Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Martin Lysík

Fraktální komprese časových řad

Fractal compression of time series

Department of Software Engineering

Supervisor: Doc. RNDr. Tomáš Skopal, Ph.D.

Field of study: Software Systems

2009

I would like to thank to Doc. RNDr. Tomáš Skopal, Ph.D. for his advices and time that he spend on me during consultations.

I proclaim that I worked on this thesis on my own and used only referred resources. I agree with public availability of this thesis.

In Prague, July 30, 2009 Martin Lysík

CONTENTS

Title: Fractal compression of time series

Author: Martin Lysík

Department: Department of Software Engineering

Supervisor: Doc. RNDr. Tomáš Skopal, Ph.D.

Supervisor's e-mail address: Tomas.Skopal@mff.cuni.cz

Abstract:

The aim of this work was looking for single dimensional distributions of fractals in real world time series and use them to compress these time series. Usability of these principles for both lossless and lossy compression was examined.

Base on the problem analysis was as first designed and implemented the basic compression algorithm. This was progressively extended with simple heuristics for better performance and also other techniques, which should have reduced its deficiencies. As the result were created two more extended compression algorithms and one algorithm with different data processing. Properties of these algorithms, output sizes and quality of decompressed data were compared on several input data and algorithms were also compared with existing compress algorithms and methods for storing time series data.

Keywords: time series, compression, fractal

Názov práce: Fraktálna kompresia časových radov

Autor: Martin Lysík

Katedra: Katedra softwarového inženýrství

Vedúci diplomovej práce: Doc. RNDr. Tomáš Skopal, Ph.D.

E-mail vedúceho: Tomas.Skopal@mff.cuni.cz

Abstrakt:

Cieľom tejto práce bolo vyhľadávanie jednorozmerných fraktálnych distribúcií v reálnych časových radoch a ich použitie na kompresiu týchto časových radov. Bola preskúmaná použiteľnosť tejto metódy na bezstratovú ako aj stratovú kompresiu.

Na základe analýzy problému bol ako prvý navrhnutý a implementovaný základný kompresný algoritmus. Tento bol postupne doplnený o jednoduché heuristiky pre rýchlejšie spracovanie dat a tiež rozširovaný o ďalšie kroky, ktoré mali minimalizovať jeho nedostatky. Ako výsledok vznikli dva rozširujúce kompresné algoritmy a jeden algoritmus s rozdielnym spôsobom spracovania dat. Chovanie týchto algoritmov, veľkosť výstupov a kvalita dekomprimovaných dát boli porovnané na rôznych vstupných dátach a algoritmy boli porovnané aj s existujúcimi kompresnými algoritmami a metódami používanými pre uchovávanie časových radov.

Kľúčové slová: časové rady, kompresia, fraktál

1 INTRODUCTION

In these times almost all data are stored in digital form in files or in databases and are processed mostly on computers or are processed by various programs or applications. One group of these data consists of time series. Time series is a sequence of values of some variable that are measured in time. It is important that these observations are in chronological sequence. [4] Most of time series data come from the physical or technical sciences (e.g. an output of seismograph in geophysics, a sequence of temperatures in meteorology), biological sciences (e.g. monitoring of air pollution or ECG records used in medicine) or from social sciences (e.g. monitoring of number and composition of population in demography or number of divorces in sociology).

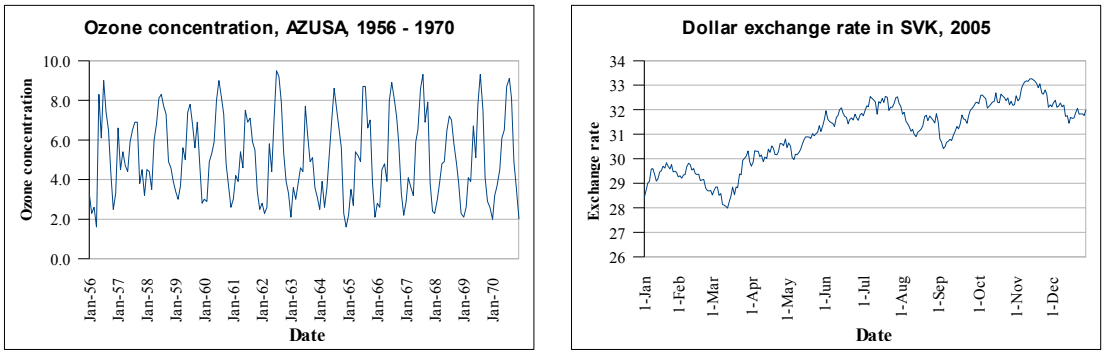

Figure 1.1 Time series examples.

In figure 1.1 we can see 2 examples of time series. Left one displays values of ozone concentration in Azusa (California) during years 1956 - 1970. One value was given for each month. [14] On the right side are displayed values of dollar exchange rate according to Slovak currency in year 2005. Each work day one value was added. [15]

Some types of time series need to be stored exactly because these data are used to calculate statistic analyses and it is desired to work with exact values. Here belong time series obtained in the meteorology (temperatures, water levels, ...) or in the financial engineering (exchange rates, stock prices, ...). On the other hand some data in time series are used only for visual check for some patterns or anomalies (ECG records) and it is sufficient to store these data in lossy compressed form.

1.1 Fractal

Fractal is generally a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole. [9] The property of fractal, that it looks like itself at different scales, is called *self similarity*. Many real-world objects have shapes that have self similar look (clouds, mountains, snow flakes, ...). Even though fractals have too irregular structure to describe them in the Euclidean geometric language, they often could be described by simple recursive formulae.

As the representative fractal the *Cantor set* could be mentioned. It is set of points in line interval and it could be created from the line by splitting it into three identical parts, remove the middle part and then repeat this routine for remaining line subintervals in infinite loop. Cantor set (or Cantor dust) is the set of points in

remained intervals. Mandelbrot noticed that occurrences of noise in electronic transmission line are distributed as points in cantor dust alternate with noise-free sequences. So when some noisy signal sequence of arbitrarily small length was taken, it always contained at least one noise-free subsequence.

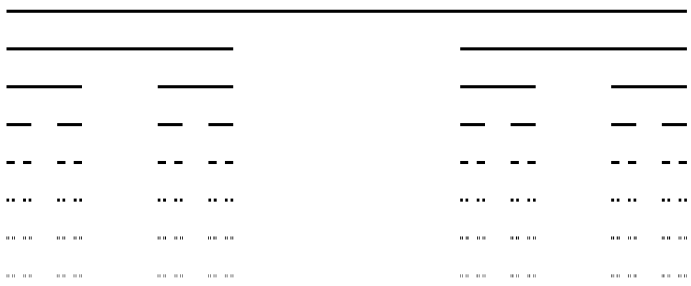

Figure 1.2 Generation of Cantor set.

Mandelbrot found fractal distribution in time series of electronic signal. According to these observations it could be reasonable to try to look for fractal distributions in other types of real-world time series and try to use them for time series compression.

1.2 Fractal compression

When introducing to fractals or fractal compression of pictures, almost always is referred some kind of copy machine (*Multiple Reduction Copy Machine* [1]) that scans given picture and prints picture that consists of reduced copies of given picture. When the process is repeated for the output pictures then these will be similar to some pattern. When this process is repeated infinite times then result picture will be always the same, no matter which initial picture has been used. To describe how single reduced picture is placed in output picture an affine linear transformation could be used. Set of these affine linear transformations is called an *Iterated Function System* and the final picture is called an *Attractor*. [1]

These principles could be generalized to all complete metric spaces and the sets of contractive functions on these metric spaces. Because domain and range of each function is whole metric space, they can be composite in the single function which will be contractive also. And according to the *Banach fixed point theorem* this function has one and only fixed point in this metric space, that can be reached by applying this function iterative on any point from this metric space.

Lets create some rules to generate some simple sequence of length 16 using these contractive mappings on $\mathbb{R}^{16}$.

- $y=f_1(x)$: $y_i=-0.5(x_{2i+7}+x_{2i+8})+7$ for $1\le i\le 4$, $y_i=0$ otherwise.
- $y=f_2(x)$: $y_{i+4}=0.25(x_{2i-1}+x_{2i})+10$ for $1\le i\le 8$, $y_i=0$ otherwise.
- $y=f_3(x)$: $y_{i+12}=0.5(x_{2i-1}+x_{2i})-7$ for $1\le i\le 4$, $y_i=0$ otherwise.

In figure 1.3 are displayed application of these mappings on three different initial sequences. In first row are displayed the initial sequences (zero sequence, some random sequence and attractor of these mappings). In next rows are displayed the sequences after 1, 2, 3 and 20 iterations. As we can see, definitions of these contractive mappings keep enough information to generate sequence that is attractor of them.

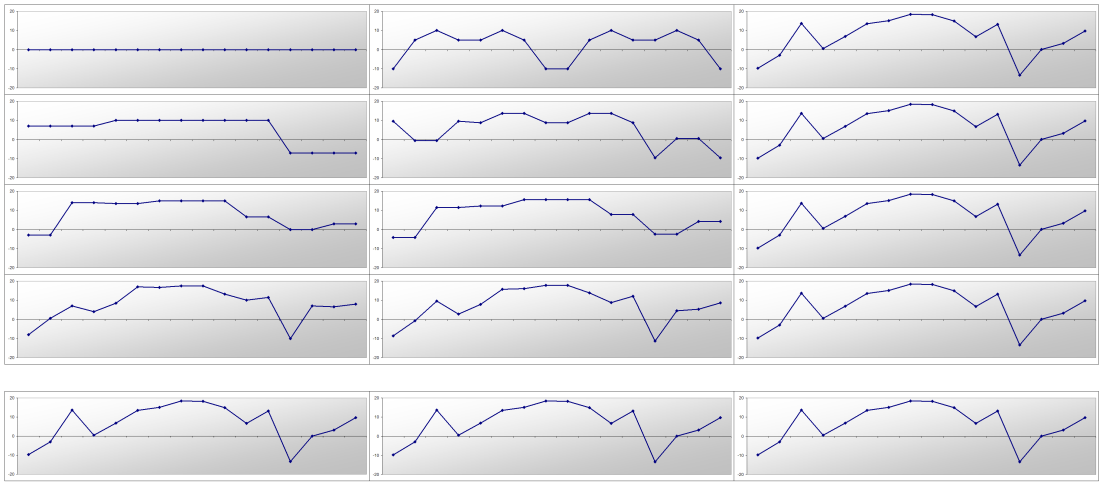

Figure 1.3 Example of iterative applying contractive mappings on 3 different initial sequences.

The aim of this work is to use these principles to encode time series with set of contractive mappings $f_1, f_2, \ldots f_m : \mathbb{R}^n \to \mathbb{R}^n$, so the function $f_1 + f_2 + \ldots + f_m$ has its fixed point similar to this time series and use these mappings to compress time series. Casually we want to split given time series into smaller subsequences and for each of these we need to find longer similar subsequence and contractive mapping that will describe this similarity. Number of these subsequences should be way smaller than length of original time series and so the space needed to encode set of transformations for these subsequences should be smaller than space needed to encode original time series.

1.3 Document organization

In second chapter brief description of methods used to process time series is given. Both lossless and lossy methods are presented. Similar principles were used to improve presented base fractal compression algorithm.

Sequential introduction to base compression technique, which is based on fractal compression method for pictures, is given in chapter 3 and in next chapter are discussed main subtasks and chosen techniques for this compression algorithm.

Fifth chapter describes error cumulation problem that affects obtained error in decompressed time series. Options which may solve this problem or at least reduce its consequences are described here and extended compression algorithms which use presented principles are introduced too.

In chapter 6 approach of processing given time series from bottom to the top using fractal compression principles is presented. Differences from top-down processing are discussed and description of bottom-up algorithm is given.

In next chapter are presented some performed experiments. Comparisons of output encoding techniques, comparisons of presented compression algorithms and comparisons with another compression methods are included in this chapter.

Then conclusion of this thesis is given and in the appendix brief information about content of included CD is listed.

2 RELATED WORKS

Nowadays many different approaches are used when processing time series data. In following sections will be introduced some time series compression techniques which are mainly used in data mining. In the last section common compression methods are listed with used compression algorithms.

2.1 Haar wavelet transformation

Haar wavelet transformation is the simplest of wavelet transformations. It groups values of the input time series into pairs of neighbours and for each pair the difference between its values is calculated and stored as a coefficient. New array of average values in these pairs is created and the process is recursively repeated until only single value (*root value*) remains. The root value with all coefficients is the information that is enough to reconstruct original sequence. It is expected that to store the root value with coefficients will take less space than to store original values of the time series. Wavelet coefficients could (and often also are) be encoded using some entropy encoding algorithm to decrease the size of stored data. Entropy encoding means lossless data compression method and as an example *Huffman coding* or *arithmetic coding* could be mentioned.

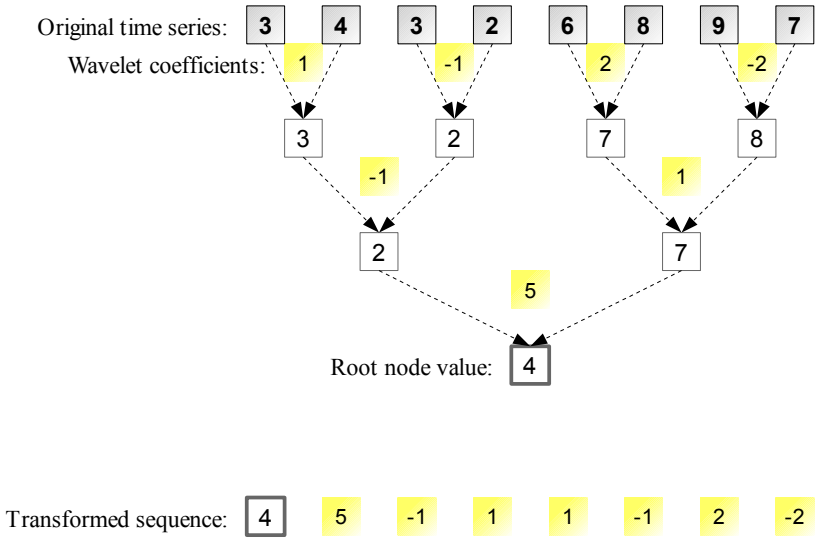

Figure 2.1 Illustration of Haar wavelet transformation on simple time series.

This method could be used both for lossless or lossy compression. To change basic algorithm to perform lossy compression is enough to cut the subtrees where desired accuracy was already achieved and instead of coefficients in this subtree, only some special symbol needs to be stored.

The advantages of Haar wavelet transformation are that it is simple, fast, memory efficient and exactly reversible. On the other hand if differences between values in the pairs are high then also transformation coefficients will be high and storing them does not need to result in data reduction.

2.2 Piecewise linear representation

Using piecewise linear representation given time series is approximated with some straight lines (segments). Number of segments is typically much smaller than length of the original time series and that means that less space is needed to store the data and also operations with the time series are more efficient. The procedures that create piecewise linear representation from given time series are called segmentation methods.

The segmentation algorithms could work with all values of time series (*batch* algorithms) or they could process time series values sequentially (*on-line* algorithms). The segmentation algorithms work with some stopping condition which describes some requested properties of compressed time series. Sometimes it is desired to split given time series into *K* segments or found the best set of segments so that in each segment maximal difference will not be higher than some specified value. Segment lines are usually calculated using linear regression.

There are these 3 categories of segmentation algorithms:

- *Sliding windows*: Time series is processed sequentially and values are added to segment from beginning of time series until some stopping condition is met. These are *on-line* algorithms.
- *Top-down*: Starts with single segment and always some segment is split in more segments until stopping condition is met. These are *batch* algorithms.
- *Bottom-up*: Starts with set of possible the smallest segments and always merges some neighbour segments into one until stopping condition is met. These are *batch* algorithms.

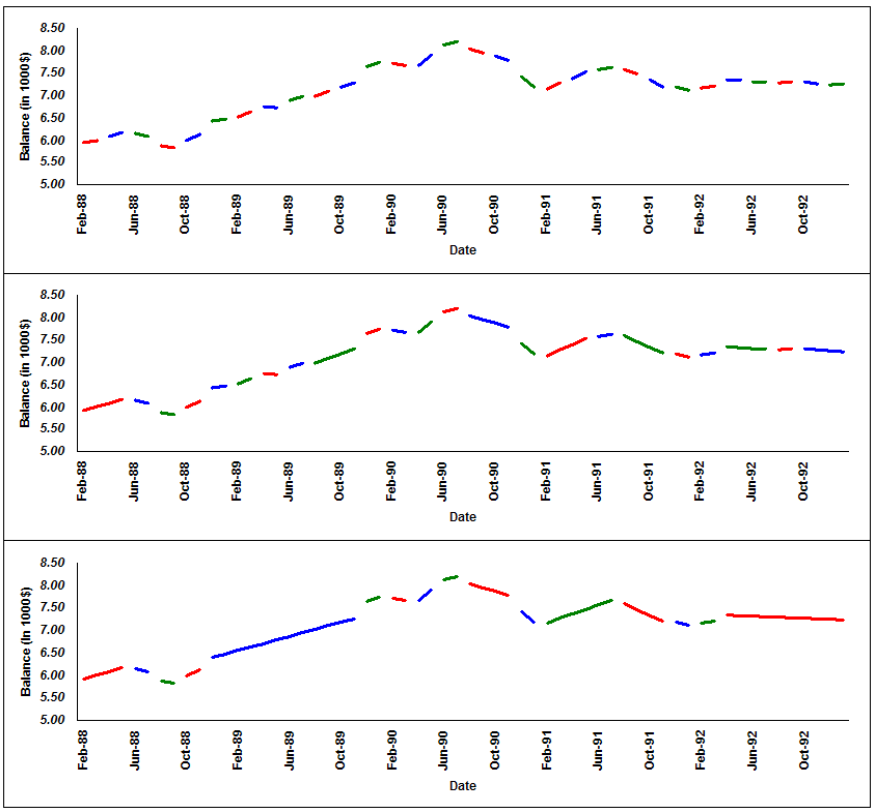

Figure 2.2 Example of the bottom-up segmentation.

In the figure 2.2 is illustrated the bottom-up segmentation algorithm on balances at the end of months in some bank from February 1988 to January 1993. [14] Linear regression was used to compute segment lines. In first graph initial state of segmentation is shown. There are 30 segments of length 2. In second graph the state after 7 segment joins is displayed and in the last graph the state after 7 more joins is shown.

According to the experimental comparison of these segmentation algorithms categories presented in [3], the *Bottom-up* category gives best results in general. On the other hand *Sliding window* algorithm gave almost always the worst performance.

