rceil$ bits
12         Let $value$ be the y-coordinate of point $P_{index}= win[index]$
13         Encode $value$ using a (column-specific) fixed number of bits
14     end
15 end
```

FIGURE 3.60: Coding routine for variant *M* of algorithm FR.

The recursive GetDisplacedPoints routine is shown in Figure 3.61. In each execution, a line segment, with endpoints given by *win* indexes i_o and i_f, is **considered/evaluated** as a candidate to model all the points in *win* between those endpoints. In line 1, both indexes are added to the *displaced_points* array. The condition in line 2 is satisfied when there are no additional points between both endpoints, which is the base case of the recursive routine. Otherwise, the routine checks if the error threshold condition is violated by any of the points in the candidate segment (line 7), and if that is the case, there are two recursive calls to the routine (lines 10-11). Notice that the last endpoint of the candidate segment in the first call is equal to the first endpoint of the candidate segment in the second call, which is the point given by the *win* index *half*, defined in line 9.

```
input : $win$: window
        $\epsilon$: maximum error threshold
        $displaced\_points$: array including the $win$ indexes of the displaced points
        $i_o$, $i_f$: $win$ indexes of the candidate line segment's endpoints
1  If they are not included, add $i_o$ and $i_f$ to $displaced\_points$, in ascending order
2  if $i_o + 1 < i_f$ then
3      return
4  end
5  Let the following points be defined: $P_o= win[i_o]$ and $P_f= win[i_f]$
6  Let $segment$ be the line segment whose endpoints are $P_o$ and $P_f$
7  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 $\epsilon$
8  if not $valid\_segment$ then
9      Let $half = \lfloor (i_o + i_f)/2 \rfloor$
10     GetDisplacedPoints($win, \epsilon, displaced\_points, i_o, half$)
11     GetDisplacedPoints($win, \epsilon, displaced\_points, half, i_f$)
12 end
```

FIGURE 3.61: GetDisplacedPoints routine for algorithm FR.

The decoding routine is shown in Figure 3.62. 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 independently (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 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 in the window. The first endpoint of the first segment is decoded in lines 9-12, while the last endpoint of each segment, which becomes the first endpoint of each subsequent segment (line 24), is decoded in lines 14-17. Next, the DecodeSegment routine is called with their coordinates as inputs (line 18). This routine (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 condition in line 20 is meant to avoid writing the sample value associated to the last endpoint of a segment, for every segment that models the points in the window, minus the last one. Otherwise, those sample values would be written in the decoded data file twice, since the last endpoint of a segment coincides with the first endpoint of the subsequent segment.

```
input : in: binary file coded with algorithm FR
        out: decoded csv data file
        w: maximum window size
        col_size: number of entries in the column
        t_col: timestamps 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 a (column-specific) fixed number of bits
6          Output value to out
7          return
8      end
9      Decode index using $\lceil \log_2 w \rceil$ bits
10     Decode value using a (column-specific) fixed number of 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 a (column-specific) fixed number of 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 = DecodeSegment(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 3.62: Decoding routine for variant *M* of algorithm FR.

3.11.1 Example

Next we present an example of the encoding of 12 samples illustrated in Figure 3.6. Recall from Example 3.8.1, that since FR is a Linear model algorithm, the specific timestamp values are required in the encoding routine, and we consider that the distance between any pair of adjacent timestamps is equal to 60. In this example we use algorithm FR with an error threshold parameter (ϵ) equal to 1, and a maximum window size (w) equal to 256.

Since there are only 12 samples to encode, a single window, *win*, which includes 12 points, is created in line 1 of the coding routine. In this case, since $|win|$ is greater than 1, the window is encoded in lines 8-14.

The GetDisplacedPoints routine in line 9 is invoked with parameters *win*, $\epsilon = 1$, $displaced\_points = []$, $i_o = 0$ and $i_f = 11$. Figure 3.63 shows the information after executing lines 1-7 in said routine. *displaced_points* is equal to $[0, 11]$, and *segment* is the line segment whose endpoints are the points in *win* with indexes 0 and 11, i.e. the first and last points in the window. Also, *valid_segment* is false, since there are three points in *win* with indexes between 0 and 11, for which the vertical distance to *segment* is greater than ϵ. Since *valid_segment* is false, $half = \lfloor 11/2 \rfloor = 5$ is obtained (line 9), and two recursive calls to the GetDisplacedPoints routine are made (lines 10-11).

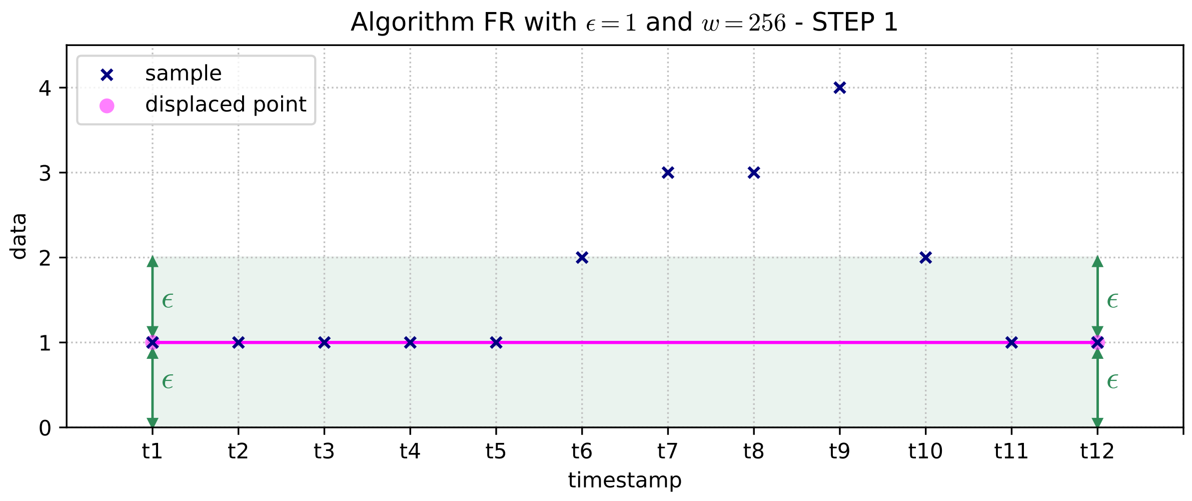

FIGURE 3.63

The first recursive call has parameters $displaced\_points = [0, 11]$, $i_o = 0$ 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 the points in *win* with indexes 0 and 5, and the error threshold constraint is satisfied, since there are no points in *win* with indexes between 0 and 5, for which the vertical distance to *segment* is greater than ϵ. This is shown in Figure 3.64. Therefore, in the execution of the first recursive call, there are no further calls made to the GetDisplacedPoints routine.