As a result of attempt to create *on-line* segmentation algorithm that will have performance comparable with *bottom-up* algorithms, *SWAB (Sliding Window And Bottom-up)* segmentation algorithm was presented in [3]. Size of the sliding window is set to the number of values enough to create 5-6 segments and *bottom-up* algorithm is applied to values in the sliding window. Information about first segment is stored, the sliding window is shifted to the right and the process is repeated. Results of *SWAB* algorithm in experimental validation are essentially identical to the results of *bottom-up* algorithm.

2.3 Common lossless data compressors

Common lossless compression methods usually use a combination of compression algorithms for texts and for number sequences. Mostly they are used to archive files. They need not to be optimal solution for compressing time series because many real-world time series need not to be stored exactly.

The *bzip2* uses the *Burrows-Wheeler block-sorting* text compression algorithm combined with the *Huffman coding*. In contrast to many other lossless compression methods *bzip2* could recover undamaged parts of damaged compressed files. [17]

The *rar* compression is based on the *Lempel-Ziv (LZ) compression algorithm* and *Prediction by Partial Matching (PPM)* compression. [18]

The *zip* compression method uses combination of the *LZ77* algorithm and the *Huffman coding*, which is generally referred as *Deflate algorithm*. [19]

Each mentioned compression method uses combination of some *dictionary compression* technique and some *statistical* or *entropy data compression* method.

3 PROBLEM ANALYSIS

As mentioned in the introduction, the main task is to transform given time series into set of contractive mappings, so that the attractor of these mappings will be the original time series or at least similar to the original time series. The set of these mappings should be able to be encoded into smaller space than is needed to encode the original time series. Then we can store description of these mappings instead of storing original time series and use this principles to compress time series.

3.1 Problem description

In general for each value a_i of given time series of length *n* we are looking for some mapping $f_i : \mathbb{R}^n \to \mathbb{R}$, so that mapping $F = (f_1, f_2, \ldots, f_n) : \mathbb{R}^n \to \mathbb{R}^n$ is contraction on $\mathbb{R}^n$ and it has a fixed point equal or similar to original time series. The simplest set of such as mappings could be obtained when we set $f_i(x) = a_i$ for $1 \leq i \leq n$. Then we get *n* constant mappings. The mapping *F* will then have fixed point equal to original time series but space needed to encode these mappings will be the same as space needed to encode values of the time series.

To decrease space needed to encode it, we must use at least some functions more than once. This technique could be reasonably used only for time series, that contain the same values more than once, but could not be used for time series with unique values, because single function will always have only one fixed point. In this case we will still need to describe the time series with *n* mappings. It is desired to use some type of non-constant mappings and the constant mappings should be used only as some backdoor when using a non-constant mapping will not satisfy quality requirements.

It is also not enough to use non-constant mappings, because when some mapping will be applied to the same initial sequence, it will always lead to the same fixed point value. It is necessary to be able to get different values for more items using the same mapping.

We need to end up with mappings as simple as possible because we want to use them to compress original time series. So only some members of the input time series should be used to calculate corresponding value in the mapping. Basic idea is to split given time series into smaller subsequences, these will be referred as *ranges* [2], and for each range we need to find another subsequence of processed time series, called *domain* [2], and mapping that will transform domain values into sequence similar to the range. As an example we can imagine that each basic mapping uses only corresponding two values from domain to calculate single value of range. For range of length *n* we should look for domain of length *2n* and corresponding mapping, which will use *(2i - 1)-th* and *(2i)-th* value of domain to calculate *i-th* range value. Illustration for range of length *4* is shown in the figure 3.1. Except information about the mapping, we need to store also information about range position, range length and domain position. As we need to describe the whole time series with ranges, it is enough to store some flag of range length and the range position could be derived from number of already processed values.

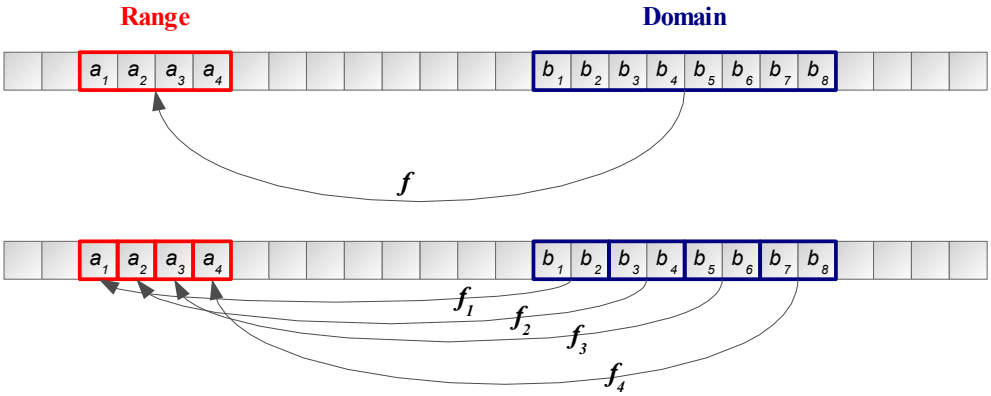

Figure 3.1 Illustration of mapping between domain and range.

3.2 Transformation choice

We need to choose which transformations will be used to describe the similarities between corresponding ranges and domains. The objective is to find some trade-off in the middle of these two options:

- *Perfect similarity description*, which will mean that more transformation types should be needed to be processed when looking for the best (or at least sufficient) transformation with corresponding domain. It will also lead to bigger space needs, when encoding these transformations in output. Realization of this option is not possible in sufficient time for general time series.
- *Small encode space for transformation*, which should mean that some better similarities won't be found and it will result in smaller fidelity of decompressed time series or it could lead to using small ranges and then the size of output could be bigger in result.

The decision was to start with simple type of transformations - linear (affine) transformations. Scaling factor between ranges and domains will be fixed to *0.5*, so domains will be twice the size of ranges. For ranges of length 4 possible transformations will look like:

$$f(x)=\begin{pmatrix} a & a & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a & a & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a & a & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a & a \end{pmatrix} x + \begin{pmatrix} b \\ b \\ b \\ b \end{pmatrix}$$

It will simply group domain values into pairs and then apply simple linear transformation $a \cdot x + b$ to the sum of each pair of domain values. To be sure that final transformation will be contraction, we will work only with the transformations those linear factor $a \in \langle -1, 1 \rangle$.

Assume that original time series contains only non-negative integer values that can be encoded each with *d* bits, so each value is in range $\langle 0, 2^d - 1 \rangle$. This means that the sum of any domain pair is in range $\langle 0, 2^{d+1} - 1 \rangle$. To get the best accuracy we need to

be able to assign value $\frac{1}{2^{d+1}-1}$ to parameter *a*. So we should encode parameter *a* with $d+1$ bits for its decimal part and a single bit should be used for the sign.

When any domain value is multiplied with value from interval $(-1,1)$, result value will fit into $(-2^{d+1}+1, 2^{d+1}-1)$ interval. Processed time series, and also processed range contains only values from interval $\langle 0, 2^d-1 \rangle$, so we need to be able to move the obtained value $a \cdot x$ precisely enough into this interval with parameter *b*. Also here $d+1$ bits should be used to encode integer part of parameter *b* and a single bit to store the sign. It is because when parameter *a* is positive or zero, then we need to describe some move in interval $\langle 0, 2^{d+1}-1 \rangle$. And when parameter *a* is negative then we first add value $2^{d+1}-1$ to the result and then we need to describe some move in interval $\langle 0, 2^{d+1}-1 \rangle$ as in previous case.

Number of bits that should be used to encode decimal part of parameter *b* can not be calculated. But when we will process only positive integers, some small amount should be enough. With bigger number of bits we will get better accuracy, but while we work only with integers it means also useless wasting of space. On the other hand very small values can cause that some good enough transformations will not be used, because we will not be able to encode them precisely enough to satisfy given error requirements. Experimental tests gave the best results for 3 or 4 bits used to encode decimal part of parameter *b*.

So when we will process time series with integer values from interval $\langle 0, 2^d-1 \rangle$, we should be prepared that to encode a single transformation we will need $(d+2)+(d+5)$ bits. To this amount we need to add size of information needed to encode position and length of the range and also position of domain. This means that is useless to describe the ranges with less than 4 values like this. When no good-enough transformation could be used for range of length 4, then it is better to store values of this range instead of storing information about some transformation.

3.3 Splitting time series into ranges

We want to use some deterministic splitting into ranges that will need only small amount of space to encode it. When we allow overlapping of ranges, we will need to store both position and length for each range. This seems to be expensive, because while only contractive mappings are used, it is enough to calculate each value using single mapping.

We will work only with non-overlapping ranges and that means it is enough to store only sequence of range lengths.

Next question could be *How to process the ranges?* Obviously chance that we will find sufficient domain and transformation for long range is very small and it should be considered as time wasting.

So we should start processing ranges of reasonable bigger length and in case when sufficient domain and transformation were not found, we will take smaller range and repeat the action until sufficient domain and transformation will be found or encode the smallest possible range directly with its values. Here we can decrease length of processed range by some constant or we can split the range into more ranges of the same length. Disadvantage of this strategy is that for each unsuccessful attempt to

find domain and transformation for some range, whole space of possible domains must be tested and that could cost a big amount of time during processing.

The second option is to process the smallest ranges first and when sufficient domain and transformation were found then try to increase length of range and try again. This looks very similar to previous option but the main difference and also advantage is that we could stop browsing domains when we found sufficient one and try to process some longer range. If no sufficient domain and transformation were found for this longer range then we will return to previous smaller range and find the best domain and transformation for it, which will be found because we have already found some sufficient domain and transformation.

3.4 Searching best domain

By phrase looking for best domain and transformation according to some range is meant to browse the space of domains determined by processed time series and calculate the best transformation, which transforms domain into the sequence that is the same as processed range or at least to the nearest possible sequence. Number of domains to process could be very big and especially when some long range is processed, chance to find good enough domain is very small and many domains are processed uselessly. To reduce this number we should categorize domains according to some similarity equivalency and then process only those domains which are similar to processed range.

More formally we need to define an equivalency relation for sequences of length n or $\mathbb{R}^n$, which will mean that two sequences are similar in some way and then we could split domains into equivalency classes of this equivalency. We must be able to determine in which equivalency class are domains that probably will be better for processed range. The perfect equivalency will reasonable decrease number of domains that need to be browsed for one range and also decrease chance that the best domain will belong to another equivalency class. Find perfect similarity equivalency is impossible, so some compromise equivalency between an *identity* (only identical sequences are considered as similar) and a *trivial equivalency* (all sequences are considered as similar) needs to be defined. Then we will need to browse only through the domains in one similarity class and the best domain of this class could be qualified as the best domain for processed range and on the other hand when no sufficient domain will be found, we could consider that we will not find it in the other similarity classes too.

We should use some kind of monotonic similarity to split domains into classes. We decided to use only linear transformations and for range of length n, only domains of length $2n$ will be browsed. Domain values are grouped into n pairs, where values in single pair are added up and considered as one domain value. When we apply a linear transformation to all values of some domain, the monotony of the result sequence will be the same, or inverse when linear factor of applied transformation is negative. When linear factor is equal to *0* then the result sequence will be constant. This case is not interesting because the result will be always the same when applying constant linear transformation to any domain.

Lets start with classification of smaller domains. To classify some domain with 4 values we could simply compare the neighbour values and create binary number for the class. Add *0* when left value is smaller than right value, otherwise add *1*. With this simple procedure we could split all domains of length 4 into classes *000*, *001*, ...,

111 or 0 to 7 in decimal. Example of simple domains of these classes is given in figure 3.2. We should notice that when we apply linear transformation with negative linear factor, we will get the sequence where every inequality operator between neighbour values of result sequence will be inverse to corresponding operator used to compare domain values. For example a domain of class *000* will create a sequence of class *111*. This could lead to idea to join these inverse classes and use only 4 classes *000/111*, *001/110*, *010/101*, *011/100* of domains. Application of any linear transformation on some domain will then create sequence of the same class and we could not get sequence of some class with no transformation applied to domain from different similarity class.

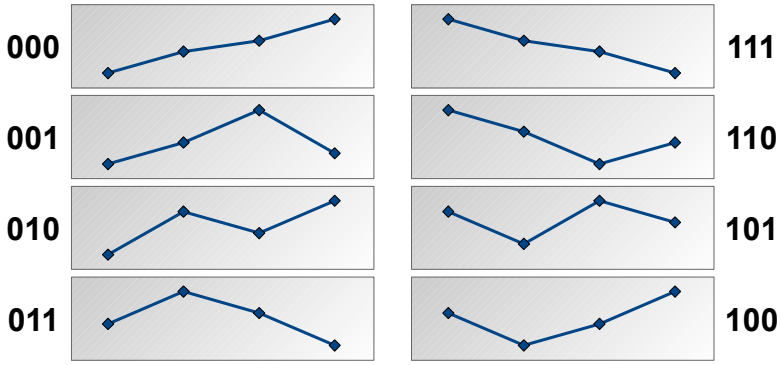

Figure 3.2 Example of domains of length 4 for each class.

Classification of longer domains can not be done this way. Because when domain length is fixed to *n* then number of classes will be 2^{n-1} (or 2^{n-2} when we group the inverse classes) and this will lead to exponential number of classes. Also single similarity class will contain only small number of domains and chance, that the best domain for processed range will be found in corresponding class, will be then very small. Better solution is to convert values of longer domain into sequence of length 4 and classify this sequence using classifier for domains of length 4. According to this we will have the same number of classes for domains of any length and it will be sufficient to define some simple classification procedure for domains of small length.

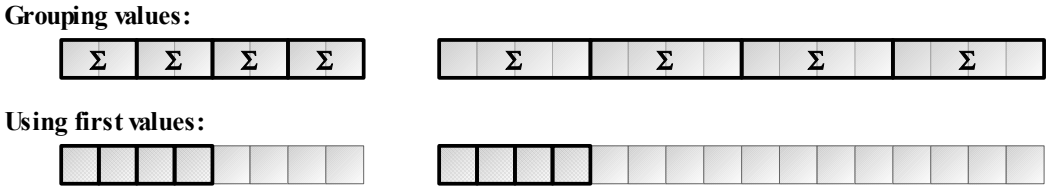

Figure 3.3 Illustration of how longer domains could be classified using classifier for short domains.

First idea for converting longer sequence to smaller is to group corresponding neighbour values and replace the group with sum of its members. But this procedure will be simple only when it will be applied to domains of lengths which are multiples of *4*. Advantage of this technique is that number of similarity class is computed using all values in the domain.

Another idea is to simply cut first domain values and classify whole domain according to them. Negative aspect of this classification is that we use only some first values and it is very probably that the best domain will fall in some another class, because first part of domain need not to fit into right class but the last values

could be a perfect match. This method should be used only to determine whether some sufficient domain for processed range exists and then all domains should be browsed to find the best one.

3.5 Transformation computing

Assume that we have range of length *n* and some domain, in which we have grouped its values (for example by sum of neighbour values) to the sequence of *n* values. Lets denominate range values as $r_1, r_2, \ldots, r_n$ and grouped domain values as $d_1, d_2, \ldots, d_n$. We are looking for the linear transformation $t(x)=ax+b$ that will be contractive and sequences $(r_1, r_2, \ldots, r_n)$ and $(t(d_1), t(d_2), \ldots, t(d_n))$ will be as similar as possible. Formally we want to find such as transformation, for which *distance* between these sequences will be minimized.

To measure distance between two sequences we will use the *Root Mean Square (RMS)* metric which defines distance between two members $x, y \in \mathbb{R}^n$ as:

$$d_{RMS}(x, y)=\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-y_i)^2}$$

So for given two sequences of length *n* (range and domain) we want to get linear transformation which minimizes the distance between range and transformed domain. It is enough to minimize the sum expression under the square root and that will minimize also the distance. We need to find linear factor *a* and constant factor *b* of the linear transformation that minimize this function:

$$f(a, b)=\sum_{i=1}^{n}(r_i-a\,d_i-b)^2$$

f is function with 2 parameters and according to the analysis it has candidates for local extremes in values where both partial derivations are equal to *0*. We need to solve two linear equations to get these candidates and determine global minimum from them. Partial derivations of the function *f* are:

$$\frac{\partial f}{\partial a}(a, b)=\sum_{i=1}^{n}-2\,d_i(r_i-a\,d_i-b)$$

$$\frac{\partial f}{\partial b}(a, b)=\sum_{i=1}^{n}-2(r_i-a\,d_i-b)$$

When we put these partial derivations to be equal to *0* and solve those linear equations, we will get these expressions for linear and constant factor of the best transformation for processed range and domain:

$$a=\frac{\frac{1}{n}\left(\sum_{i=1}^{n} r_i\right)\left(\sum_{i=1}^{n} d_i\right)-\sum_{i=1}^{n} r_i d_i}{\frac{1}{n}\left(\sum_{i=1}^{n} d_i\right)^2-\sum_{i=1}^{n} d_i^2}$$

$$b=\frac{\sum_{i=1}^{n} r_i-a\sum_{i=1}^{n} d_i}{n}$$

When fraction denominator for linear factor is zero then these equations have not any solution, or it has infinite set of solutions when the numerator is also zero. Otherwise it has exactly one solution.

We must also notice that even if we find transformation which perfectly transforms domain into sequence very similar to the range values, this transformation need not to be a contraction. We could not use any non-contractive transformation because then composited result transformation could be non-contractive transformation also and we will not be able to generate time series using our set of transformations.

3.6 Cumulative computations

As we can see in the expressions to calculate factors of linear transformation, many sums of range or domain values need to be calculated. When we consider that any single domain will be processed at least within all ranges of the same similarity class, calculation of these sums again and again will slow down processing of given time series. For both the array of range values and the array of grouped domain values, cumulative arrays should be used to store cumulative sums of first i values. For domain array we could also calculate cumulative array with sum of squares of first i values. This could be done at the beginning of processing time series and then to obtain the sum of range values at position p and of length l, we will simply calculate the difference $a[p+l]-a[p]$ using corresponding cumulative array a.

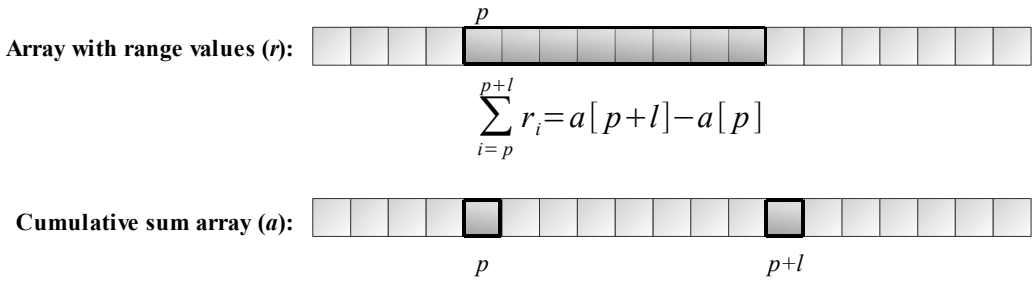

Figure 3.4 Illustration of using cumulative array.