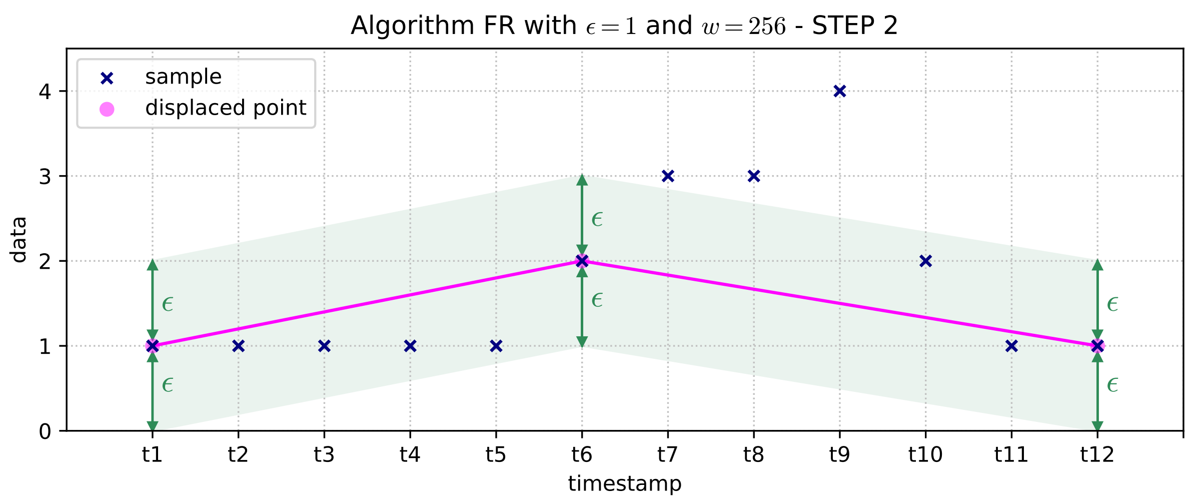

FIGURE 3.64

However, this is not the case in the execution of the second recursive call, which has parameters *displaced_points* $= [0, 5, 11]$, $i_o = 5$ and $i_f = 11$. Here, Figure 3.64 shows that the error threshold constraint is violated by three points in the window, which means that, once again, *half* $= \lfloor(5 + 11)/2\rfloor = 8$ is obtained, and two recursive calls to the GetDisplacedPoints routine must be made. In both of these executions, the error threshold constraint is satisfied, so there are no further calls made to the recursive routine.

After the original invocation of the GetDisplacedPoints routine is completed (line 9 in the coding routine, shown in Figure 3.60), *displaced_points* is equal to $[0, 5, 8, 11]$. This information is shown in Figure 3.65. Observe that all the points in *win* satisfy the error threshold constraint.

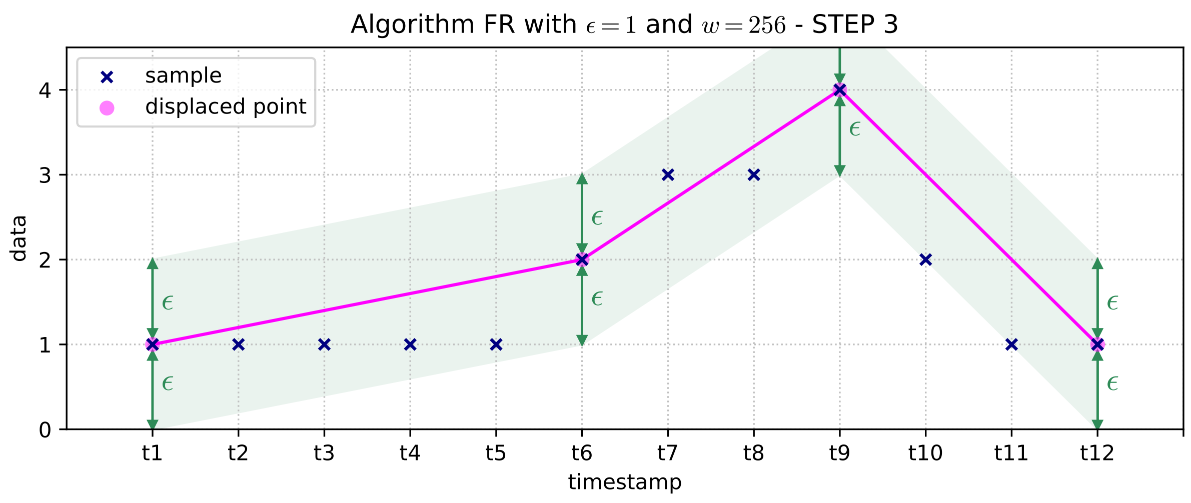

FIGURE 3.65

In this example, the data points in *win* are modeled by three line segments, whose respective endpoints are the four displaced points, which are encoded in lines 10-14 of the coding routine. Encoding an endpoint requires $\lceil \log_2 w \rceil = \log_2 256 = 8$ bits for encoding its *win* index, and *column*.total_bits bits for encoding its y-coordinate. The four encoded endpoints and the associated segments are shown in Figure 3.66. In this figure we also display the values of the decoded samples, which are the values that the decoding routine, shown in Figure 3.62, would write to the decoded csv data file.

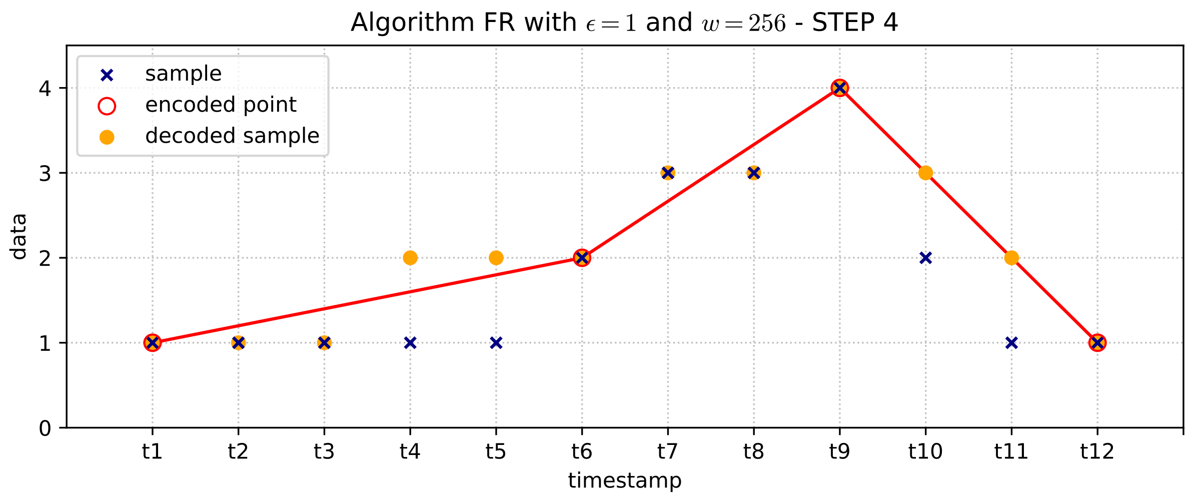

FIGURE 3.66

3.12 Algorithm GAMPS

Algorithm GAMPS [16], also known as Grouping and AMPlitude Scaling, 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 processed and encoded. For GAMPS we define both variants, *M* and *NM*.

In Figure 3.67 we show the coding routine for variant *M*. Notice that the first input consists of all the columns in the csv data file, while in the routines for the Constant and Linear model algorithms, presented in previous sections, it is a single column. This is because those kind of algorithms only exploit the temporal correlation in the data, thus they encode each data column independently. However, Correlation models, such as GAMPS, also exploit the spatial correlation, so the data columns are encoded simultaneously. Another difference with the previously presented algorithms, is that GAMPS is an offline encoder, since the whole dataset is available when the coding routine begins.

```
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 similar columns, where in each subset, a single
     base column is defined (to be encoded with maximum error threshold parameter $\epsilon_b$),
     and the rest of columns in the subset are considered ratio columns (to be encoded with
     maximum error threshold parameter $\epsilon_r$)
2  Encode the number of subsets using $\lceil \log_2 |columns| \rceil$ 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 $\lceil \log_2 |columns| \rceil$ bits in every case
7      Encode base_col by calling the coding routine for variant M of algorithm APCA
         with parameters base_col, out, $\epsilon_b$, $w$ // routine in Figure 3.11
8      foreach column in ratio_cols, ratio_col, do
9          Let trans_ratio_col be obtained by applying the following simple linear
             transform ratio_col/base_col
10         Encode trans_ratio_col by calling the coding routine for variant M of algorithm
             APCA with parameters trans_ratio_col, out, $\epsilon_r$, $w$ // routine in Figure 3.11
11     end
12 end
```

FIGURE 3.67: 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). In each subset, a single base column is defined, and the rest of columns are considered ratio columns. A facility location problem, whose goal is to find a grouping that reduces the number of bits required for encoding the columns, is resolved [16]. Its inputs are the set of columns, *columns*, and the maximum error threshold, ϵ, and the output is the grouping and the maximum error threshold parameters, ϵ_b and ϵ_r, for the base and ratio columns, respectively. To solve the facility location problem we reuse the source code from the framework cited in [8]. 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 3-12), 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, shown in Figure 3.11, 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, then encoded by invoking the APCA coding routine[1], with an error threshold equal to ϵ_r.

Notice that the transformation applied to a ratio column, in line 9, consists in dividing each sample by the base column sample for the same timestamp. The original ratio signals may be rough, but, since there is a high degree of spatial correlation among base and ratio signals in a group, 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. Therefore, the key idea behind algorithm GAMPS is that, under certain circumstances, it can be more convenient to exploit the spatial correlation between the signals and encode the slowly varying transformed signals, rather than independently encode the original rough signals.

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 is decoded. For decoding the data columns in each subset, first, the base column is decoded by invoking the decoding routine for algorithm APCA. Then, each of the transformed ratio columns is decoded, also by invoking the APCA decoding routine, and the transformation applied in line 9 is reverted, by multiplying the decoded base column by the decoded transformed ratio column.

[1]Since the transformed ratio columns may include of non-integer values, in this case the EncodeWindow routine for algorithm APCA, shown in Figure 3.12, must encode the *mid_range* as a float, i.e. using 4 bytes.

3.12.1 Example

Next we present an example of the encoding of three signals with 12 samples each, illustrated in Figure 3.68. Notice that, since the specific timestamp values are irrelevant for algorithm APCA, they are also irrelevant for this algorithm. In this example we use algorithm GAMPS with an error threshold parameter (ϵ) equal to 0, and a maximum window size (w) equal to 256.

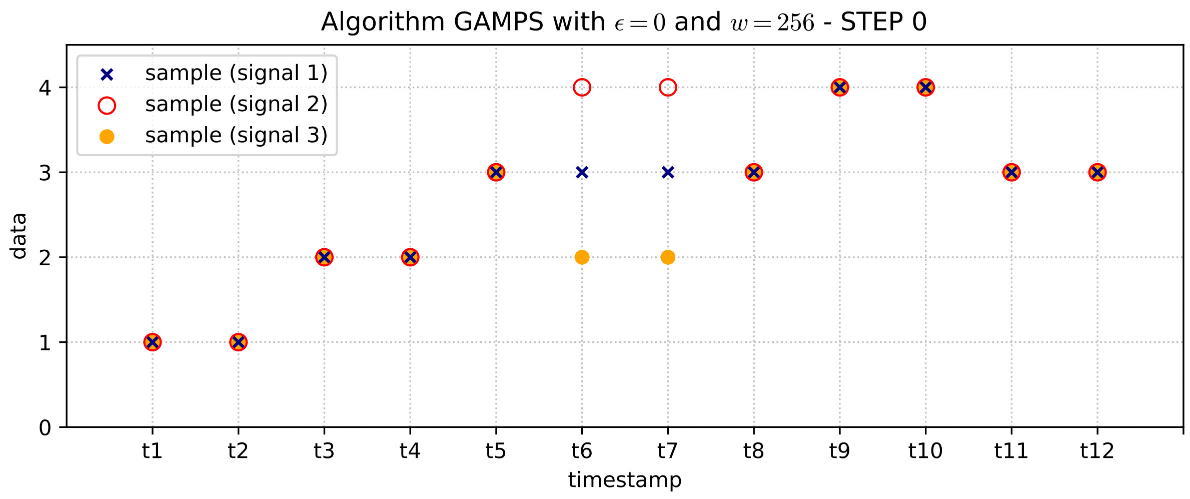

FIGURE 3.68

There is a high degree of spatial correlation among the three signals, with the sample values matching for 10 of the 12 timestamps. Therefore, 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).

Next, in line 7, the base column is encoded by calling the coding routine for algorithm APCA. Since ϵ_b is equal to 0, the encoded samples match the original samples. 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 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. In Figure 3.69, the encoded samples of the three signals are shown.

We recall that the coding routine for algorithm APCA always uses the same amount of bits for encoding a window. If the three signals were independently encoded with APCA, the first one would require encoding 5 windows, and the second and third signals would require encoding 7 windows each. However, in this case, only 3 windows are required for encoding each of the transformed ratio signals with algorithm APCA. 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.

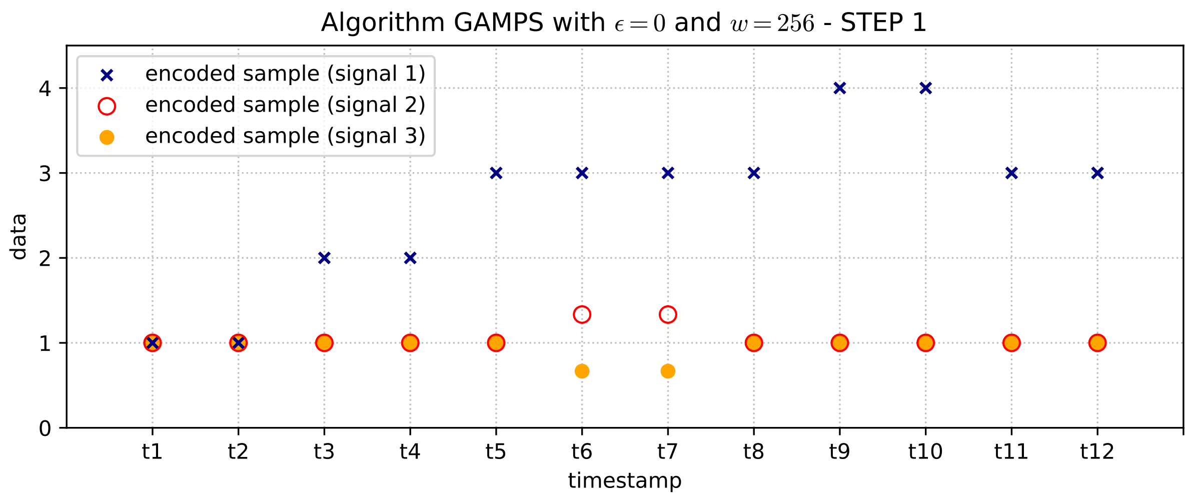

FIGURE 3.69

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* invokes the coding routine for variant *M* of algorithm APCA (lines 7 and 10 in Figure 3.67), which can only handle sample values, since the position of the gaps is already encoded (recall line 7 in Figure 3.1). However, the coding routine for variant *NM* must instead invoke the coding routine for variant *NM* of algorithm APCA, which is able to handle both sample values and gaps.