At the beginning of processing time series we do not know, how time series will be split into ranges, so we need only to create array of cumulative sum values corresponding to the array of range values. Then when processing some particular range, sum is calculated using this cumulative array. We also need to create cumulative arrays for domain values. One for cumulating sum values and one for cumulating sum of value squares. If we know that we will work only with the domains of some lengths (for example only with lengths 16, 8 and 4) we could compute sum of domain values and sum of domain values squares also at the beginning and store these values in some domain structure for each domain. Then we should just browsing these structures to find the best domain for each range.

Even when we precalculate the cumulative arrays for arrays of range and domain values we still need to calculate the scalar product for each processed range and domain. But for each pair of range and domain, the transformation will be computed maximally once, so here no computations will be repeated redundantly.

3.7 Error measuring

When looking for the best domain with transformation for processed range, we need some method to measure an approximation quality and according to this value we can control browsing through domains. We should be able to find the best possible approximation and determine if this approximation is sufficient.

There are many metrics on the space $\mathbb{R}^n$ that could be used to calculate distance between two sequences of length *n* and these can be used to measure error. In first place we should use the *Root Mean Square* metric, which is also used to calculate coefficients of the best transformation for processed range and domain. The root mean squared error value *d* ensures that approximated values differs on average *d* units from original values and not much more than *d* units.

Another metric that could be used, simply calculates average value of differences between the corresponding values. It is similar to *RMS* metric but using this metric we can get low error values also when some values (or one value) are more different. For example error between sequences $(1,2,3,4)$ and $(1,2,3,8)$ will be 1 but difference between fourth values is 4. So using this metric some values could be more dispersed and still sufficient error value could be obtained. The formula to compute distance using this metric is:

$$d_{MEAN}(x, y) = \frac{1}{n}\sum_{i=1}^{n} |x_i - y_i|$$

A metric, that will sufficiently handle the dispersed values, returns as distance between two sequences maximal difference between the corresponding values. This metric ensures that approximated values will fall inside constantly wide zone around original time series. The formula to calculate difference between two sequences using this metric is:

$$d_{MAX}(x, y) = max\{|x_i - y_i|; 0 \le i \le n\}$$

3.8 Generating time series from stored transformations

This is very simple procedure. We just need to read the transformations and store them in structures with information about range position, range length and position of domain. Then apply this set of transformations in some iterations on any sequence of length as original time series. These contractive transformations converge very fast and usually 10 iterations is enough to get good attractor approximation of these transformations.

To apply each transformation only if it is still needed we could compare the pre and post state of processed range and if difference between them is lower than some little constant, we need not to apply this transformation again in next iterations. This saves a little time in decompression and also the transformations will not be applied uselessly.

4 BASE SOLUTION

At the beginning the goal was to implement some simple algorithm, which will perform some simple splitting of given time series into ranges, then will find the best domain with corresponding transformation for each range and outputs this information in form that could be given as an input to another algorithm and this will recalculate compressed time series. These options and properties were chosen for this algorithm:

- Algorithm expects sequence of positive integers as input.
- It works only with ranges of length $4, 8, \ldots, 2^d$. At the beginning given time series is split into ranges of the biggest length and these are processed one by one. Either some good enough domain with corresponding transformation is found or the range is split into two smaller ranges of half length and these are processed in the same way.
- For range of length *n* only the domains of length *2n* will be browsed. Domain values are created from original time series by sum up the neighbour values.
- To simplify browsing through domains are these divided into similarity classes. For each possible domain is created an object with domain position, sum of its values and sum of squares of its values. These objects are stored according to the length and similarity class. When looking for the best domain during processing some range, only domains of the same similarity class are browsed. Both sums are calculated using corresponding cumulative array as described in section 3.6.
- As mappings between domains and ranges are used only contractive linear transformations. If no sufficient domain was found for range of length 4 then the exact range values are stored. Used transformations are described in detail in section 3.2 and calculation of linear transformation coefficients could be found in section 3.5.
- Quality of an approximation during processing some range could be measured using one of three metrics (d_{RMS}, d_{MEAN} or d_{MAX}) listed in section 3.7. One of these metrics is selected and sufficient maximal error value is given as compression parameter.
- Information about domain position and used transformation are stored to the output file immediately after the range is processed.

Default compression algorithm:

Input: time series *ts*, maximal error *err*, output binary file *outFile*

1. Append information about length of *ts* into *outFile*.
2. Split *ts* into ranges of maximal length and store these ranges in stack *rStack* in reverse order. {At the top will be first range.}
3. **WHILE** *rStack* is not empty
4. Pop range from *rStack* into *r*.
5. **IF** length of *r* is smaller than minimal range length **THEN**
6. Append length and exact values of range *r* to *outFile*.
7. **ELSE**
8. Find best domain *d* and corresponding transformation *t* for range *r*.
9. **IF** *t* and *d* generates range *r* with sufficient accuracy **THEN**
10. Append information about range *r* length, *t* and *d* into *outFile*.
11. **ELSE**
12. Split range *r* into ranges *r1* and *r2* of half length.
13. Push ranges *r2* and *r1* to the *rStack*. {*r1* will be next processed range.}
14. **END IF**
15. **END IF**
16. **END WHILE**

4.1 Base algorithm decomposition

The basic compression and decompression routines were encapsulated into single class, which provides also methods that allow to specify user preferences for compression. All processes were split into small consistent actions, so for derived classes is enough to override some methods to perform some extra tasks to achieve better approximation or to perform some tasks in different way.

The lowermost, in the hierarchy of compression classes, is abstract class *ICompressionInfo* which only defines an interface for two basic methods

- Compress method, which converts given time series into set of contractive transformations according to specified maximal error and store these transformations into given output binary file.
- Decompress method, which is inverse to compress method. It reads set of transformations from given input binary file and applies them in few iterations to initial sequence to recalculate decompressed time series.

From this class is derived another abstract class *BaseCompressionInfo*. This class contains common members used in compression or decompression and also defines methods that initializes them or work with them. To these members belongs:

- Input and output binary file.
- Array of range values with corresponding cumulative array for sum.
- Classifier for domains or ranges.
- Error computer that calculates difference between two sequences.

This class defines also the skeletons for compress and decompress methods. Each method is split into smaller logic-consistent tasks. Compression process (method *Compress*) consists of these basic steps:

- Compression initialization. Here information derived from processed time series should be calculated and structures for domains should be initialized.
- Writing header.
- Current compression logic.
- Cleaning and releasing used temporal structures.

Skeleton for decompression (*Decompress* method) looks like:

- Reading header.
- Decompression initialization.
- Decompression routine.
- Cleaning and releasing used data structures.

And finally the simplest versions of the routines for compression and decompression are implemented in class *DefaultCompressionInfo* derived from class *BaseCompressionInfo*. Compression method implements base compression algorithm described in previous section.

When processing particular ranges, instead of writing some series of bits for each range length, after some range is processed then *1-bit* is written when sufficient domain and transformation were found for it and information about this transformation and domain position are written too. If the range needs to be split then *0-bit* is written to output file and new two ranges are processed in the same way. If no sufficient domain and transformation were found for range of length 4, then *0-bit* is written to output file followed by the values of this range.

In top of figure 4.1 is displayed an example of processing first 32 values of some time series. ***X*** mark means that range was not processed successfully (range marked with *0* flag) and need to be split into smaller ranges. ***OK*** marks represent successfully processed ranges and these are marked with *1* flag. Empty cells at the bottom represents single values of time series that need to be output. And at the bottom is displayed output information. *0/1* cells represent single bits, ***T*** is for transformation with domain position and ***V*** is for single range value.

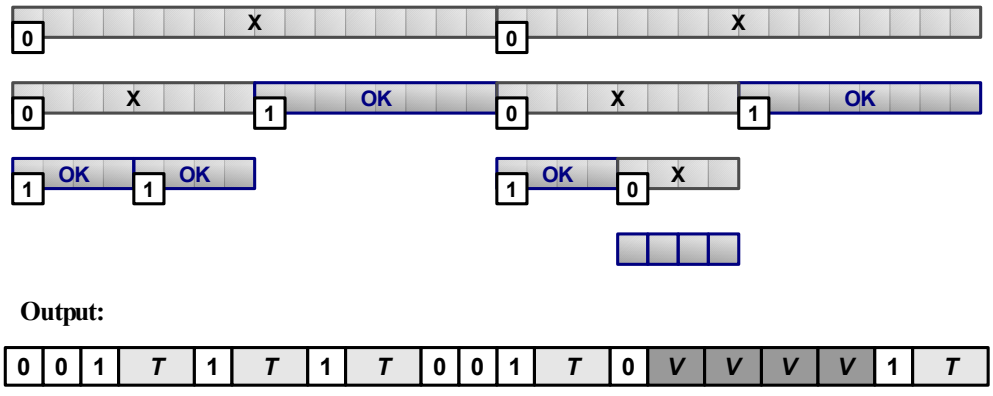

Figure 4.1 Illustration of processing ranges and corresponding output.

When reading information about particular ranges in decompression procedure, range of maximal length is expected first. If *1-bit* is read then information about range follows. Otherwise information about two ranges of half length is expected. If the processing goes down to the range of length 4 and *0-bit* follows then 4 range values are read.

4.2 Domain container

In the compression algorithm we need to browse through domains several times. For this reason is better to calculate the information about particular domains at the beginning and store them in some structure, from which domains could be accessible for browsing. For each domain its values, sum and sum of squares are stored and this information could be simply used for computation of the best corresponding linear transformation.

For purposes of base compression algorithm only some simple structure to handle operations with domains is needed. For each length and similarity class one linked list of corresponding domain structures is held. These features are provided by *DomainContainer* class, which is also the base class for other domain containers used in derived compression algorithms.

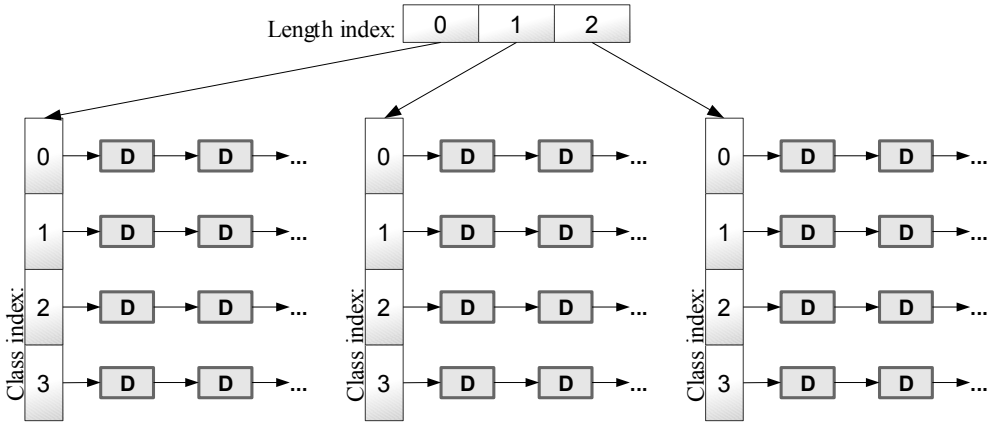

Figure 4.2 Storing domain structures using *DomainContainer*.

4.3 Classifiers

All compression algorithms use domain and range similarity classifier to split the set of domains into smaller classes. These domains are browsed and used to calculate corresponding transformation only when some range of the same similarity class is processed. The number of different similarity classes is needed to initialize correctly domain container, so each classifier must provide methods which returns:

- Total number of similarity classes.
- Index of similarity class for given domain or range.

In base compression routine all classifiers described in section 3.4 could be used. Each one is implemented as a separate class derived from *IClassificator* abstract class which defines interface for classifiers.

4.4 Error computers

Requirement for sufficient accuracy for processed range and found domain with corresponding transformation could be evaluated using different metrics. During the compression we must be able to calculate some error value, which can be used to determine the best domain for particular range and then we need to be able to check if this domain and transformation describe this range with sufficient accuracy. For these purposes an abstract class *IErrorComputer* was written and it defines methods which:

- Compute error (difference) between two sequences.
- Determine if obtained error value is sufficient according to the specified maximal error.

Single error computer class was created for each mentioned metric for sequences presented in sections 3.5 and 3.7.

4.5 Decompression algorithm

Reverse algorithm corresponding to described compression algorithm was created too. It reads information about the transformations for particular ranges and then in few iterations it simply applies this set of transformations on zero-sequence of the same length as compressed time series. Usually some small number (around 10) of iterations is needed to get close enough to the attractor.

Default decompression algorithm:
Input: input binary file *inFile*, number of iterations *itNum*
Output: decompressed time series *ts*

1. Read length of time series from *inFile* and store it into *len*.
2. Initialize zero-sequence *ts* of length *len*.
3. Read information about ranges from *inFile* and store corresponding transformations in list *trList*.
4. *itNum* times apply transformations in *trList* to sequence *ts*.
5. **RETURN** *ts*

Decompression algorithm could be simply modified, so it will evaluate only the transformations which still change generated sequence in reasonable way. If the difference of processed range before and after the transformation is applied, is smaller than some small ε, this transformation should be removed from the list of transformations and need not to be evaluated again.

In following figure are displayed first 200 values during decompression process. As an input was used ECG data compressed at 99.5% accuracy. Generated sequence changes a lot during first few iterations, when transformations approximate to their fixed points. In displayed example total number of used iterations in decompression was 18 but after the eighth iteration, applied transformations change the sequence only in very small rate.

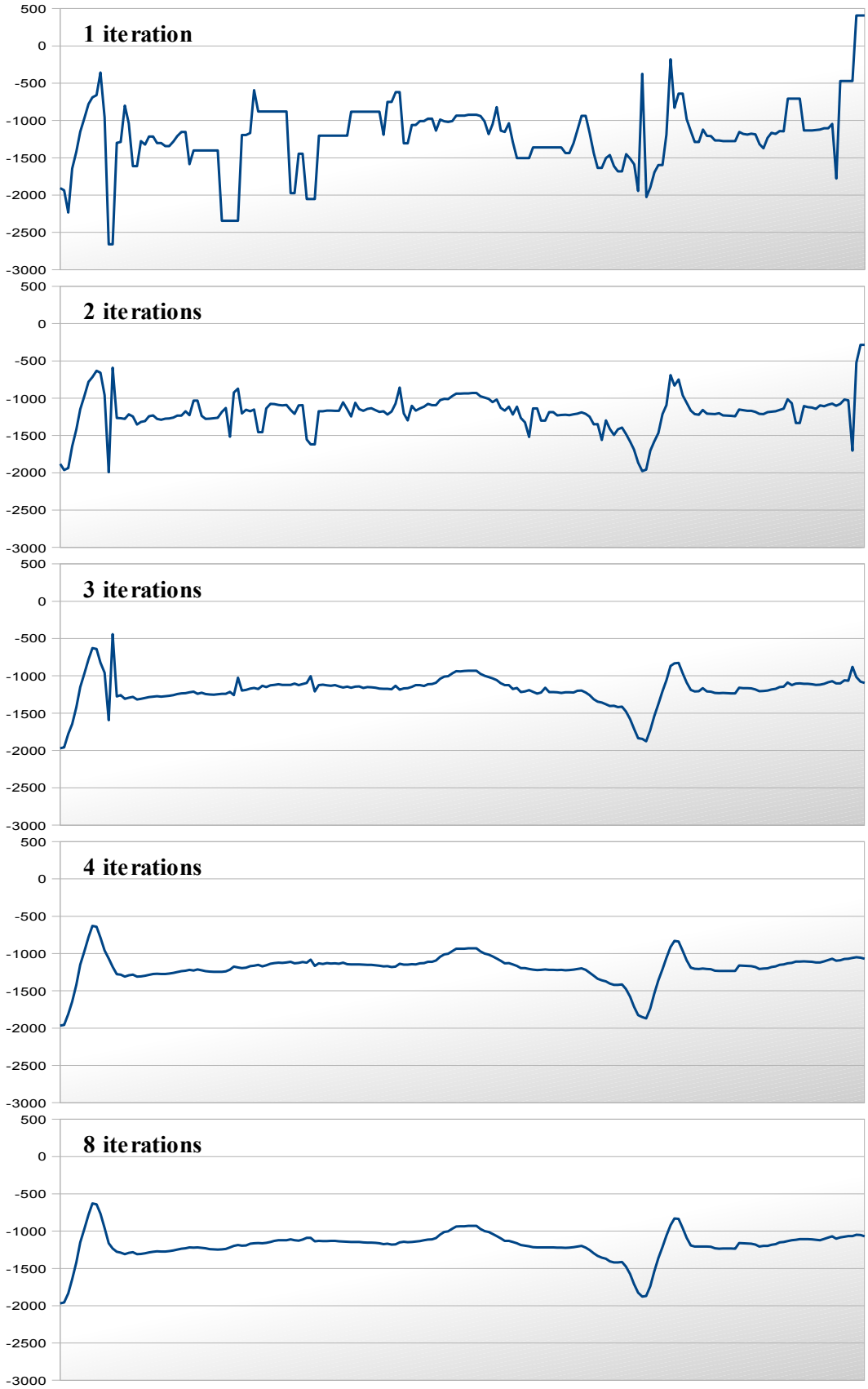

Figure 4.3 Illustration of how generated sequence is changed during decompression.

5 ALTERNATIVE IMPROVEMENTS

Even when specified maximal error is used in choosing domains and transformations for particular ranges, this error is not guaranteed. Usually the maximal error condition is fulfilled globally for the whole time series when *RMS* or *MEAN* metric is selected, but there may be some ranges for which this condition is not fulfilled. When *MAX* metric is used then if this condition is not fulfilled local for some ranges, it could not be fulfilled also globally.

When processing longer time series then almost always there will be some ranges that will not fulfil specified error condition. During decompression process the time series values are computed by applying set of contractive transformations to it and both domain values and range values are taken from the generated time series. Each value of time series belongs to one and only range, so each value is calculated by one and only transformation. On the other side single time series value could belong to more domains. Here *error cumulation* could occur. Some single value is computed and difference from corresponding original value is d_1 and this value belongs to domain that is used to calculate another value. During compression when second range was processed, some sufficient error value d_2 was obtained. But in decompression also the error d_1 from calculation of its domain values will affect the final error for value in the second range. Like this error could be cumulated through some long chain of dependent transformations. Unfortunately we could not determinable detect these cumulations during compression process without simulating the decompression process.