3.13 Other

EL SIGUIENTE PÁRRAFO ESTABA ORIGINALMENTE EN LA SECTION 3.2

All the algorithms are implemented in C++. Our implementations of algorithms PWLH [12], SF [14] and GAMPS [16] reuse part of the source code from the framework cited in [8][2]. The implementations of the remaining algorithms [10, 11, 13, 15] are entirely ours.

EL SIGUIENTE PÁRRAFO ESTABA ORIGINALMENTE EN LA SECTION 3.3

The version of the AC algorithm we used in our project is the CACM87 implementation [22, 23]. It is written in C and it is one of the most standard implementations. One of its advantages is that it allows to effortlessly set a custom model for the source. However, we had to overcome a minor obstacle to make it work within our scheme. In the CACM87 implementation, the coder closes the encoded file after it has encoded the last symbol. This implies that the decoder recognizes that there are no more symbols left to decode once it reads the last byte of the encoded file. But this is not the case in our masking variant scheme, since after the AC coder has encoded the position of all the gaps in the data, our coding algorithm still has to encode all the data values before closing the encoded file (recall lines 7-9 in Figure 3.1). The problem materialized in the decoding process, because after the AC decoder had decoded the last byte corresponding to the position of the gaps (i.e. the last byte encoded by the AC coder), our decoding algorithm would occasionally continue processing bytes corresponding to the encoded data values, which naturally resulted in an error. The solution we found was to flush the current byte in the stream, before and after executing the AC algorithm, both in the coding and the decoding routines.

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

Chapter 4

Experimental Results

In this chapter we present our experimental results. The main goal of our experiments is to analyze the 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 significative 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/affects** 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 each file, with the results obtained when using the optimal window size for the whole dataset. The results indicate that the **impact/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 the tested coding algorithms, for larger error thresholds APCA is the best algorithm for compressing every data type in our experimental data set, while for lower thresholds the recommended algorithms are PCA, APCA and FR, depending on the data type. If we also consider algorithm gzip, **there isn't an algorithm that is better for compressing every data type for any threshold value / for no threshold value there exists an algorithm that is better for compresing 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* and *SF*), 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 a threshold parameter e (every encoder except *Base*), we test all the values of e in the set $E = \{1, 3, 5, 10, 15, 20, 30\}$, where each threshold is expressed as a percentage fraction of the standard deviation of the data being encoded. For example, for certain data with a standard deviation of 20, taking $e = 10$ implies that the near-lossless compression allows for a maximal per-sample distortion of 2 sampling units.

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

We assess the compression performance of a CAI mainly 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, 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.,

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

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

To compare the compression performance between 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 $c_1, c_2 \in \mathit{CI}$ for the data type z of a certain file f is given by

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

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

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

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

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

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

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

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

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

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

4.2 Comparison of Masking and Non-Masking Variants

In this section, we compare the compression performance of the masking and non-masking variants of every coding algorithm in A_M. 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 threshold parameter, 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 a threshold parameter $e \in E$, for the data type z of a certain dataset d, is given by

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

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 threshold parameter $e \in E$, we calculate the RD between $c_{<a_M,w^*_M,e>}$ and $c_{<a_{NM},w^*_{NM},e>}$, as defined in (4.5), where $w^*_M = \text{OWS}(a_M, e, z, d)$ and $w^*_{NM} = \text{OWS}(a_{NM}, e, z, d)$.

As an example, in figures 4.1 and 4.2 we show the CR and the RD, as a function of the threshold parameter, obtained for two data types of two different datasets. Figure 4.1 shows the results for the data type “SST” of the dataset SST, and Figure 4.2 shows the results for the data type “Longitude” of the dataset Tornado. 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.

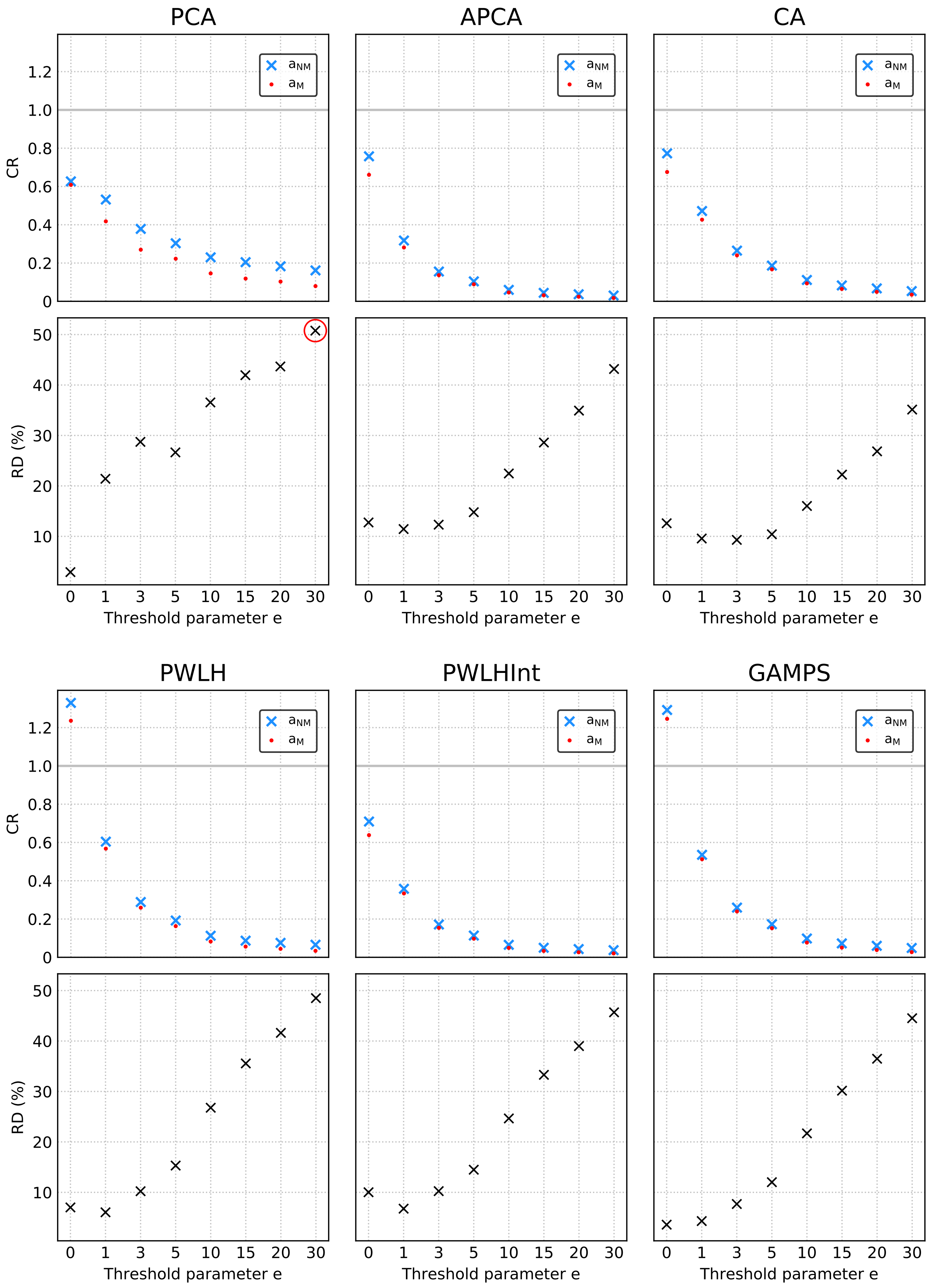

FIGURE 4.1: CR and RD plots for every pair of algorithm variants $a_M, a_{NM} \in A_M$, for the data type "SST" of the dataset SST. In the RD plot for algorithm PCA we highlight with a red circle the marker for the maximum value (50.78%) obtained for all the tested CAIs.

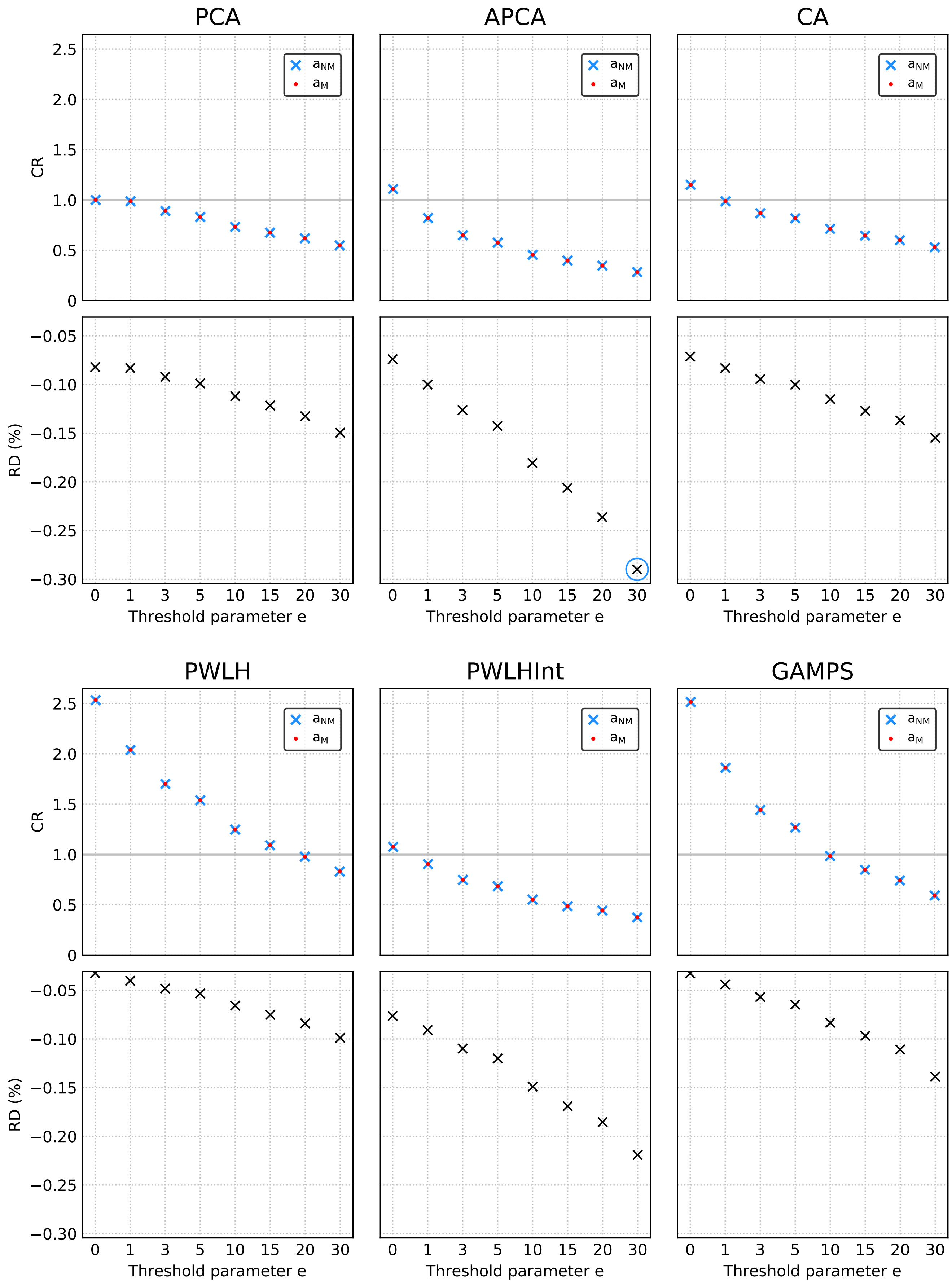

FIGURE 4.2: CR and RD plots for every pair of algorithm variants $a_M, a_{NM} \in A_M$, for the data type "Longitude" of the dataset Tornado. In the RD plot for algorithm APCA we highlight with a blue circle the marker for the minimum value (-0.29%) obtained for all the tested CAIs.

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 threshold parameter $e \in E$, and we calculate the RD between the CAIs $c_{<a_M, w^*_M, e>}$ and $c_{<a_{NM}, w^*_{NM}, e>}$, obtained by setting the OWS for the masking variant a_M and the non-masking variant a_{NM}, respectively. Since we consider 8 threshold parameters 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 amount 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 description 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 also notice 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_W^*, which is obtained from V^* by discarding algorithm SF, which doesn't have a window size parameter (recall this information from Table 3.1). Also, 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 $a_v \in V_W^*$, and a threshold parameter $e \in E$, for the data type z of a certain file f is given by

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

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 $a_v \in V_W^*$, and threshold parameter $e \in E$, we calculate the RD between $c_{<a_v, w^*_{global}, e>}$ and $c_{<a_v, w^*_{local}, e>}$, as defined in (4.2), where $w^*_{global} = \mathrm{OWS}(a_v, e, z, d)$ and $w^*_{local} = \mathrm{LOWS}(a_v, e, z, f)$. In what follows, we denote the OWS and the LOWS as w^*_{global} and w^*_{local}, respectively.

As an example, in figures 4.3 and 4.4 we show w^*_{global}, w^*_{local}, and the RD between $c_{<a_v, w^*_{global}, e>}$ and $c_{<a_v, w^*_{local}, e>}$, as a function of the threshold parameter e, obtained for the data type $z =$ "VWC", for two different files of the dataset $d =$ IRKIS. Figure 4.3 shows the results for the file $f =$ "vwc_1202.dat.csv", and Figure 4.4 shows the results for $f =$ "vwc_1203.dat.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 threshold 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 (10.68%) 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 .