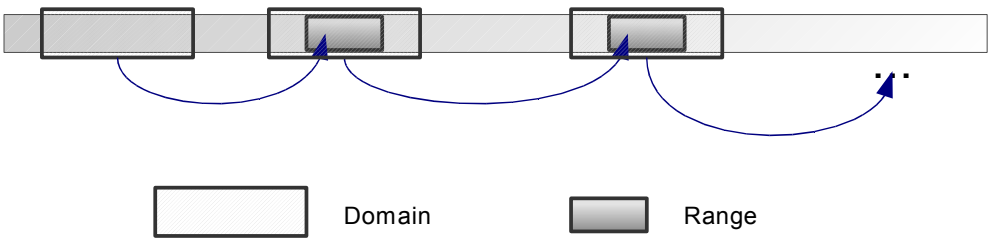

Figure 5.1 Error cumulation illustration.

An error cumulation occurs when ranges are generated using long chain of dependent transformations. In following sections some techniques to reduce error cumulation are presented. First idea is based on attempts to minimize the error passed through these chains with actualization of particular domain values and structures. Second idea tries to prevent from creating long chains of dependent transformations by dynamically changing the set of allowed domains for particular ranges.

5.1 Smoothing domain values

First idea to fix this problem or at least minimize its consequences could be to topologically order all ranges after compression process and then realize some corrective action for each range in this order. In first spots have to be ranges that are not generated using any transformation but their values are stored in exact form.

Then on the next spots will be ranges that are calculated by some transformation applied to domain which consists only of values from previous ranges. As corrective action could be taken recalculation of linear transformation coefficients according to current domain values, which will be used during decompression process.

But ranges could not always be ordered in specified way. Only when many values of time series were stored as exact values and other ranges were generated using domains of these exact values or of already generated values. According to chaotic character of the range-domain pairs, there will almost always be the cycle in this oriented dependency graph. So we could order ranges according to almost topological order, where minimal number of range values will be generated using values from forward ranges. Then for ranges that depends only on values from previous ranges is enough to perform correcting action once and for the other ranges correcting action could be done while some error decrease is obtained.

If all range values are calculated from domains using transformations then we will not be able to find range that is not generated using values from some another range or ranges. This situation will occur when bigger sufficient error is specified for compression, because sufficient domain and transformation will be probably found and it will not be necessary to store exact range values in the output file.

Disadvantage of this idea is also that it is processed after compression and some other domain could be meanwhile better for generating some particular range than domain that was found during compression and now is affected by cumulative error. So as better idea seems to be actualizing corresponding domain values after the range was processed in compression with expected values generated using found domain and transformation. This will assure that domains for further ranges will be searched in actualized domains.

Even when using above technique in the result some long chain of dependent transformations could occur. This could contain ranges for which domain was found in not yet processed part of time series. To avoid this situation to happen, domain values corresponding to each processed range should be recalculated when the range is processed and also transformations which depend on values of processed range should be actualized. Again coefficients of these transformations are recalculated using current domain values and the original range values. By recalculating these transformations some other domain values could get out-of-date and should be recalculated and transformations that use them should be too. Number of transformation actualization in each step should be limited, because according to their chaotic character, cyclic dependency chain could occur and actualization process could result in infinite loop.

As the result during compression we will work with current domain values, that should be used for processed ranges. This covers the „left-to-right“ transformation dependencies. And when some processed range harms domains for previously processed ranges, transformations for these ranges are updated. This covers the „right-to-left“ transformation dependencies.

These processing principles are used in the smoothing compression algorithm represented by *SmoothCompressionInfo* class. First difference is that after some range is processed, information about found transformation and domain position is not saved immediately to the output file but it is stored in temporal structures, where this information could be modified later. All these information are written to the output file at the end of compression. Second difference is that after some not

constant transformation is used to describe processed range, each transformation that depends on values of this range, will be actualized and actualization process is repeated for transformations that depends also on these actualized transformations. Each transformation is actualized maximally once in each actualization process. Transformations that need to be recalculated are stored in queue and are served first in first out.

Smoothing compression algorithm:

Input: time series *ts*, maximal error *err*, output binary file *outFile*

1. Append information about length of *ts* into *outFile*.
2. Initialize container for compression transformations *trCont*.
3. Split *ts* into ranges of maximal length and store these ranges in stack *rStack* in reverse order. {At the top will be first range.}
4. **WHILE** *rStack* is not empty
5. Pop range from *rStack* into *r*.
6. **IF** length of *r* is smaller than minimal range length **THEN**
7. Append length and exact values of range *r* to *outFile*.
8. **ELSE**
9. Find best domain *d* and corresponding transformation *t* for range *r*.
10. **IF** *t* and *d* generates range *r* with sufficient accuracy **THEN**
11. Add information about *t* and *d* into container *trCont*.
12. Actualize domain values affected by range *r*.
13. Recalculate transformations dependent on *t*.
14. **ELSE**
15. Split range *r* into ranges *r1* and *r2* of half length.
16. Push ranges *r2* and *r1* to the *rStack*. {*r1* will be next processed range.}
17. **END IF**
18. **END IF**
19. **END WHILE**
20. Append information about transformations in *trCont* into *outFile*.

In actualization of transformation are recalculated the transformation coefficients according to the original range values and recently actualized domain values. Problem here is that actualized transformation need not to be contractive. When not-actualized transformation has linear coefficient with value near to 1, it is very probably that linear coefficient will get over 1. This will lead to big error values because linear coefficients are stored only as the cut decimal parts of them. Also ranges generated by transformations that depend on these range values will result in big differences in decompression. In this case the range generated by such as transformation, should be stored using its exact values instead.

5.2 Dynamic domain container

In the basic compression algorithm domain values were precalculated before compression and for each domain simple structure was created and stored according to domain length and similarity class. Domain structures of corresponding length and class were only browsed as linked list during compression.

In the smoothing compression algorithm we need to be able to actualize domain values. The question is if better solution is to actualize domain structures in some container or to actualize only the array of domain values and cumulative arrays and create particular domain structures as needed. The second option is simpler but it does not consider similarity classification of domains, so domain structures must be created for all domains even if their similarity class is not the same as the class of processed range. First option is more complicated because the domain similarity class after actualization could be changed and this change should be considered in the container too.

In figure 5.2 is displayed the dynamic container for domains which is used in smoothing compression algorithm. It simply extends the basic domain container described in section 4.2. It provides pointer to the first domain structure in linked list corresponding to specified length and similarity class. Domains of particular length and similarity class are now stored in two-way linked list to simplify the process of moving domain between similarity classes. According to requirement of being able to actualize domain structures affected by change in some range, pointers to these domain structures are also stored according to their length and position in two dimensional array of pointers. Using this pointer array we can access the domains according to their position and actualize them according to changes in processed range.

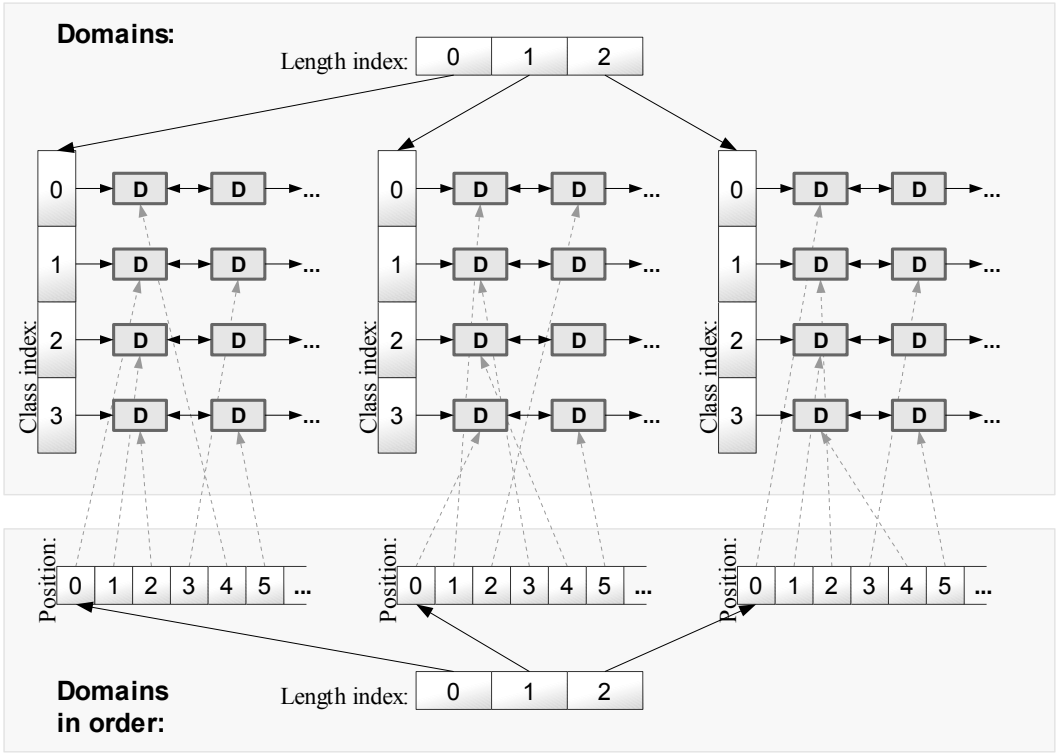

Figure 5.2 Storing domain structures in *DynamicDomainContainer*.

5.3 Transformation container

In smoothing compression algorithm the information about transformations and domain positions are not written to the output file immediately but they need to be stored in some structure. This structure has to provide these routines:

- Add and remove information about transformations.

- Search transformations according to position of corresponding domain and provide iteration through stored transformations according to domain position. This will be used when looking for the candidates of dependent transformations according to some specified range.
- Write information about stored transformations into output binary file in the same format as base compression does.

The transformations in transformation container should be stored according to position of corresponding range, so at the end of compression process this information could be output in correct order. On the other hand when looking for dependent transformations on processed range, we need also to find these dependent transformations according to their domain position. Then dependent transformations could be simply separated in two phases. First set of the candidates for dependent transformations is found and then domains of these transformations are tested for non-empty intersection with processed range. Simple illustration is given in the figure 5.3.

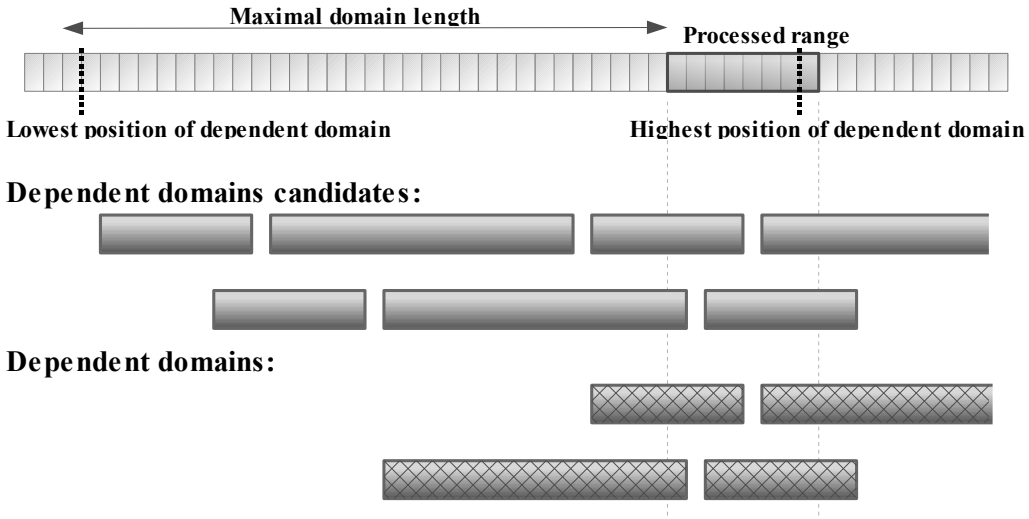

Figure 5.3 Searching for dependent transformations for range.

To write transformations into output file, basic compression process is simulated to write correctly control bits and transformations in container are iterated according to their range positions.

5.4 Sequential processing

Sequential processing of particular subsequences is technique which should avoid creating long chains of dependent transformations and error cumulation. It is based on principle that for each range only some subset of domains will be available. Affected domain structures will be actualized too, but now not after each processed range, only after some subsequence of ranges has been processed.

Compression algorithm needs two more values to be specified. Length of the basic subsequence of range values, which for simplicity needs to be multiple of maximal range length. Second parameter is length of the sliding window for domains. Only domains based on time series values from current sliding window will be available for ranges from processed subsequence. Values of these parameters are required also in decompression process so they need to be stored in the output file.

Skeleton of compression is the same as in the base algorithm but this time given time series is processed in smaller steps and each step corresponds to single subsequence of specified length. To find sufficient domains for ranges in processed sequence, only domains created from values in sliding window could be used. At the beginning sliding window is empty. Before first subsequence is processed, its values are added to sliding window, corresponding domain structures are created and stored in domain container and then ranges of this subsequence are processed. Then values in processed subsequence are recalculated in the same way as in decompression process and according to these values, corresponding domain structures are actualized too. With next subsequences the process is repeated. When sliding window is full, the oldest sequence is replaced with current sequence.

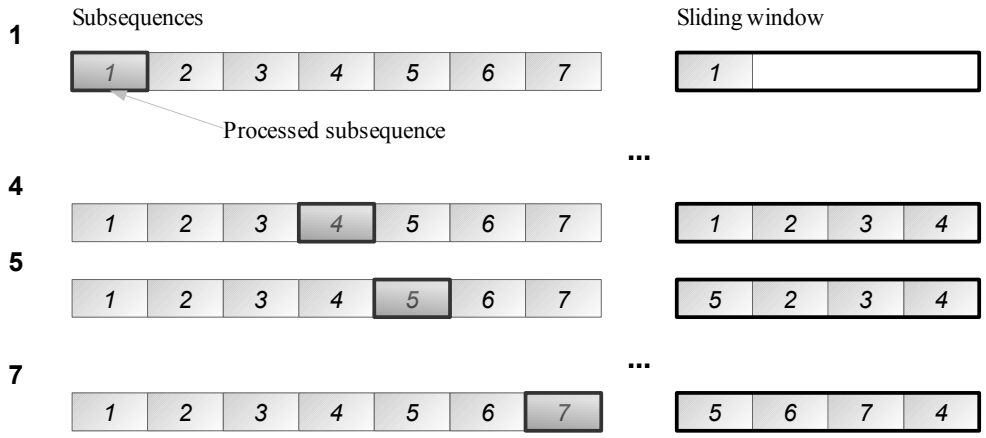

Figure 5.4 Illustration of processing subsequences of time series.

In figure 5.4 is simply illustrated sequential compression process where length of sliding window is equal to four subsequence lengths. For first subsequence only domains created from values in it could be used. Then for next processed subsequences, the set of values for domain structures is extended and during processing fourth subsequence, domains created of all four subsequences could be used and the sliding window will be full from this moment. Before processing fifth subsequence, space for new subsequence values must be made in the sliding window, so values from the first subsequence are replaced by fifth subsequence. Generally during processing any subsequence only domains created from current and previous $n-1$ subsequences could be used where n is the length of sliding window expressed in number of subsequence lengths.

This technique need not necessary minimize lengths of chains of dependent transformations, but it decrease error cumulation on these chains to subchains whose all transformation domains and ranges belong to the same subsequence. It is because subsequence occurs in sliding window first time before this subsequence is processed. So domains created from values of this subsequence could be used at first time for ranges in this subsequence and because no domain actualization is performed during processing this subsequence, error could be cumulated over dependent transformations here. If any domain from previous subsequences is used, it will have already actualized values and no further transformation could affect it, because only domains from previous subsequences could be used for ranges in that subsequence. And if some domain will be used for range in further subsequences, domain values will be actual, because after the subsequence is processed, all

corresponding domain structures will be actualized. Output information about ranges and domain positions are specified relatively to the current state of sliding window.

Sequential compression algorithm:

Input: time series *ts*, maximal error *err*, length of subsequence *subLen*, length of sliding window *swLen*, output binary file *outFile*

1. Append information about length of *ts*, *subLen* and *swLen* into *outFile*.
2. Split *ts* into subsequences of length *subLen*.
3. **FOR EACH** *seq* in these subsequences
4. Add values from *seq* to sliding window or replace values of the oldest sequence with these values.
5. Actualize domain structures according to sliding window.
6. Process ranges in *seq* in the same way as in **default compression algorithm** and store used transformations in *trList*.
7. Apply transformations from *trList* on *seq* in sliding window. {It will actualize values in sliding window corresponding to *seq*.}
8. **END FOR**

In figure 5.5 is shown how processing of time series could result. With red arrows are displayed the transformations that could produce error cumulation during decompression. That are only the transformations whose range and domain belong to the same subsequence. If in some subsequence some longer chain of dependent transformation occurs, it could lead to error cumulation but it will not affect other subsequences because after subsequence is processed, corresponding domain structures will be actualized and in next subsequences current domains will be used. So transformation from one subsequence to another will not cumulate any dependency error.

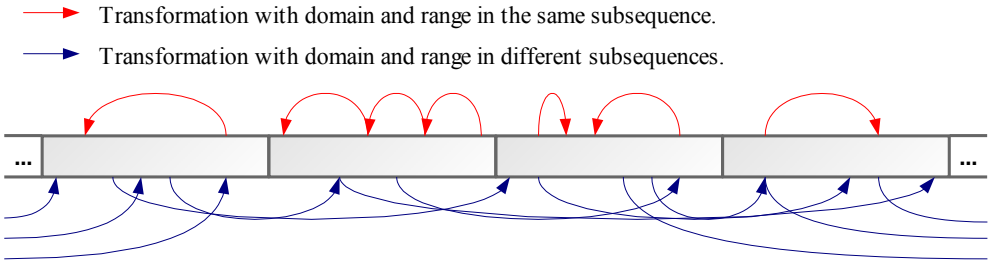

Figure 5.5 Illustration of transformation set obtained with sequential processing.

Decompression process is realized in sequential steps too. Transformations for single subsequence are read and applied to corresponding subsequence part in the sliding window. Then generated values are copied into result array. This is repeated for each subsequence and when sliding window is full, values for next subsequence will be generated in place of the oldest subsequence in sliding window.

Sequential compression could be simply modified to work like sliding window processing algorithm. Now whole time series is expected as input and then it is only processed sequentially. Compression routine needs to be modified so at each step it will get values of single subsequence, process them and wait for next subsequence.

Sequential decompression algorithm:

Input: input binary file *inFile*, number of iterations *itNum*

Output: decompressed time series *ts*

1. Read length of time series into *len*, subsequence length into *subLen* and sliding window length into *swLen*.
2. Set position of current subsequence in sliding window *pos* to 0.
3. **WHILE** not whole time series is processed
4. Read information about transformations for single subsequence.
5. *itNum* times apply read transformations on values in sliding window.
6. Add values from subsequence at position *pos* to *ts*.
7. Increase *pos* by 1 or set it again to 0 when it reaches *swLen*.
8. **END WHILE**
9. **RETURN** ts

5.5 Domain container for sequential processing

Sequential compression algorithm needs to be able to insert domains based on processed subsequences into domain container in particular steps and actualize domain structures according to changes in processed subsequence. Dynamic container for domains used in smoothing compression provides methods for actualization of domain structures but all domains could be inserted into container only at the beginning.

For these purposes dynamic domain container was extended with methods for inserting or replacing domains and simulate cyclic sliding window logic for domain values too. Structure of stored information is the same as shown in figure 5.2, but only domains from current sliding window are stored in the container at one time. It is initialized with length of subsequence and length of sliding window used in sequential compression algorithm and among common methods it provides these specific functionalities:

- Adding new domain structures according to the specified subsequence of range values. These domains are added to next subsequence position in the sliding window. When sliding window is full, domains from the oldest subsequence are replaced with domains from added subsequence.
- Actualization of domain structures corresponding to lastly added subsequence according to new values of this subsequence.

6 BOTTOM-UP PROCESSING

In the base version of compression algorithm time series is split into ranges so that range of the maximal length is taken first and we look for appropriate domain for it, that will not produce bigger than specified sufficient error. If we failed for this range, we split it into two ranges of half length and repeat the same procedure. If we have not succeed for small range, we describe it with its values. An idea of the bottom-up processing is based on opposite range processing. At the beginning we start with the shortest ranges and then we join them into longer ranges so that specified maximal error is not exceeded.

At the beginning is whole time series described by its values and split into ranges of length 2. There is a sufficient transformation for each range of length 2, but to encode it like 2 single values takes less space than writing information about transformation and domain position. In the second step for each neighbour ranges is calculated how big error their join into single range will produce. Then error value of the best pair of ranges is compared with given sufficient error and if it is bellow then these ranges are joined and this process is repeated for the current set of ranges. If the best pair results in bigger error than specified sufficient error, compression process is ended and the information about length, transformation and domain position is written to output file for each range.

This algorithm is very similar to the bottom-up segmentation algorithm described in second chapter and ranges corresponds with segments. First difference is that here ranges are described using transformation and domain position and unlike segmentation algorithm, this algorithm does not guarantee that the error obtained in decompression will not exceed specified sufficient error. It is again because error could be cumulated through the chains of dependent transformations during decompression.

For simplicity also here only domains of length $2n$ are browsed when range of length n is processed. Now we have not some small set of possible lengths for ranges, so we do not know lengths of domains that will be needed during compression. Storing information about all possible domains in some temporal structures seems worthless. Just the array with domain values and the arrays with cumulative information will be calculated at the beginning. Domain structures will be created from these arrays as needed. Because we will not use any domain structures we cannot classify domains into similarity classes with the classification used in base compression algorithm. But we know that each range will have at least 4 values, so we could classify domains according to their first 4 values as described in the section 3.4 and instead of domain structures only the pointers to first domain values will be ordered according to the similarity class. When some range will be processed, first its similarity class will be computed using its first 4 values and then simply the corresponding array of pointers to first domain values will be browsed. In the figure 6.1 a simple illustration of domain classification in bottom-up algorithm is given.

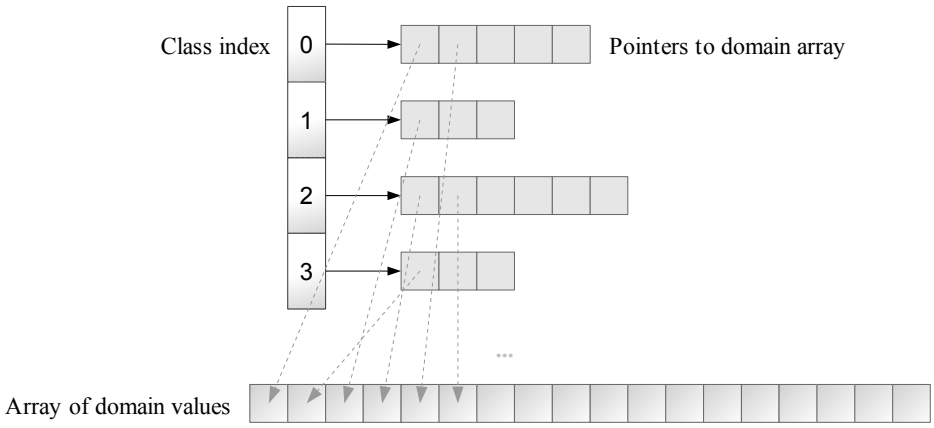

Figure 6.1 Illustration of domain classification in bottom-up algorithm.

Bottom-up compression algorithm:

Input: time series *ts*, maximal error *err*, output binary file *outFile*

1. Split *ts* into ranges of length 2.
2. **WHILE** true
3. Compute join cost for each neighbour pair of ranges.
4. Find range pair with the lowest join error.
5. **IF** join error of this pair is less than *err*
6. Join ranges in this pair into single range.
7. **ELSE**
8. **BREAK** {Error in this range is bigger than specified max error.}
9. **END IF**
10. **END WHILE**
11. Append information about left ranges into *outFile*.

The join cost need not to be calculated in each iteration for each neighbour pair of ranges. In the first iteration we need to calculate it for each pair but in the next iterations almost all of these information will still be valid. Only the pairs that contains last joined range need to be recalculated.

The decompression algorithm is almost the same as the default decompression algorithm. Only reading information about range lengths is different, because here exact range lengths needs to be stored not only some control bits. Other information are read in the same way and then also transformations are applied as in the default decompression process.

7 EXPERIMENTS

After the skeleton of default fractal compression algorithm for time series was implemented, mostly experiments to find out how to encode particular output sequences were performed. Derivation of number of bits that should be used to encode values in particular output sequences is presented in section 3.2 in analysis chapter. Technique how could we save some more bits in encoding information about linear transformations is presented in following section.

Other experiments were performed to compare and verify differences between presented compression algorithms and possible compression options.

7.1 Linear transformation serialization

When encoding linear transformation we could save some bits if integer part of constant factor will be encoded only using needed number of bits. This amount could be determined by linear factor and the sign of constant factor.

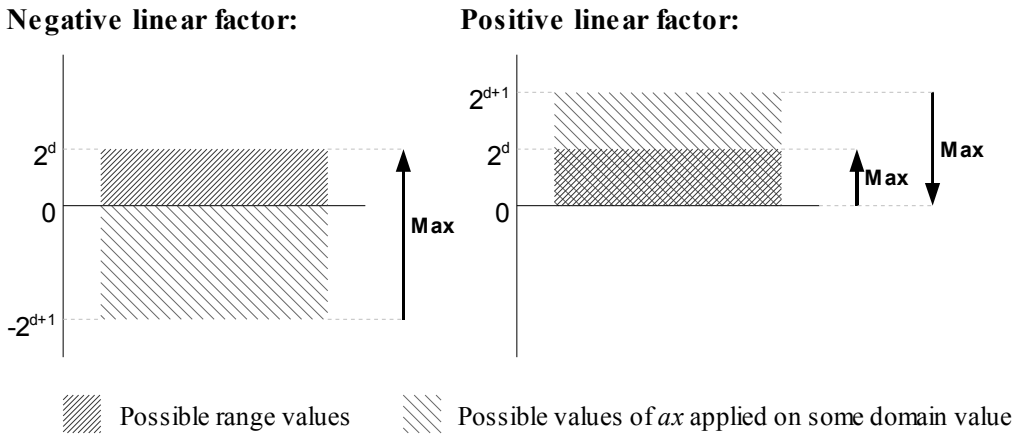

Figure 7.1 Illustration of maximal constant factor value according to sign of linear factor.

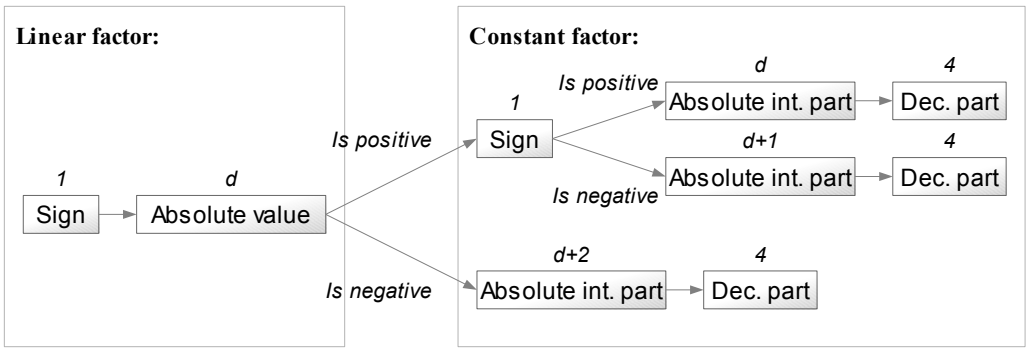

Figure 7.2 Diagram of linear transformation serialization.

In compression routines we work only with positive numbers. Assume that to encode single value of given time series we need *d* bits, so maximal value of this time series

is lower than 2^d. When linear factor is negative, value of constant factor is used to add some value to make the result positive and fit to interval of range values. In this case sign of constant factor will always be plus and absolute value of constant factor will fit in interval $\langle 0, 2^{d+1}+2^d \rangle$. So *d+2* bits are needed to encode integer part of constant factor and we do not need to store any information about its sign, because it will always be positive.

For positive linear factor, constant factor could be either negative or positive. When it is negative, its absolute value must be from interval $\langle 0, 2^{d+1} \rangle$. For positive constant factors their absolute value will be in interval $\langle 0, 2^d \rangle$. Here we need to store information about sign of constant factor and to encode its absolute integer part we need to use *d* or *d+1* bits, depending on its sign. Using this technique we will not need to encode each absolute value of integer part of constant factor with *d+2* bits, but only with number of bits needed to encode it with sufficient accuracy. In the figure 7.1 is shown illustration of described ideas and in the figure 7.2 is displayed diagram which represents how linear transformation is serialized.

7.2 Time series used in following experiments

It was quite difficult to collect reasonable set of real-world time series of various kinds. There are available many time series examples for educational purposes, on which statistical or data mining methods could be illustrated well, but they are very short to use them for testing compression algorithms for time series.

Long time series could be found mostly as outputs of some medical devices like ECG or respirator. Many medical time series could be found in [11] or [14]. Many oceanographic measurements data in long sequences could be found in [16]. For our experiments these 4 time series were chosen. List is followed by table which contains summary information about these time series.

- **ECG** - single output sequence of ECG data [11]. Original values were multiplied by *1000* to convert them to integer values.
- **Power** - power demands of some research facility during single year [11].
- **Resp** - respiration values during waking up [11]. Original values were multiplied by *100* and then rounded to integer values.
- **SedTr** - sediments transports in some ocean [16]. Original values were multiplied by *200* and left as real values. Total data size given in following table is the size of these values rounded to integers.

	ECG	Power	Resp	SedTr
Length	3750	35040	24125	19494
Max value	-340	2152	354076	71982.8
Min value	-2800	614	-56124	-13780.76
Average value	-1136.5	1144.04	3635.86	6833.21
Median	-1115	1011	-1324	1124.18
Bits for value	13	13	19	18
Total size	6095 B	56941 B	60314 B	43863 B

Table 7.1 Summary information about time series used in experiments.

7.3 Encoding of compression information

In compression procedure we need to store several information. In file header are stored information about compressed time series and information about chosen compression parameters, which are required in decompression. All parts of file header are encoded to binary form one by one and each part is encoded using well known number of bits which is sufficient for it.

Then information about particular ranges follows. Technique of storing information about range lengths is described in detail in section 4.1 and illustrated on figure 4.1. Each range is described either as a sequence of its exact values or as a linear transformation with domain position. In the result these 5 sequences are stored in the output file:

- Lengths of ranges (*RanLen*).
- Linear coefficients of used linear transformations (*LinF*).
- Constant coefficients of used linear transformations (*ConFInt*, *ConFDec*).
- Domain positions corresponding to non-constant linear transformations (*DomPos*).
- Exact values of processed time series (*ExVal*).

This information must be encoded using lossless compression technique. In following tables are shown numbers of bits needed to encode these sequences using fixed number of bits per value (*binary coding*) and using the *Huffman coding*. When sequence contains values bigger than $2^8 - 1$, then these values are also split to values smaller than 2^8 and Huffman coding is applied on this split sequence too - these numbers of bits are displayed in brackets. Constant factors are stored as two sequences, first represents the integer parts and second represents the decimal parts. Comparisons were made on all four time series mentioned in section 7.2. The goal is to achieve different lengths of particular sequences, that is why only one dynamic parameter here will be maximal sufficient error value. For each time series 5 tests using error values corresponding with different measures of accuracy were made. For other compression parameters default values were selected. List of chosen parameter values is shown in the table 7.2. Most of these parameter values will be used also in following experiments.

	ECG	Power	Resp	SedTr
Compression type	default	default	default	default
Error computer	MSE	MSE	MSE	MSE
Classifier	None	None	None	None
Domain density	1	1	1	1
Linear factor bits	default (12)	default (11)	default (19)	default (17)
Constant factor int. bits	default (13)	default (12)	default (20)	default (18)
Constant factor dec. bits	default (4)	default (4)	default (4)	default (4)
Value bits	default (12)	default (11)	default (19)	default (17)

Table 7.2 Compression parameters used in sequence encoding tests.

Err		RanLen	LinF	ConFInt	ConFDec	DomPos	ExVal
99.95% (1.26)	**Binary**	1631	10582	11080	3256	8954	5736
	Huffman	1315	15785 (11487)	19083 (11451)	3301	14054 (9732)	6911 (5927)
99.9% (2.46)	**Binary**	1579	11115	11636	3420	9405	2520
	Huffman	1114	17274 (12135)	20099 (12196)	3500	15041 (10204)	3570 (2863)
99.75% (6.15)	**Binary**	965	7605	7928	2340	6435	744
	Huffman	1010	13569 (8729)	13643 (8607)	2445	11360 (7116)	1116 (1003)
99.5% (12.3)	**Binary**	585	5317	5530	1636	4499	72
	Huffman	842	9458 (6338)	9441 (6196)	1741	7632 (5117)	103 (106)
99% (24.6)	**Binary**	407	4173	4320	1284	3531	24
	Huffman	545	7326 (5145)	7327 (5003)	1389	5711 (4091)	26 (26)

Table 7.3 Sequences encoding test results for ECG data.

Err		RanLen	LinF	ConFInt	ConFDec	DomPos	ExVal
99.95% (205.1)	**Binary**	15274	97140	102173	32380	121425	28028
	Huffman	10690	134556 (97283)	118842 (94635)	32485	200922 (116155)	37789 (27567)
99.9% (410.2)	**Binary**	14468	97524	102656	32508	121905	8888
	Huffman	9385	136082 (97659)	121799 (95573)	32613	202432 (116637)	15042 (8829)
99.75% (1025.5)	**Binary**	10892	78168	82280	26056	97710	1188
	Huffman	8817	114355 (78391)	104185 (77215)	26161	166831 (93747)	2111 (1421)
99.5% (2051)	**Binary**	7126	55824	58670	18608	69780	264
	Huffman	7934	86694 (56141)	79870 (55468)	18713	120652 (67309)	440 (455)
99% (4102)	**Binary**	4114	37824	39625	12608	47280	0
	Huffman	4889	63006 (38150)	57867 (37385)	12713	80962 (46017)	0 (0)

Table 7.4 Sequences encoding test results for power demands data.

Err		RanLen	LinF	ConFInt	ConFDec	DomPos	ExVal
99.95% (0.77)	**Binary**	3473	49720	50805	9944	34748	418
	Huffman	3894	82303 (51963)	82315 (48849)	10049	51993 (34723)	586 (733)
99.9% (1.54)	**Binary**	2189	36900	37592	7380	25802	342
	Huffman	2600	60539 (38720)	60371 (36398)	7485	35813 (25514)	478 (609)
99.75% (3.85)	**Binary**	1563	30660	31186	6132	21434	266
	Huffman	1637	49955 (32322)	49835 (30352)	6237	26356 (20956)	372 (494)

99.5% (7.7)	**Binary**	1531	30340	30864	6068	21210	266
	Huffman	1589	49411 (31884)	49339 (30062)	6173	25785 (20715)	372 (494)
99% (15.4)	**Binary**	1525	30300	30824	6060	21182	190
	Huffman	1575	49343 (31847)	49271 (29871)	6165	25748 (20684)	268 (359)

Table 7.5 Sequences encoding test for respiration data.

Err		RanLen	LinF	ConFInt	ConFDec	DomPos	ExVal
99.95% (42.88)	**Binary**	5525	51678	53207	11484	37730	34102
	Huffman	5995	85226 (51310)	84411 (49942)	11566	56517 (37431)	59591 (35259)
99.9% (85.76)	**Binary**	5427	55386	57019	12308	40614	16762
	Huffman	5149	92002 (55061)	91077 (53707)	12397	58862 (40126)	28367 (17527)
99.75% (214.44)	**Binary**	5265	54990	56596	12220	40306	12750
	Huffman	4885	91210 (54649)	90351 (53385)	12314	58046 (39788)	21329 (13397)
99.5% (428.8)	**Binary**	4925	52416	53947	11648	38304	10914
	Huffman	4870	86601 (52139)	85985 (50738)	11734	54694 (37853)	18110 (11521)
99% (857.6)	**Binary**	4231	46764	48157	10392	33922	8670
	Huffman	4657	76510 (46690)	76004 (45485)	10454	47071 (33738)	14173 (9210)

Table 7.6 Sequences encoding test for sediments transports in ocean.

As we can see in tables above, when range lengths are encoded as described in section 4.1 it is almost always cheaper than using Huffman coding. Huffman coding gives better results only when number of small ranges is much greater than number of other bigger lengths. Huffman coding is not better when number of ranges of lengths *2* is the greatest, but when most of ranges have length *4* then Huffman coding gives better results.

Linear factor values, which are numbers from $(-1,1)$ interval are multiplied with 2^d, where *d* is number of bits that are used to encode absolute value of linear factor. Huffman encoding is better here only for sediments transports data, but the reason is that in this sequence is one very long constant sequence and many linear factors are set to zero value and Huffman coding encodes them using some smaller amount of bits than the others. In general seems to be sufficient to encode linear factors using static amount of bits per single value.

Decimal parts of constant factors are almost always regularly dispersed as can be seen in many cases, where difference is about 100 bits, which is the number of bits needed to encode information about Huffman tree for current sequence and each value is encoded using the same number of bits as in binary encoding.

To describe respiration data many constant transformation were used and many constant coefficients have not any decimal part. That is why Huffman coding gives better results also here. For integer parts of constant factors Huffman encoding, applied to split values into *8-bits* parts, results in lower space needs than binary encoding. Only for ECG data it needs about 500 bits more than binary encoding. The

reason for this is, that in compression of these data maximally 855 values for constant factors need to be stored and values of them are in range $(-2158, 5254)$, so there is small chance that this sequence will contain more items of the same value. For integer parts of constant factors should be better to use Huffman encoding when number of transformations is not much smaller than size of interval of these values.