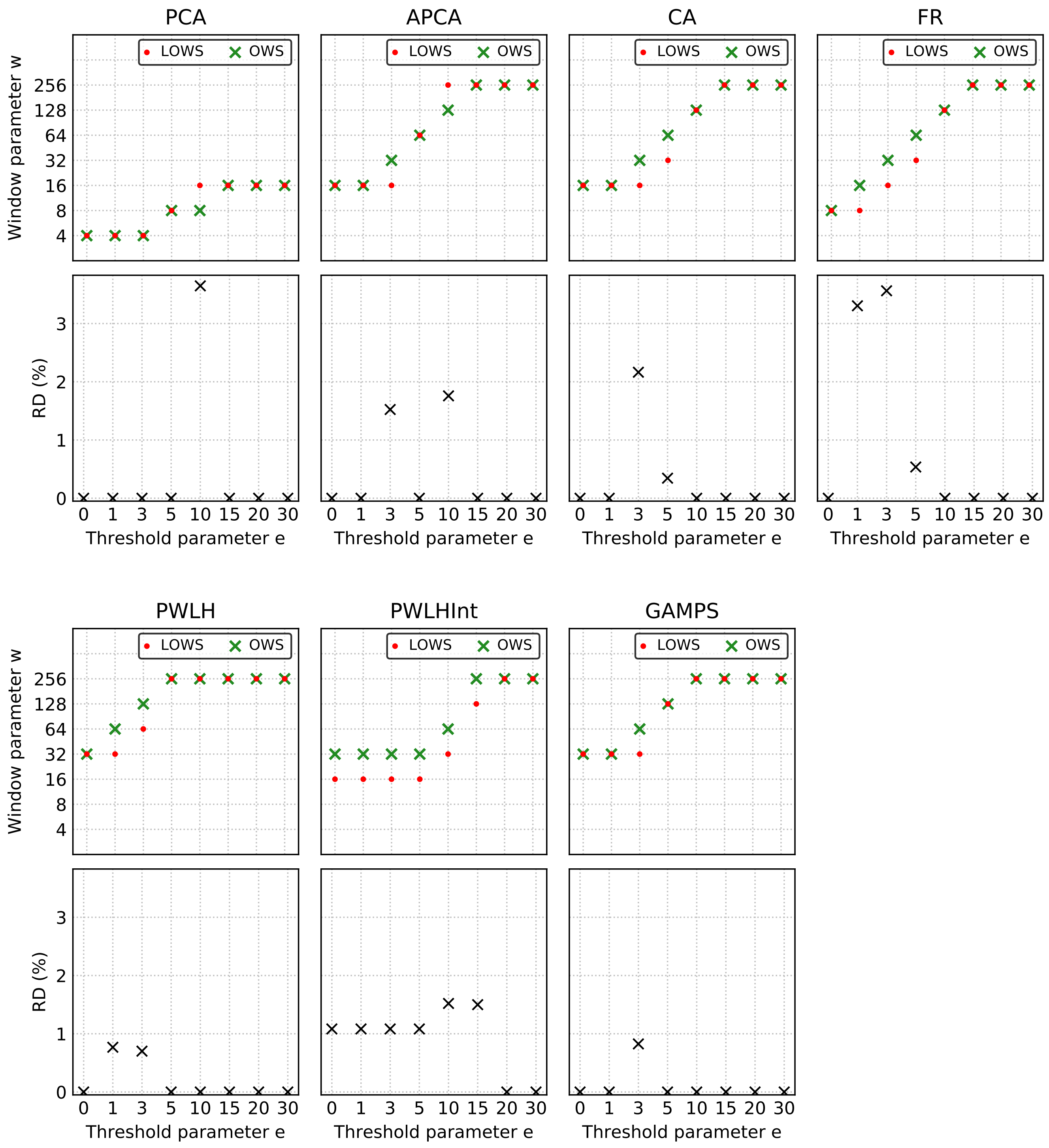

FIGURE 4.3: Plots of w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the threshold parameter e, obtained for the data type "VWC" of the file "vwc_1202.dat.csv" of the dataset IRKIS.

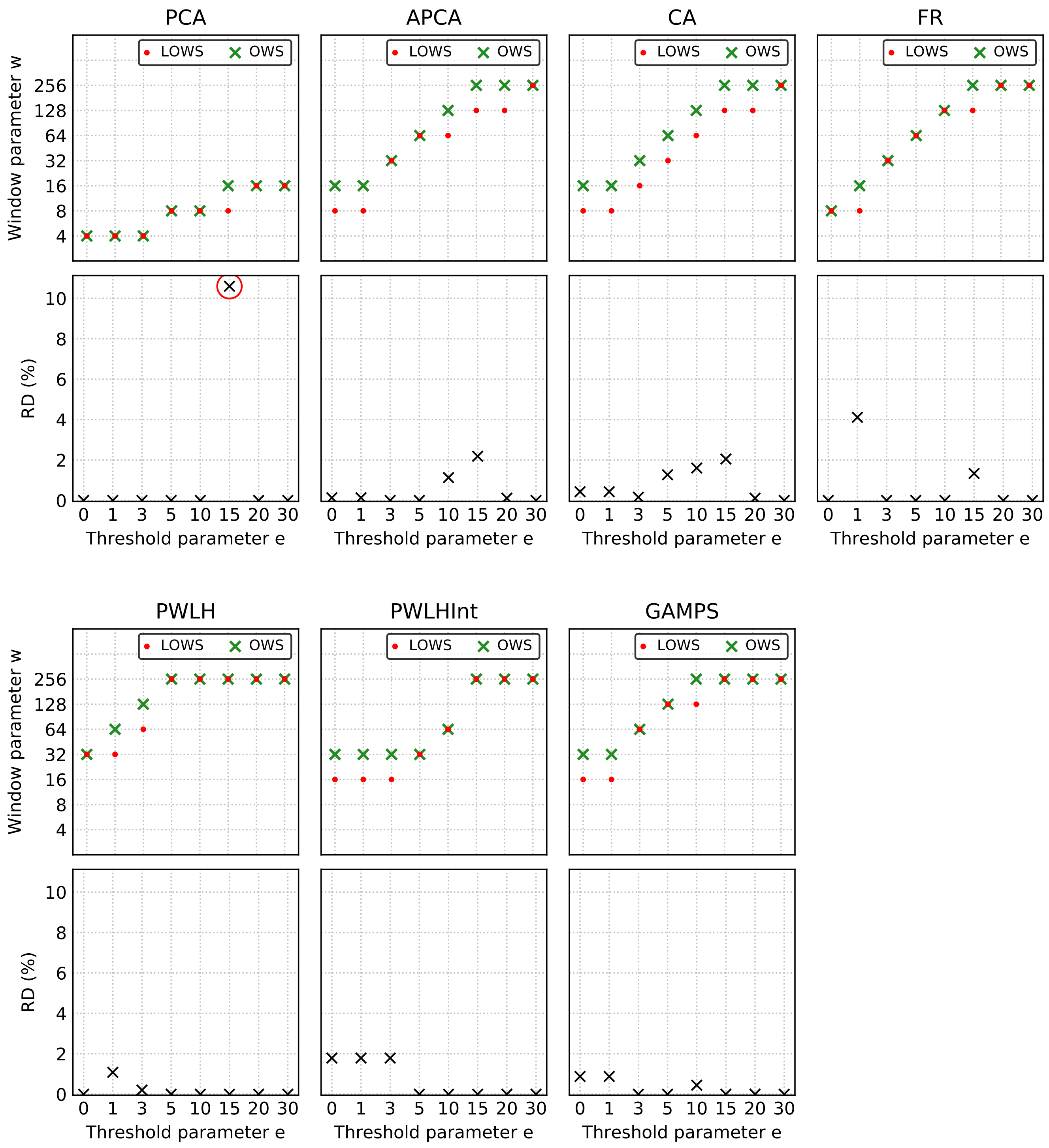

FIGURE 4.4: Plots of w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the threshold parameter e, obtained for the data type "VWC" of the file "vwc_1203.dat.csv" of the dataset IRKIS. In the RD plot for algorithm PCA we highlight 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 threshold 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 threshold parameters and there are 13 files with a single data type and 4 files with 3 different data types each, for each algorithm we compare a total of $8 \times (13 + 4 \times 3) = 200$ pairs of CAIs. Table 4.2 summarizes the results of these comparisons, aggregated by algorithm and the range to which the RD belongs.