Domain positions are dispersed almost regularly in most of cases. Huffman encoding mostly gives better results for longer time series, but differences from binary encoding are not very big.

To encode sequence of exact time series values, binary encoding leads to better results in almost all cases. It is because these items have random values for which no sufficient domain, not even for range of length *4*, was found. For greater error values number of exactly encoded values goes rapidly down and that also leads to worse results of Huffman coding.

In the result Huffman coding will give lower sequence size only for range lengths, when number of ranges of length *4* is the biggest and usually for integer parts of constant factors. Also size of compressed file, where for each sequence better encoding method was used, is not much smaller than size of file compressed only with binary encoder. So implemented solution uses only binary encoding with fixed number of bits per corresponding value.

Numbers of bits for linear factors and decimal part of constant factor could be given as parameters to compression. Values for number of bits used for integer part of constant factor and for encoding the exact values could be specified also, but if specified number is lower than number of bits needed to encode the sequence exactly, all values are binary shifted in order to fit to specified amount of bits. Number of bits used to encode domain position is determined by length of given time series and chosen value of domain density.

To calculate number of bits needed to encode particular sequence using Huffman coding, a simple program was written. Program reads sequence of integers from specified text file, builds Huffman tree for these values and then calculates sum of bits needed to encode information about this tree and also the sequence. This application was written mainly for this purpose and it does not convert specified values into binary file encoded using Huffman coding.

7.4 Comparison of presented algorithms

Four fractal compression algorithms were described in previous chapters. The base compression algorithm uses only simple principles to encode given time series using contractive transformations. The smoothing and the sequential algorithms try to reduce error cumulation with recalculating transformation coefficients during compression or with processing given time series in smaller sequences. The bottom-up compression algorithm starts with the smallest ranges and these are merged while produced local error is sufficient.

Each compression method was used twice for each test time series described in section 7.2. Compression parameters were set to the same values as listed in table 7.2. These values were taken and compared through compression algorithms:

- Size of compressed file, which is encoded only using binary coding. Value obtained by using better encoder (binary or Huffman) for each sequence in output file is given in brackets (*Output size*).
- Value of root mean squared error between original and decompressed time series (*Total RMS*).
- Maximal difference between the corresponding values in original and decompressed time series (*Max difference*).
- Maximal root mean squared error between the corresponding subsequences in original and decompressed time series (*Max sl. RMS*). Error was computed for subsequences of lengths 4, 8 and 16.

Err		Default	Smooth	Sequential	Bottom-up
99.9% (2.46)	**Output size**	4971 B (4913 B)	4958 B (4905 B)	5022 B (4993 B)	4626 B (4481 B)
	Total RMS	0.94	0.92	0.91	1.24
	Max difference	5.73	5.44	5.48	6.2
	Max sl. RMS (4)	3.84	3.34	3.08	3.59
	Max sl. RMS (8)	2.89	2.64	2.45	3.18
	Max sl. RMS (16)	2.19	2.07	2.09	2.37
99.5% (12.3)	**Output size**	2216 B (2216 B)	2221 B (2221 B)	2316 (2316 B)	2089 B (2034 B)
	Total RMS	8.42	7.83	7.87	8.78
	Max difference	36.3	33	30.64	35.88
	Max sl. RMS (4)	22.43	20.46	19.41	20.47
	Max sl. RMS (8)	18.55	16.73	16.38	17.55
	Max sl. RMS (16)	15.52	13.44	13.97	15.72

Table 7.7 Algorithms comparison for ECG data.

Err		Default	Smooth	Sequential	Bottom-up
99.9% (1.54)	**Output size**	47255 B (45069 B)	47264 B (45140 B)	47524 B (45249 B)	45312 B (42366 B)
	Total RMS	0.61	0.61	0.64	0.71
	Max difference	4.2	4.01	3.53	4.47
	Max sl. RMS (4)	2.87	2.48	2.15	2.64
	Max sl. RMS (8)	2.16	2.13	1.74	2.09
	Max sl. RMS (16)	1.81	1.8	1.43	1.74
99.5% (7.69)	**Output size**	26330 B (25600 B)	26578 B (25851 B)	30159 B (29597 B)	25570 B (22983 B)
	Total RMS	5.33	4.73	4.48	5.95
	Max difference	29.53	27.63	23.49	45.56
	Max sl. RMS (4)	19.56	17.74	14.01	24.03
	Max sl. RMS (8)	15.32	12.99	11.83	19.29
	Max sl. RMS (16)	11.56	10.47	9.43	14.03

Table 7.8 Algorithms comparison for power demands data.

Err		Default	Smooth	Sequential	Bottom-up
99.9% (410.2)	**Output size**	13787 B (13602 B)	13778 B (13617 B)	15807 B (15745 B)	11506 B (10772 B)
	Total RMS	215.53	207.66	207.75	259.7
	Max difference	1350.8	1282.28	1330.54	1882.6
	Max sl. RMS (4)	898.34	777.7	844.52	1098.77
	Max sl. RMS (8)	657.88	569.74	684.2	891.24
	Max sl. RMS (16)	485.55	461.19	530.42	679.87
99.5% (2051)	**Output size**	11296 B (11134 B)	11309 B (10832 B)	10642 B (10572 B)	4208 B (4052 B)
	Total RMS	392.26	375.74	591.29	1596.68
	Max difference	4944.83	4850.04	5417.62	9019.26
	Max sl. RMS (4)	3075.97	3075.78	3437.05	7972.68
	Max sl. RMS (8)	2690.72	2686.71	2790.75	6810.92
	Max sl. RMS (16)	2122.14	1990.28	2110.94	5066.72

Table 7.9 Algorithms comparison for respiration data.

Err		Default	Smooth	Sequential	Bottom-up
99.9% (85.76)	**Output size**	23451 B (22901 B)	23588 B 23039 B)	23522 B (23060 B)	24078 B (19932 B)
	Total RMS	24.18	23.45	29.52	33.45
	Max difference	231.83	231.7	252.22	643.32
	Max sl. RMS (4)	148.03	148	142.24	324.76
	Max sl. RMS (8)	117.17	115.94	106.92	234.31
	Max sl. RMS (16)	98.28	98.07	97.49	171.27
99.5% (428.82)	**Output size**	21530 B (21031 B)	21701 B (21199 B)	20770 B (20338 B)	18968 B (15684 B)
	Total RMS	107.41	103.79	112.31	288.1
	Max difference	1056.28	1044.22	1010.82	13616.65
	Max sl. RMS (4)	691.45	691.49	584.08	9405.35
	Max sl. RMS (8)	555.28	518.25	510.19	7612.52
	Max sl. RMS (16)	493.1	453.46	463.63	5383.04

Table 7.10 Algorithms comparison for sediments transports data.

First we could notice, that even when better encoding method from binary or Huffman coding was chosen for each encoded result sequence, size of result compressed file is quite the same as if only binary encoding was used. Sizes of the result files, where also Huffman encoding was used, were smaller maximally within *5%*. For ECG data in top-down compression algorithms with *99.5%* accuracy gave the Huffman coding even worse results than binary encoding for every output sequence. Little different case is the bottom-up algorithm which encodes lengths of ranges using binary encoder and this sequence usually contains small set of values and therefore Huffman coding is almost always better to use to encode them.

Top-down algorithms derived from the base compression algorithm (smoothing and sequential) for roughly the same size of output file produce better quality of decompressed time series. They beat default compression algorithm in almost all measured quality statistics and they give about *5% - 10%* better quality.

Bottom-up compression algorithm leads to smaller size of compressed file but also to smaller quality than the other (top-down) algorithms. Output file size ratios correspond with quality ratios between bottom-up algorithm and other algorithms. The main reason for these differences is that in bottom-up compression algorithm range lengths are not bounded and ranges could be processed with almost maximal sufficient errors. In decompression these errors could be also cumulated when transformation for particular range is dependent on some other ranges which could also be processed with bigger error value.

Too big quality differences in decompressed sequences are between the bottom-up algorithm and the top-down algorithms in last sediments transports time series. Maximal difference between corresponding values is as far as three times bigger for 99.9% compression sufficient accuracy (around 200 for default algorithm against 643 for bottom-up algorithm) and with 99.5% accuracy the differences are even bigger (about 1068 for default and about 13000 for bottom-up). First thoughts could be like, that bottom-up compression algorithm does not work correctly, but there is an explanation for these differences. There are three main reasons for this:

- Length of ranges is not bounded in bottom-up algorithm.
- Processed sequence of sediments transports in the ocean is not very common. It contains very long sequence of zero values which could be caused by loss or damage of data in this period.
- As error measure is used R*MS* metric and this error is computed according to the whole processed range.

In the bottom-up compression algorithm neighbour ranges which produce the lowest error value are joined first. So ranges in this long zero sequence are processed in the beginning of compression. Step by step these zero values are grouped into one range of length 2784. An error produced by this range is equal to *0*, because this range could be simply described with constant zero transformation. Right before this zero sequence are values very different from zero (*3331*, *7805*, *10288* and *-13480*) and right after this sequence is value *-13480*. Error values for range pairs with this long range are calculated and almost constant transformation with values near zero is used. Thanks to this big number of zero values, value of total *RMS* error of these joined ranges will be low enough to be declared as sufficient. And then this segment is also joined with another long segment which consists mostly of values around *190*.

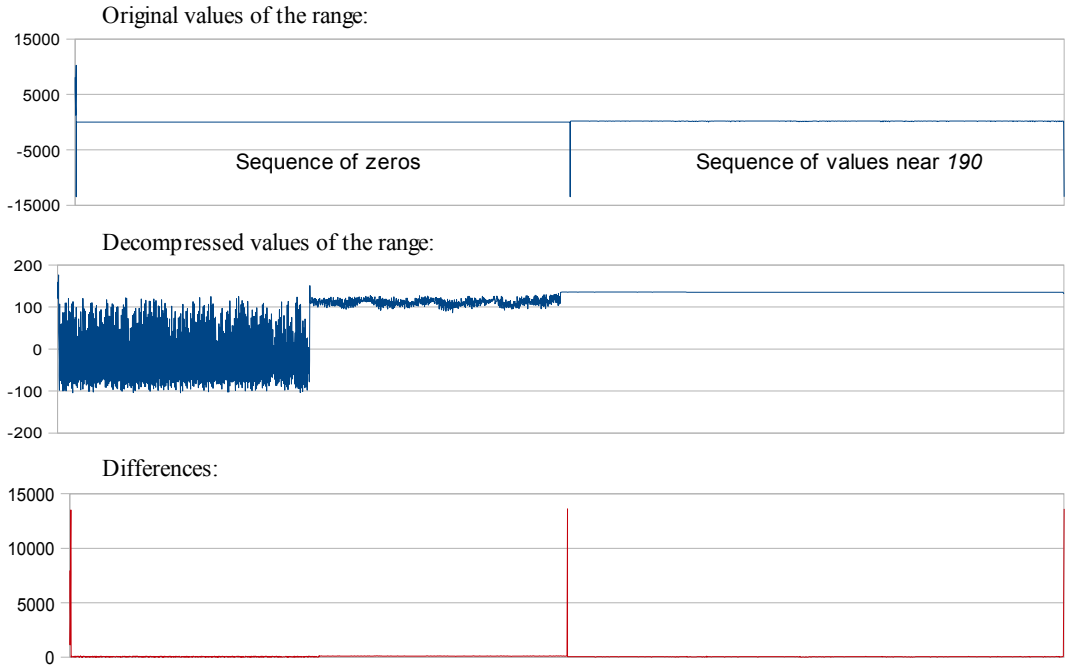

Figure 7.3 Original and decompressed values of longest range created in bottom-up compression of sediments transports and difference values.

As we can see in the figure 7.3, original values of this longest range contain long zero subsequence, long sequence of values mostly near *190* and between and next to these subsequences are items with high absolute values. These constant and almost constant subsequences keep the final RMS error value low so a few items with high differences could be added to this range and total RMS error value will be still sufficient.

Average difference value between original and decompressed time series for this range is *80.79* and according to specified maximal sufficient error *428.82* could it be declared as sufficient. But when someone looks at original and decompressed time series, he may not say that the decompressed time series is generated from compressed original time series.

There are more options which should prevent from defects like this. For example range length could be bounded within some reasonable value or *MAX* metric could be used instead of *RMS* metric or simply values of very different items could be appended to compression output file in exact form and then after decompression process they could replace generated values.

7.5 Using different error computers

For each compression algorithm sufficient error value is specified as input parameter. This error value is used to test whether found transformation generates processed range with the error that is under this specified error value. By given error value is meant that in decompressed time series we want single value to differ from original value maximally (or approximately maximally) with given error value.

In created solution all three error computers described in section 3.7 are available. In following experiments were time series compressed with each type of error computer and the other compression parameters were set to default values as listed in table 7.2.

As input time series were used the ECG data and the respiration data and each were compressed using different sufficient error values. In following tables these information are presented:

- **Err** - chosen percentual accuracy, used error value is given in brackets.
- **E.C.** - used error computer.
- **Size** - size of compressed file encoded only using binary encoder.
- **RMS** - RMS difference between the original and decompressed time series.
- **Diff** - maximal difference between corresponding values.
- **Max Sl. RMS** - maximal RMS difference value between corresponding sliding subsequences of lengths 4, 8 and 16.
- **Max Sl. MEAN** - maximal average difference between corresponding sliding subsequences of lengths 4, 8 and 16.

Err	E.C.	Size	RMS	Diff	Max Sl. RMS	Max Sl. MEAN
99.95% (1.23)	**MSE**	5166 B	0.56	2.87	1.88 | 1.61 | 1.23	1.57 | 1.28 | 0.87
	MEAN	5141 B	0.61	3.38	1.95 | 1.5 | 1.2	1.61 | 1.19 | 0.9
	MAX	5262 B	0.44	2.2	1.46 | 1.11 | 0.83	1.22 | 0.89 | 0.62
99.9% (2.46)	**MSE**	4971 B	0.94	5.73	3.84 | 2.89 | 2.2	3.43 | 2.38 | 1.72
	MEAN	4728 B	1.39	9.13	5 | 4.1 | 3.54	4.33 | 3.25 | 2.54
	MAX	5135 B	0.67	4.3	2.72 | 2 | 1.49	2.26 | 1.48 | 0.99
99.75% (6.15)	**MSE**	3263 B	4.3	19.03	12.49 | 9.73 | 7.49	11.32 | 8.31 | 6.24
	MEAN	2836 B	5.67	29.05	16.68 | 12.69 | 10.21	13.53 | 10.3 | 8.1
	MAX	4437 B	2.02	9.89	6.6 | 4.73 | 4.21	6.55 | 4.44 | 3.66
99.5% (12.3)	**MSE**	2216 B	8.42	36.3	22.43 | 18.55 | 15.52	21.21 | 16.03 | 12.86
	MEAN	1959 B	10.17	52.6	28.67 | 24.96 | 20.68	24.95 | 20.79 | 15.82
	MAX	2995 B	5.33	20.69	13.66 | 12.7 | 9.33	12.47 | 11.58 | 8.49
99% (24.6)	**MSE**	1729 B	12.15	58.43	37.69 | 28.4 | 24.26	35.03 | 26.93 | 20.16
	MEAN	1679 B	13.61	79.33	44.11 | 38.48 | 31.51	38.31 | 32.47 | 25.25
	MAX	2128 B	9.67	40.27	23.75 | 19.33 | 16.59	22.98 | 17.66 | 14.46

Table 7.11 Comparison of error computers on ECG data

Err	E.C.	Size	RMS	Diff	Max Sl. RMS	Max Sl. MEAN
99.95% (205.1)	**MSE**	18650 B	111.7	645.8	412.7 | 314.8 | 248.9	391.4 | 267 | 212.1
	MEAN	16799 B	148.29	1387.3	759.5 | 540 | 397.8	581.3 | 465.5 | 300.8
	MAX	23746 B	70.05	423.5	315.4 | 248.2 | 190.5	280.3 | 204 | 137.5
99.9% (410.2)	**MSE**	13787 B	215.53	1350.8	898.3 | 657.9 | 485.5	782.7 | 570.6 | 389.3
	MEAN	12770 B	265.16	1971	1115.2 | 900 | 670.9	1073.1 | 805.5 | 543.2
	MAX	18138 B	127.65	659.8	452.2 | 384.9 | 322.1	420.6 | 349.6 | 286.1
99.75% (1025.5)	**MSE**	11416 B	349.08	2597.2	1816 | 1495.7 | 1203.7	1730.1 | 1298.3 | 1044.1
	MEAN	11361 B	381.23	5238.5	3012.8 | 2227.3 | 1616.6	2484.7 | 1772.3 | 1383
	MAX	12762 B	279.72	1923.2	1221.5 | 1030.5 | 861.8	1131.4 | 926 | 755.2

99.5% (2051)	**MSE**	11296 B	392.26	4944.8	3076 | 2690.7 | 2122.1	2576.9 | 2302.7 | 1761.1
	MEAN	11293 B	419.42	5238.5	3263.5 | 2717.2 | 2158.4	3117.3 | 2252.5 | 1711.9
	MAX	11393 B	382.77	3397	2026 | 1615.1 | 1372.9	1710 | 1472.2 | 1164.3
99% (4102)	**MSE**	11271 B	433.51	8322.3	5306 | 4389 | 4070	4770 | 3870 | 3354
	MEAN	11253 B	505.08	22061.7	13111 | 11575 | 8948	11128 | 9299 | 6716
	MAX	11293 B	433.3	4788.8	3080 | 2469 | 2192	2691 | 2235 | 1949

Table 7.12 Comparison of error computers on respiration data.

As expected because of particular error computers definition, *MEAN* error computer is the least strict and will declare also not very good transformation for processed range as sufficient. Maximal difference between corresponding values depends on the worse processed range and its length. In ideal case (without error cumulation) this maximal difference is bounded with value $len \cdot maxerr$. And for smaller sufficient error values most of longer ranges are processed with error which is just bellow specified maximal error. On the other side the chance that the worse sufficient domain (except one value all other match perfectly and this one differs much from original value) will be chosen as the best is very close to zero. For smaller error values in ECG data maximal difference is about 3 times more than specified error. For bigger error values in ECG data and for each result in respiration data, maximal ranges were used often and also maximal difference is about 5 times greater than specified error.

Values of maximal sliding *MEAN* errors look as expected. It is between specified maximal error value and its double value. Usually it is near *1.5* multiply of given sufficient error. This occurs when some processed neighbour ranges, both have most of total mean error value in the half closer to this neighbour. For smaller sizes of sliding window (4 or 8) this maximal *MEAN* error is greater. This is also the result of non-uniform splitting of this error in longer subsequences.

On the other hand the *MAX* error computer is the most strict one. It takes care about more different values and allows to process only ranges where differences between all corresponding values are lower than specified sufficient error. Using this error computer problem ranges for *MSE* or *MEAN* error computer will not be processed. As we can see in result table for ECG data, almost in all quality statistics gives *RMS* error computer about two times worse values and *MEAN* error computer about three times worse values than *MAX* error computer. For greater sufficient error values differences between error computers are smaller. In respiration data these statistics are more closer. Reason for this is bounded maximal length for ranges. For big sufficient error value for almost each range of maximal length some sufficient domain with transformation will be found and produced error is not usually the same as in the worst scenario. In table 7.13 we can see comparisons of error computers using bottom-up algorithm in which maximal length for processed ranges is not bounded.

Even for *MAX* error computer the maximal difference between corresponding values is not under specified maximal sufficient error value. Usually the maximal difference is about *1.5* times greater than specified value of sufficient error. Here the cause is only error cumulation through dependent transformation chains in decompression. As we can see in the table 7.11 for ECG data, if small value of sufficient maximal error is chosen, value of maximal difference between corresponding values is smaller than this specified value (for accuracy *99.95%* and *99.9%*). But here many ranges are

split to the smallest sizes and at the end their exact values are output. These exact ranges reset cumulated error value to *0* in transformation dependency chains.

Err	E.C.	Size	RMS	Diff	Max Sl. RMS	Max Sl. MEAN
99.95% (1.23)	**MSE**	5136 B	0.54	3.35	1.98 | 1.5 | 1.16	1.66 | 1.18 | 0.85
	MEAN	4971 B	0.72	4.9	2.61 | 1.93 | 1.52	1.96 | 1.45 | 1.2
	MAX	5338 B	0.36	1.83	1.14 | 0.87 | 0.7	1.12 | 0.83 | 0.59
99.9% (2.46)	**MSE**	4625 B	1.24	6.2	3.59 | 3.18 | 2.37	3.36 | 2.74 | 2.05
	MEAN	4340 B	1.71	10.58	6.13 | 4.46 | 3.61	5.28 | 3.59 | 2.6
	MAX	4997 B	0.71	3.56	2.5 | 1.8 | 1.53	2.2 | 1.45 | 1.3
99.75% (6.15)	**MSE**	3133 B	4.14	16.08	10.6 | 8.46 | 6.9	9.62 | 7.86 | 6.26
	MEAN	2720 B	5.59	29.11	16.18 | 12.19 | 9.53	13.55 | 10.94 | 8.19
	MAX	4085 B	2.2	11.51	7.08 | 5.73 | 4.73	6.02 | 4.67 | 3.74
99.5% (12.3)	**MSE**	2089 B	8.78	35.88	20.47 | 17.55 | 15.72	19.73 | 16.35 | 14.1
	MEAN	1802 B	11.69	66.45	37.35 | 32.74 | 24.04	32.04 | 27.66 | 18.7
	MAX	2908 B	5.37	26.75	15.76 | 12.26 | 9.84	13.79 | 9.71 | 8.68
99% (24.6)	**MSE**	1298 B	19.76	112.81	88.45 | 72.46 | 56.53	83.64 | 66.11 | 48.23
	MEAN	1145 B	23.9	207.03	116.64 | 85.64 | 63.53	94.39 | 65.62 | 54.83
	MAX	2037 B	9.71	38.95	29.26 | 23.69 | 17.93	27.5 | 20.77 | 15.13

Table 7.13 Comparison of error computers on ECG data using bottom-up algorithm.

As mentioned in previous paragraphs, when bottom-up algorithm is used then even for given bigger sufficient error values, watched quality properties are very different for particular error computers. For bigger values of sufficient maximal error are differences between error computers even bigger. Differences in output sizes are quite corresponding to differences in watched quality properties and even for greater values of maximal sufficient error are differences between output sizes very big.

Err	E.C.	Size	RMS	Diff	Max Sl. RMS	Max Sl. MEAN
99.5% (428.82)	**MSE**	18968 B	288.1	13616.65	9405 | 7613 | 5383	34484 | 52906 | 53319
	MAX	20801 B	95.93	810.3	647 | 459 | 327	626.3 | 347.8 | 322.2

Table 7.14 Statistics for sediments transports compressed using bottom-up algorithm using *MSE* and *MAX* error computer.

In table 7.14 are compared *MSE* and *MAX* error computers used in bottom-up compression of sediments transports. This case was already discussed in previous section, because obtained difference values were much bigger than specified sufficient error. Now *MAX* error computer was used and obtained results look much better than for *MSE* error computer. So problem with the very long range, which was described with almost constant transformation and led to huge differences, is solved. In following figure 7.4 are displayed values from problem range from figure 7.3. Now this subsequence is described using 7 ranges and 2 of them have length 2, so their exact values are stored. As we can see in second graph, decompressed subsequence looks more like original subsequence. Long sequence of zeros was described using constant zero transformation so there no error is produced. Second almost constant sequence of values near *190* is described with single almost constant transformation and differences here are mostly under *100* so according to specified

maximal sufficient error *428* is it alright. Subsequences with the huge values are processed as ranges of smaller lengths.

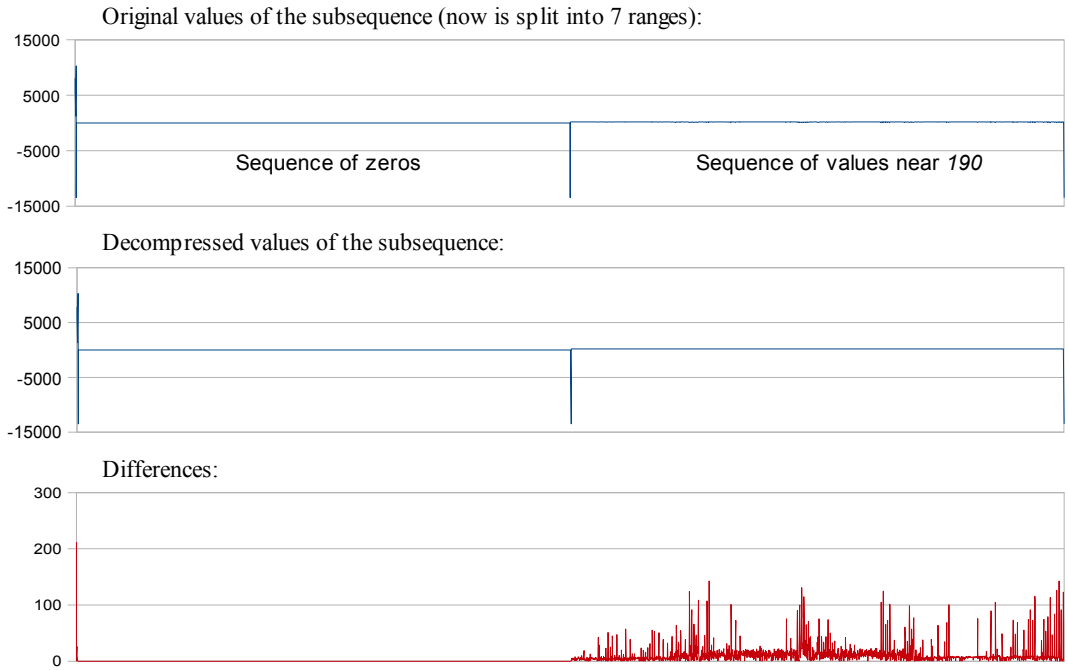

Figure 7.4 Original and decompressed values of problem subsequence compressed by bottom-up algorithm using *MAX* error computer.

7.6 Reduction of browsed domains

In each presented algorithm the value of domain density need to be specified as input parameter. This value determines distance between the first values of neighbour domains. Smaller domain density value means more domains, greater value means smaller number of domains. For smaller number of domains also less bits is needed to encode domain position.

In following experiments time series were processed by the base compression algorithm using different domain density values. The other compressed parameters were set to the same values as listed in table 7.2. The same information except maximal sliding *MEAN* error were collected as in experiments from previous section. In result tables domain density is referred by **DomDens** label.

Err	DomDens	Size	RMS	Diff	Max Sl. RMS
99.95% (1.23)	**1**	5166 B	0.56	2.87	1.88 | 1.6 | 1.22
	2	5114 B	0.61	3.18	2.08 | 1.61 | 1.29
	4	5136 B	0.61	3.58	2.02 | 1.61 | 1.24
99.9% (2.46)	**1**	4971 B	0.94	5.73	3.84 | 2.89 | 2.2
	2	4927 B	0.98	5.74	3.76 | 2.82 | 2.15
	4	4890 B	1.06	5.69	3.35 | 2.54 | 1.97
99.75% (6.15)	**1**	3263 B	4.3	19.03	12.49 | 9.73 | 7.49
	2	3358 B	4.29	19.03	12.16 | 9.6 | 7.99
	4	3510 B	4.2	18.4	12.51 | 10.19 | 7.87

99.5% (12.3)	**1**	2216 B	8.42	36.3	22.42 | 18.55 | 15.52
	2	2240 B	8.55	32.68	22.93 | 18.42 | 14.81
	4	2273 B	8.61	48.69	31.74 | 26 | 19.85
99% (24.6)	**1**	1729 B	12.15	58.43	37.69 | 28.4 | 24.26
	2	1715 B	12.86	68.58	37.68 | 30.97 | 26.53
	4	1727 B	13.55	76.81	44.41 | 34.34 | 27.72

Table 7.15 Different domain density used for ECG data.

Err	DomDens	Size	RMS	Diff	Max Sl. RMS
99.95% (42.88)	**1**	24227 B	16.47	121.17	69.62 | 54.37 | 48.93
	2	24459 B	17.23	136.82	76.77 | 65.4 | 49.23
	4	24937 B	17.27	138.75	77.51 | 59.64 | 50.06
99.9% (85.76)	**1**	23451 B	24.18	231.83	148.03 | 117.17 | 98.28
	2	23299 B	27.48	232.11	148.03 | 118.74 | 100.13
	4	23326 B	30.19	232.1	142.1 | 115.76 | 96.81
99.75% (214.44)	**1**	22777 B	41.63	440.71	301.09 | 228.59 | 184.1
	2	22482 B	47.04	440.69	296.19 | 231.58 | 188.07
	4	22268 B	52.1	491.75	326.48 | 275.64 | 224.78
99.5% (428.8)	**1**	21530 B	107.41	1056.28	691.44 | 555.28 | 493.09
	2	21347 B	104.81	1061.87	690.19 | 570.84 | 440.57
	4	21196 B	103.92	1092.16	690.32 | 531.73 | 420.52
99% (857.6)	**1**	19028 B	343.22	2791.25	1973.83 | 1683.56 | 1329.7
	2	18979 B	345.1	3221.97	1886.26 | 1570.33 | 1233.08
	4	19196 B	321.74	3289.39	1853.87 | 1431.87 | 1233.57

Table 7.16 Different domain density used for sediments transports data.

Err	DomDens	Size	RMS	Diff	Max Sl. RMS
99.95% (0.77)	**1**	49564 B	0.31	2.23	1.49 | 1.12 | 0.82
	2	48685 B	0.33	2.49	1.41 | 1.06 | 0.81
	4	47948 B	0.35	2.34	1.38 | 1.05 | 0.82
99.9% (1.54)	**1**	47255 B	0.61	4.2	2.87 | 2.16 | 1.81
	2	46957 B	0.61	4.02	2.65 | 2.18 | 1.62
	4	46517 B	0.63	5.48	3.52 | 2.6 | 2
99.75% (3.85)	**1**	37048 B	2.23	12.43	8.6 | 7.13 | 5.53
	2	37589 B	2.17	14.1	8.99 | 6.97 | 5.18
	4	38123 B	2.1	12.3	9.79 | 7.46 | 5.88
99.5% (7.7)	**1**	26295 B	5.33	29.53	19.56 | 15.31 | 11.55
	2	27132 B	5.25	29.93	18.01 | 13.65 | 11.34
	4	28132 B	5.09	28.93	16.67 | 14.06 | 11.09
99% (15.4)	**1**	17693 B	10.13	52.71	32.13 | 27.15 | 22.25
	2	17973 B	10.3	59.8	33.44 | 25.28 | 22.36
	4	18692 B	10.18	57.54	35.1 | 27.69 | 22.52

Table 7.17 Different domain density used for power demands data.

Err	DomDens	Size	RMS	Diff	Max Sl. RMS
99.95% (205.1)	**1**	18650 B	111.7	645.78	412.73 | 314.76 | 248.88
	2	18992 B	114.08	656.98	376.19 | 303.86 | 258.55
	4	19559 B	115.55	651.34	377.1 | 302.64 | 249.72
99.9% (410.2)	**1**	13787 B	215.53	1350.8	898.34 | 657.88 | 485.55
	2	14042 B	216.52	1232.97	726.17 | 605.16 | 547.39
	4	14340 B	221.29	1228.5	783.25 | 687.76 | 578.34
99.75% (1025.5)	**1**	11416 B	349.08	2597.21	1815.98 | 1495.7 | 1203.73
	2	11271 B	376.58	2880.8	1926.58 | 1517.85 | 1275.88
	4	11204 B	402.46	3080.75	1963.68 | 1588.2 | 1270.71
99.5% (2051)	**1**	11296 B	392.26	4944.83	3075.97 | 2690.72 | 2122.14
	2	11108 B	426.3	6070.18	3972.97 | 3212.17 | 2358.7
	4	10913 B	479.54	6189.13	4026.06 | 3248.45 | 2392.06
99% (4102)	**1**	11271 B	433.51	8322.3	5305.56 | 4389 | 4070.31
	2	11091 B	454.14	8322.3	5304.56 | 4328.38 | 3922.73
	4	10905 B	485.95	6189.13	4026.06 | 3248.44 | 2429.76

Table 7.18 Different domain density used for respiration data.

As we can see in these tables, no one of domain density values could be declared as the best one. Results for each domain density are very similar for output file size and also for measured quality properties. As default value for domain density the middle way and the value *2* was chosen.

With smaller number of browsed domains optimal domain could be rejected and there are two possible ways how processing of the range will continue. Some other sufficient domain could be found for the range, but error value will be greater. Or there will not be any sufficient domain and this range will be split into two smaller ranges. This will usually lead to decrease of the total error value, but also it will increase size of the output file.

The next way to decrease number of browsed domains in compression is to specify some classifier for domains. Then for each range only domains from the same similarity class will be browsed. In next experiment all implemented classifiers were used in compression of the ECG data. The other compression parameters were set to the same values as shown in table 7.2. As compression algorithms base and bottom-up algorithms were chosen, because different domain classifications are used in them. In base (and also in the derived top-down algorithms) longer domains are classified using grouping corresponding values and in bottom-up algorithm only the first four values from domain of any length are taken and used to classify the domain. Type of classifier is specified in column named **Class** and particular classifiers are referred using these labels:

- *NONE* when no classification was used.
- *INCDEC* when simple increase-decrease classification was used.
- *MONO* when derived classification from *INCDEC* was used and opposite classes were merged into single class.

Err	Class	Size	RMS	Diff	Max Sl. RMS
99.95% (1.23)	**NONE**	5166 B	0.56	2.87	1.88 | 1.6 | 1.22
	MONO	5166 B	0.59	3.41	1.92 | 1.43 | 1.19
	INCDEC	5211 B	0.59	3.44	1.85 | 1.51 | 1.18
99.9% (2.46)	**NONE**	4971 B	0.94	5.73	3.84 | 2.89 | 2.2
	MONO	4965 B	0.98	5.68	3.82 | 2.85 | 2.18
	INCDEC	5013 B	1.04	5.34	3.22 | 2.66 | 2.1
99.75% (6.15)	**NONE**	3263 B	4.3	19.03	12.49 | 9.73 | 7.49
	MONO	3317 B	4.19	19.03	12 | 9.37 | 7.22
	INCDEC	3445 B	4.22	18.96	10.56 | 8.91 | 7.31
99.5% (12.3)	**NONE**	2216 B	8.42	36.3	22.42 | 18.55 | 15.52
	MONO	2226 B	8.44	36.59	21.99 | 17.85 | 14.17
	INCDEC	2297 B	8.55	35.61	21.55 | 17.41 | 14.87
99% (24.6)	**NONE**	1729 B	12.15	58.43	37.69 | 28.4 | 24.26
	MONO	1755 B	12.35	61.17	38.41 | 32.16 | 26.95
	INCDEC	1794 B	12.79	66.48	37.11 | 32.38 | 26.39

Table 7.19 Different similarity classifiers used in base compression algorithm for ECG data.

For small values of sufficient maximal error was expected that differences between used classifiers will be quite similar. Its because here during compression mostly ranges of length *4* are processed and these are classified using their exact values (grouping of values is not performed). Probability that the best domain for processed range will be in different similarity class is very small. On the other hand differences in output sizes and quality properties are similar also for greater values of sufficient maximal error.

Presented domain classifiers are based on similarity relations and domains (or ranges) that belongs to the same similarity class are also similar and this similarity class does not change even when linear transformation is applied on the domain. Exception is increase-decrease classifier where if transformation has negative linear factor class index changes to inverse class. As default domain classifier monotonic (*MONO*) classifier was chosen. It gives almost the same results as when no classifier is used and each domain is processed only within ranges of the same class - so dissimilar ranges are not processed uselessly.

Err	Class	Size	MSE	Diff	Max Sl. MSE
99.95% (1.23)	**NONE**	5136 B	0.54	3.35	1.97 | 1.5 | 1.16
	MONO	5157 B	0.52	2.49	1.61 | 1.26 | 1.03
	INCDEC	5225 B	0.52	3.44	1.8 | 1.52 | 1.12
99.9% (2.46)	**NONE**	4625 B	1.24	6.2	3.59 | 3.18 | 2.37
	MONO	4726 B	1.18	6.64	3.89 | 3.06 | 2.41
	INCDEC	4830 B	1.2	7.51	4.21 | 3.02 | 2.49
99.75% (6.15)	**NONE**	3133 B	4.14	16.08	10.6 | 8.46 | 6.89
	MONO	3251 B	4.15	18.62	11.82 | 9.18 | 7.55
	INCDEC	3430 B	4.14	16.8	11.12 | 8.54 | 8.01

99.5% (12.3)	**NONE**	2089 B	8.78	35.88	20.47 | 17.55 | 15.72
	MONO	2143 B	8.84	34.8	25.59 | 21.44 | 16.97
	INCDEC	2285 B	9.13	39.34	24.68 | 20.13 | 16.68
99% (24.6)	**NONE**	1298 B	19.76	112.81	88.45 | 72.46 | 56.52
	MONO	1405 B	19.17	114.53	88.54 | 72.46 | 54.3
	INCDEC	1427 B	19.09	104.37	59.2 | 45.73 | 37.39

Table 7.20 Different similarity classifiers used in bottom-up compression algorithm for ECG data.