	RD (%) Range				
Algorithm	**0**	**(0,1]**	**(1,2]**	**(2,5]**	**(5,11]**
PCA	186 (93%)	3 (1.5%)	4 (2%)	2 (1%)	5 (2.5%)
APCA	174 (87%)	13 (6.5%)	7 (3.5%)	6 (3%)	0
CA	172 (86%)	16 (8%)	6 (3%)	6 (3%)	0
FR	171 (85.5%)	14 (7%)	8 (4%)	7 (3.5%)	0
PWLH	184 (92%)	13 (6.5%)	3 (1.5%)	0	0
PWLHInt	173 (86.5%)	9 (4.5%)	13 (6.5%)	4 (2%)	1 (0.5%)
GAMPS	182 (91%)	13 (6.5%)	5 (2.5%)	0	0
Total	1,242 (88.7%)	81 (5.8%)	46 (3.3%)	25 (1.8%)	6 (0.4%)

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 third 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 88.7% 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. In 97.8% of the cases, the RD is less than or equal to 2%. This means that, for the vast majority of CAI pairs, either the OWS and the LOWS match or they yield roughly the same compression performance. This result suggests that we could fix the window size parameter in advance, for example by optimizing over a training set, without 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 6 cases (0.4%) in which the RD falls in the range (5, 11], most of which (5 cases) involve the algorithm PCA. The maximum value taken by RD (10.6%) is obtained for the data type "VWC" of the file "vwc_1203.dat.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 algorithm PCA 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 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 threshold parameter $e \in E$, we calculate the CR of $c_{<a_v, w^*_{global}, e>}$, as defined in (4.4), where $w^*_{global} = \text{OWS}(a_v, e, z, d)$. The following definition is useful for analyzing which CAI obtains the best compression result for a specific data type.

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

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

When $c^b(z,d,e) = c_{<a^b_v, w^{b*}_{global}, e>}$, we refer to a^b and w^{b*}_{global} 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 datasets. As an example, in Figure 4.5 we show the CR and the window size parameter w^*_{global}, as a function of the threshold parameter, obtained for each algorithm, for the data type “SST” of the dataset ElNino. For each threshold parameter $e \in E$, we use blue circles to highlight the markers for the minimum CR value and the best window parameter (in the respective plots corresponding to the best 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. So in this case, algorithm PCA is the best coding algorithm, and 256 is the best window size. For the remaining seven values for the threshold 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.

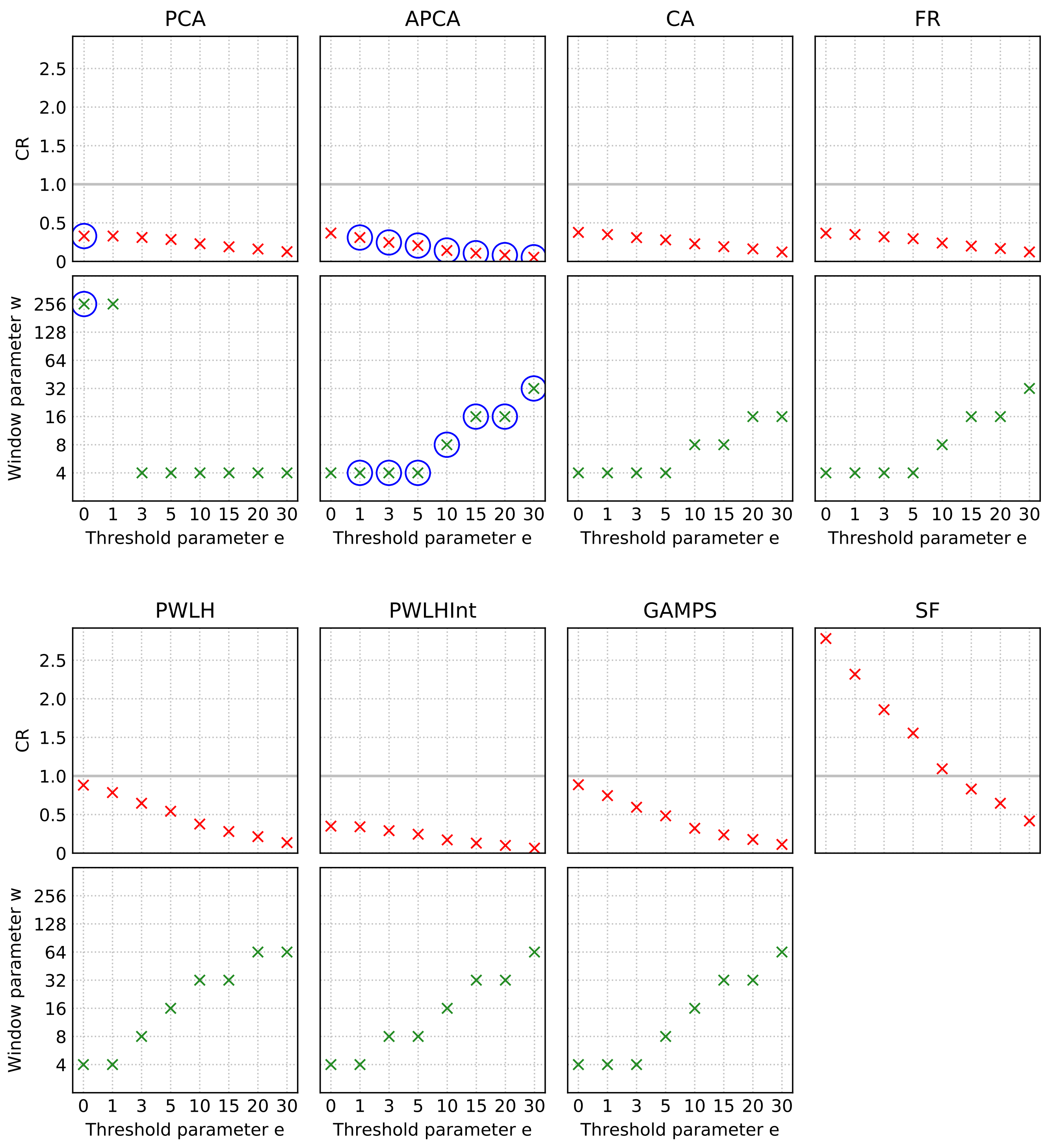

FIGURE 4.5: CR and window size parameter plots for every algorithm, for the data type "SST" of the dataset ElNino. For each threshold parameter $e \in E$, we use blue circles to highlight the markers for the minimum CR value and the best window size parameter (in the respective plots corresponding to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

		e = 0		e = 1		e = 3		e = 5		e = 10		e = 15		e = 20		e = 30	
Dataset	Data Type	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w
IRKIS	VWC	0.20	4	0.18	4	0.12	5	0.07	6	0.03	7	0.02	8	0.02	8	0.01	8
SST	SST	0.61	8	0.28	3	0.14	5	0.09	6	0.05	7	0.03	8	0.02	8	0.02	8
ADCP	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
	Z. 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
	M. 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
	AirTemp	0.33	8	0.33	8	0.30	2	0.27	2	0.22	2	0.19	3	0.17	3	0.13	4
	SST	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 thresholds on each data type of each dataset. Each row contains information relative to certain data type. For each threshold, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the window size parameter 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) which are used by the best CAI for at least one of the 168 possible data type and threshold parameter combinations. Algorithm APCA is used 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 used 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 threshold parameter $e \in E$ is given by

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

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

The maxRD metric is useful for assessing the compression performance of a coding algorithm a on the set of data types as a whole. Notice that maxRD is always non-negative. A satisfactory result (i.e. close to zero) can only be obtained when a achieves a good compression performance *for every data type*. In other words, bad compression performance *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 maxRD(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 $a \neq$ APCA, maxRD(a, e) > 0 for every $e \geq 10$.

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

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

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

Again, minmaxRD is always always non-negative. Notice that minmaxRD(e) = 0 for certain e, if and only if there exists a minmax coding algorithm a such that 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 maxRD(APCA, e) = 0 for every $e \geq 10$.

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

	maxRD (%)							
Algorithm	e = 0	e = 1	e = 3	e = 5	e = 10	e = 15	e = 20	e = 30
PCA	40.51	42.27	53.11	62.01	71.71	75.28	77.15	80.21
APCA	33.25	15.64	9.00	29.96	0	0	0	0
CA	38.28	38.28	54.68	63.12	65.44	72.94	77.21	81.84
PWLH	73.46	72.93	72.52	82.14	83.24	86.86	88.94	91.19
PWLHInt	29.72	34.00	49.94	68.95	76.68	69.96	74.72	79.89
FR	48.75	49.85	52.21	52.70	54.82	55.35	54.48	64.72
SF	92.46	92.23	92.16	92.11	91.67	91.24	90.95	91.33
GAMPS	72.51	72.37	69.76	68.88	67.16	67.14	67.18	66.16

TABLE 4.4: maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^*$ and threshold parameter $e \in E$. For each e, the cell corresponding to the 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 threshold 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 threshold 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 [24]. This algorithm only operates in lossless mode (i.e. the threshold 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 [25].

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 threshold, 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. Notice 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 threshold 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 near-lossless algorithms performance 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

		e = 0		e = 1		e = 3		e = 5		e = 10		e = 15		e = 20		e = 30	
Dataset	Data Type	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w
IRKIS	VWC	0.13		0.13		0.12	5	0.07	6	0.03	7	0.02	8	0.02	8	0.01	8
SST	SST	0.52		0.28	3	0.14	5	0.09	6	0.05	7	0.03	8	0.02	8	0.02	8
ADCP	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
	Z. 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
	M. 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
	AirTemp	0.33	8	0.33	8	0.30	2	0.27	2	0.22	2	0.19	3	0.17	3	0.13	4
	SST	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 thresholds on each data type of each dataset, including the results obtained by gzip. Each row contains information relative to certain data type. For each threshold, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the window size parameter for the best algorithm (the one that achieves the minimum CR), which is identified by a certain cell color described in the legend above the table. Algorithm gzip doesn't have a window size parameter, so the cell is left blank in these cases.

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

We observe that, for every e, the minmaxRD values are rather high, the minimum being 26.58% (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% ($e = 3$ and $e = 5$). 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 had the possibility of selecting gzip as a compression algorithm, APCA would no longer be the obvious choice for compressing any data type when $e \geq 10$, as we had concluded in the previous section.

	maxRD (%)							
Algorithm	e = 0	e = 1	e = 3	e = 5	e = 10	e = 15	e = 20	e = 30
GZIP	26.66	47.99	73.79	82.94	91.10	93.94	95.40	96.69
PCA	73.93	73.54	68.27	67.21	71.71	75.28	77.15	80.21
APCA	59.11	58.39	54.36	54.36	54.35	54.33	54.33	42.85
CA	73.82	73.45	69.31	68.29	65.44	72.94	77.21	81.84
PWLH	87.51	87.45	87.36	87.26	87.01	86.86	88.94	91.19
PWLHInt	71.26	71.01	70.71	68.95	76.68	69.96	74.72	79.89
FR	76.46	76.12	72.78	71.97	67.37	64.31	64.01	64.72
SF	96.30	96.13	96.09	95.96	95.86	95.74	95.63	95.04
GAMPS	83.73	83.72	83.72	83.72	83.72	83.71	83.71	78.78

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

4.5 Conclusions

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 4.6.

In our research we have also compared the compression performance of the coding algorithms' masking and non-masking variants. 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 significative 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/affects** 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 **impact/effect** of using the optimal global window size instead of the optimal local window size for each file is rather small.

4.6 Future Work (TODO)

Some ideas:

- Consider non-linear models (e.g. Chebyshev Approximation)
- Consider new datasets
- Consider other metrics (e.g. RMSE)
- Investigate why certain algorithms perform better on certain data types

- Create universal coder, with every algorithm as a subroutine

Bibliography

[1] N. Wever. IRKIS Soil moisture measurements Davos. SLF. https://doi.org/10.16904/17, 2017. [Accessed 9 November 2020].

[2] 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.

[3] NOAA - TAO Data Download. https://tao.ndbc.noaa.gov/tao/data_download/search_map.shtml, 2016. [Accessed 9 November 2020].

[4] El Nino Data Set. https://archive.ics.uci.edu/ml/datasets/El+Nino, 1999. [Accessed 9 November 2020].

[5] SolarAnywhere - Data. https://data.solaranywhere.com/Data, 2020. [Accessed 9 November 2020].

[6] Storm Prediction Center - Severe Weather Database Files (1950-2017). https://www.spc.noaa.gov/wcm/#data, 2016. [Accessed 9 November 2020].

[7] TAO - History of the Array. https://tao.ndbc.noaa.gov/proj_overview/taohis_ndbc.shtml, 2013. [Accessed 9 November 2020].

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] 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.

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

[16] 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.

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

[18] Arithmetic Coding + Statistical Modeling = Data Compression. `https://marknelson.us/posts/1991/02/01/arithmetic-coding-statistical-modeling-data-compression.html`, 1991. [Accessed 9 November 2020].

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

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

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

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

[23] Data Compression With Arithmetic Coding. `https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html`, 2014. [Accessed 9 November 2020].

[24] GNU Gzip. `https://www.gnu.org/software/gzip/`, 2018. [Accessed 9 November 2020].

[25] GNU Gzip Manual - Invoking gzip. `https://www.gnu.org/software/gzip/manual/html_node/Invoking-gzip.html`, 2018. [Accessed 9 November 2020].

[26] Solomon W. Golomb. Run-length encodings (Corresp.). *IEEE Transactions on Information Theory*, 12(3), 1966.