In base compression algorithm always all values are used to calculate index of similarity class and even when values of domain or range need to be grouped this similarity domain filtering does not change compression results very much. For bottom-up compression algorithm are differences between used similarity classifiers small only when given sufficient error is small. For greater values of sufficient error first output sizes start to differ noticeable and values of compression quality are still quite similar, but then for even greater error values also here differences are bigger. Reason for this was already described in analysis part in section 3.4 and it is that only first four values are used to calculate similarity class for domains and ranges of all lengths. This could filter out also some good domains only because they do not match with their first four values.

7.7 Fractal compression vs. segmentation

In section 2.2 the piecewise linear representation of time series was introduced and also some segmentation algorithms were described which transforms given time series into segments described by single lines. Presented fractal compression algorithms also split time series into set of ranges and these are described using transformation and domain position. Some advantages and disadvantages could be list when these types of algorithms are confronted.

- Piecewise linear representation uses only straight lines to describe particular segments. In presented fractal compression algorithms also ranges that does not look like single line could be processed and according to this, total number of ranges could be much smaller than when only lines are used.
- In piecewise linear representation number of bits used to encode particular segments is constant (when no other encoding method is used to encode result sequences) and information stored for single segment are enough to recalculate values of this segment. So whole time series need not to be computed when only some values are desired. In fractal compression, thanks to dependencies between the processed ranges, position of information about particular range in output file could not be determined and if yes, also to recalculate values of this range, we need to recalculate values of all ranges on which this one depends.
- Segmentation algorithms could guarantee that the specified quality or the output size criteria will be satisfied. Stop condition for segmentation algorithms could be simply checked after each iteration. In fractal compression algorithms specified maximal sufficient error is used when determining whether range will be processed or split, but according to error cumulation it could not be guaranteed. In fractal bottom-up compression

algorithm given criteria could be tested for whole time series, but after each compression iteration, decompression process will have to be simulated and this would slow down compression process.

In following experiments the base and the bottom-up fractal compression algorithms were compared to the bottom-up segmentation algorithm. Time series was first compressed using fractal compression algorithms and obtained values of output size, root mean squared error and maximal difference between corresponding values were then used in stop conditions of segmentation algorithm. Only bottom-up segmentation algorithm, which uses linear regression to describe values using single line in segments, was used because in experiments presented in [12] bottom-up segmentation algorithm gives the best results.

Size of compressed output file and standard compression quality properties are collected for each compression algorithm. Used algorithm is listed in column **Alg** in comparison tables. By label **FRACTAL** compared fractal compression algorithm is named (either default or bottom-up) and values listed in table 7.2 are given to it as compression parameters. The bottom-up segmentation algorithms will be distinguished by following labels:

- **S_RMS**, which means that as stop condition reaching specified maximal root mean squared error value will be considered. The RMS error value obtained from fractal compression algorithm was passed as input parameter.
- **S_MAX**, which means that as stop condition will be considered reaching specified maximal difference between corresponding values. Value of obtained maximal difference was used as input.
- **S_SIZE**, which means that size of output file will be considered as stop condition in segmentation algorithm. Size of compressed file created by fractal compression was given as input parameter.

Err	Alg	Size	RMS	Diff	Max Sl. RMS
99.95% (1.23)	**FRACTAL**	5166 B	0.56	2.87	1.88 | 1.61 | 1.23
	S_RMS	8108 B	0.57	3.93	2.51 | 2 | 1.61
	S_MAX	8191 B	0.53	2.5	2.5 | 2.02 | 1.68
	S_SIZE	5164 B	4.13	16.56	9.84 | 8.17 | 7.21
99.9% (2.46)	**FRACTAL**	4971 B	0.94	5.73	3.84 | 2.89 | 2.2
	S_RMS	7701 B	0.94	5.29	3.25 | 2.7 | 2.31
	S_MAX	7063 B	1.65	5.71	5 | 4.15 | 3.24
	S_SIZE	4969 B	4.41	16.97	10.8 | 8.99 | 7.42
99.75% (6.15)	**FRACTAL**	3263 B	4.3	19.03	12.49 | 9.73 | 7.49
	S_RMS	5045 B	4.3	16.56	10.69 | 8.99 | 7.21
	S_MAX	3574 B	7.44	19	17.57 | 13.91 | 12.94
	S_SIZE	3260 B	8.37	33.01	23.38 | 19.26 | 15.03
99.5% (12.3)	**FRACTAL**	2216 B	8.42	36.3	22.43 | 18.55 | 15.52
	S_RMS	3245 B	8.43	33.01	23.38 | 19.26 | 15.03
	S_MAX	2140 B	13.44	36	32.09 | 29.06 | 24.31
	S_SIZE	2215 B	13.06	49.79	41.73 | 31.49 | 25.55

99% (24.6)	**FRACTAL**	1729 B	12.15	58.43	37.69 | 28.4 | 24.26
	S_RMS	2365 B	12.16	48.77	37.73 | 29.77 | 22.68
	S_MAX	1490 B	19.38	58.35	53.85 | 41.19 | 35.61
	S_SIZE	1725 B	16.82	74.49	49.12 | 40.66 | 31.48

Table 7.21 Comparison of fractal base compression algorithm with segmentation algorithms on ECG data.

Err	Alg	Size	RMS	Diff	Max Sl. RMS
99.95% (1.23)	**FRACTAL**	5136 B	0.54	3.35	1.98 | 1.5 | 1.16
	S_RMS	8140 B	0.54	3.93	2.51 | 2 | 1.61
	S_MAX	8103 B	0.62	3.29	2.5 | 2.02 | 1.68
	S_SIZE	5135 B	4.17	16.56	9.84 | 8.17 | 7.21
99.9% (2.46)	**FRACTAL**	4625 B	1.24	6.2	3.59 | 3.18 | 2.37
	S_RMS	7619 B	1.24	6.58	4.5 | 3.7 | 3.09
	S_MAX	6859 B	1.86	6.19	5.19 | 4.62 | 3.48
	S_SIZE	4622 B	4.95	20.27	11.97 | 9.7 | 8.31
99.75% (6.15)	**FRACTAL**	3133 B	4.14	16.08	10.6 | 8.46 | 6.9
	S_RMS	5154 B	4.14	16.56	9.84 | 8.17 | 7.21
	S_MAX	4017 B	6.39	16.03	13.54 | 12.05 | 10.31
	S_SIZE	3130B	8.83	35.14	23.38 | 19.26 | 15.94
99.5% (12.3)	**FRACTAL**	2089 B	8.78	35.88	20.47 | 17.55 | 15.72
	S_RMS	3135 B	8.8	35.14	23.38 | 19.26 | 15.94
	S_MAX	2150 B	13.38	35.83	32.09 | 29.06 | 24.31
	S_SIZE	2085 B	13.83	74.49	41.73 | 31.64 | 25.55
99% (24.6)	**FRACTAL**	1298 B	19.76	112.81	88.45 | 72.46 | 56.53
	S_RMS	1538 B	19.81	102.87	59.61 | 50.12 | 45.41
	S_MAX	831 B	41.15	111.7	100.48 | 91.51 | 81.1
	S_SIZE	1297 B	24.32	123.29	79.86 | 65.77 | 55.73

Table 7.22 Comparison of fractal bottom-up compression algorithm with segmentation algorithms on ECG data.

As we can see in these tables, when segmentation algorithm is run with specified stop value of root mean squared error achieved using fractal compression algorithm, size of output of segmentation algorithm is always bigger than output size of fractal compression. Coefficient between corresponding outputs has value around *1.6*. In comparison with fractal base compression algorithm gives this segmentation algorithm almost always better results for maximal difference between corresponding values and also for maximal sliding RMS error between the corresponding subsequences. On the other hand fractal bottom-up compression algorithm is better mostly also in these quality properties.

When stop condition was set to the maximal difference between corresponding values, segmentation algorithm leads to bigger size of output when this difference is set to smaller values. Also RMS error and sliding RMS error values are much greater than for fractal compression algorithms.

Fractal compression algorithms give better quality results also when input time series is processed by segmentation algorithm to fit given output size. For smaller values of maximal sufficient error used in fractal compressions are these differences very big and with increased value of sufficient error they get little closer.

For these experiments simple bottom-up segmentation algorithm, as described in second chapter, was implemented. It describes each segment using single line which is stored as values of its end points. To calculate ideal line for particular segment, linear regression algorithm from linear regression library [13] was used. Result application simulates only encoding process and time series described by result set of segments is computed at the end of segmentation algorithm.

7.8 Fractal compression vs. lossless compression methods

In the last experiments were compared output sizes of fractal compression algorithms with lossless compression methods. Given values of sufficient maximal error were small and was corresponding to *99.95%* accuracy. As lossless compression methods were chosen the ***bzip2*** [17], the ***rar*** [18], the ***zip*** [19] and also the simple Huffman coding.

	Fractal	bzip2	rar	zip	Huff
ECG	5166 B	3325 B	4144 B	4308 B	3771 B
Resp	18650 B	19036 B	30229 B	26462 B	25109 B
Power	49564 B	35444 B	34728 B	48010 B	43788 B
SedTr	24227 B	30740 B	26338 B	36973 B	39184 B

Table 7.23 Comparison of base fractal compression with lossless compression methods.

Results in table 7.23 could be summarized as follows. When maximal sufficient error value given to fractal compression algorithm is small value, then usually lossless compression methods lead to better output sizes. For respiration data and sediments transports data, error value corresponding with *99.9%* accuracy has great value (*205.1* or *42.88*) and that could be main reason, why for these time series gives fractal compression smaller output sizes, but also quality of decompressed result is more smaller than in lossless compression methods.

When very good approximation is desired than it may be better to compress the data using some lossless compression method, because fractal compression will lead to greater output size, so better choice will be to have exact data compressed using less space.

7.9 Internal measurements

In this last experiments section some internal measurements of the fractal compression algorithms will be presented. Information about how number of ranges described using transformation and domain position changes with different input error values, how many values are stored as exact values, how size of compressed file changes with different error values, will be presented in this section.

In the figure 7.5 we could notice that the difference between number of ranges processed with sufficient transformation for base and bottom-up compression algorithms is almost constant for accuracies between *99.75%* and *99.25%*. Then

number of ranges described using transformation in the default compression algorithm is decreased slower than for the bottom-up algorithm. Reason for this is that number of ranges is lower bounded with *length / 16* for the base compression algorithm but for the bottom-up it is not. Number of values that need to be stored exactly with increasing error goes down very fast and for accuracy *99.75%* is this number usually very low.

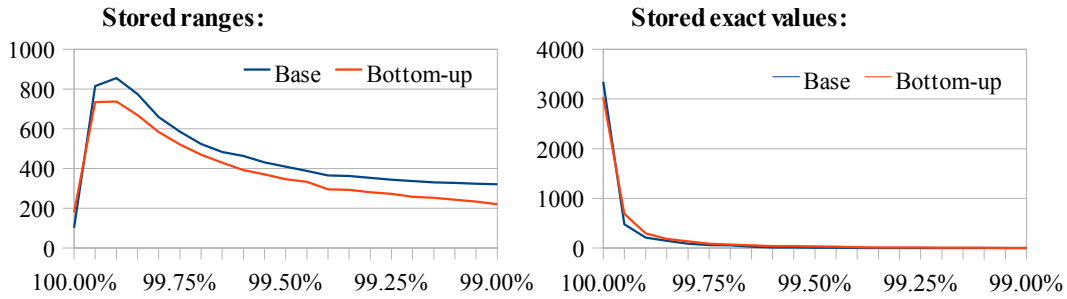

Figure 7.5 Number of ranges described using transformation and domain position and number of stored exact values for different values of sufficient error for ECG data.

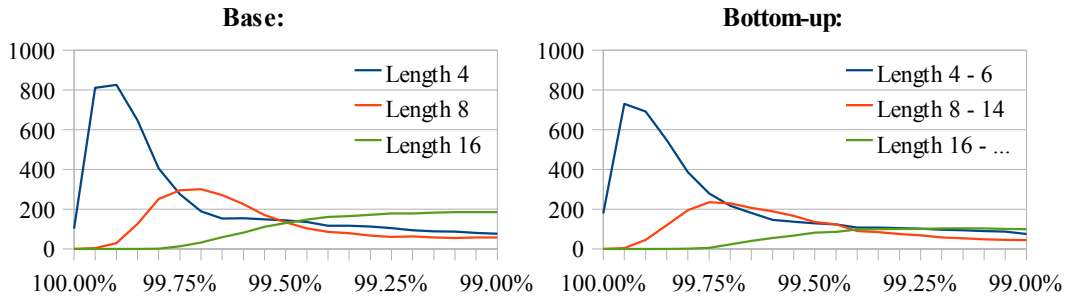

Figure 7.6 Number of ranges described using transformation and domain position for different values of sufficient error for ECG data.

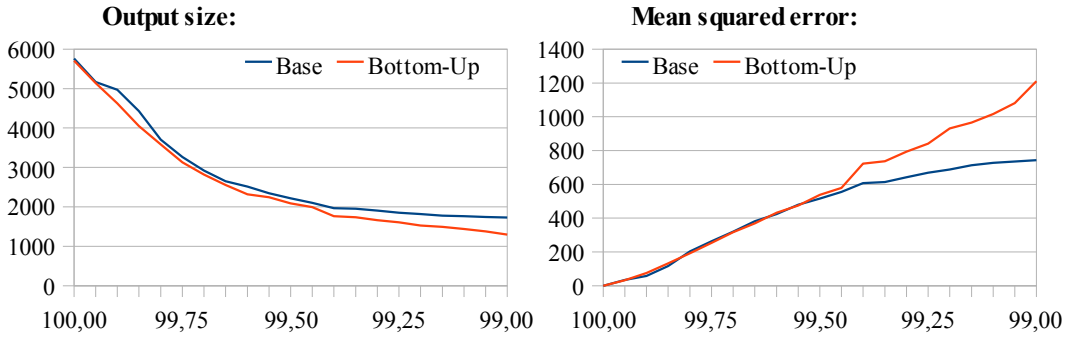

Figure 7.7 Comparison of output size and obtained mean squared error for different values of sufficient error for ECG data.

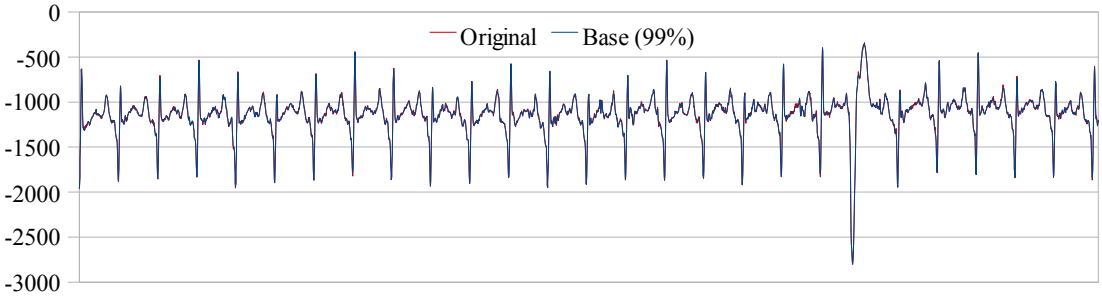

Figure 7.8 Decompressed ECG data, compressed using base algorithm with 99% accuracy error value. Behind with red colour is drawn original ECG time series.

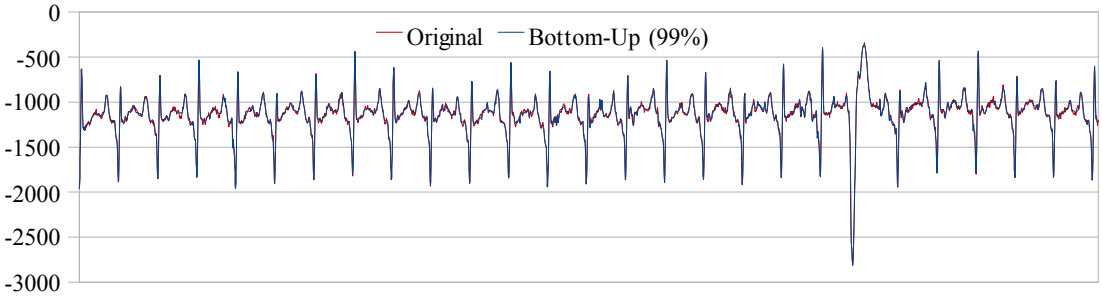

Figure 7.8 Decompressed ECG data, compressed using bottom-up algorithm with 99% accuracy error value. Behind with red colour is drawn original ECG time series.

8 CONCLUSION

The aim of this thesis was to try to use characteristics of fractals to describe real world time series using the set of contractive transformations and to use these principles to compress these time series.

Four compression algorithms based on fractal principles were described and implemented. According to performed experiments, we could notice that added heuristics do not affect very much quality of decompressed data, but reasonably increase compression speed or that derived algorithms really reduce impact of error cumulation in the base compression algorithm and cost of space is almost the same or that bottom-up processing leads to smaller output sizes but worse quality of decompressed data, even still sufficient according to given input parameters.

In comparison with the segmentation processing of time series give presented fractal algorithms better results in almost all tests. Advantages of fractal algorithms is that particular subsequences need not to be described by straight line as in the segmentation algorithms.

For very small values of maximal sufficient error give presented fractal compression algorithms worse output sizes than common lossless compression methods. So when some time series need to be stored with higher accuracy, better solution should be to use some lossless compression method. Only for greater values of sufficient error produces fractal compression methods output of smaller size.

There are many options for further investigation of presented algorithms. Different processing of ranges could be used. Combined error computer could be designed, which will use mean-square error but also will not allow huge differences between corresponding values. Some improvements to reduce error cumulation should be added to the bottom-up algorithm or some better domain classification could be used. Like this we can create a long list of possible improvements or different approaches that could be used in presented algorithms.

REFERENCES

[1] Peitgen H.-O., Jürgens H, Saupe D.: *Chaos and Fractals* - second edition, New Frontiers of Science, 2004.

[2] Nelson M., Gailly J.-L.: *The Data Compression Book*, M&T Books, 1995.

[3] Last M., Kandel A., Bunke.H.: *Data Mining In Time Series Databases*, World Scientific Publishing Co. Pte. Ltd., 2004.

[4] Cipra T.: *Analýza časových řad s aplikacemi v ekonomii*, SNTL Praha, 1986.

[5] Kigami J.: *Analysis on Fractals*, Cambridge University Press, 2001.

[6] Addison P. S.: *Fractals and chaos: an illustrated course*, Institute of Physics Publishing Brystol and Philadelphia, 1997.

[7] Barnsley M.: *Fractals Everywhere*, Academic Press, 1988.

[8] Culik K., Dube S.: *Methods for Generating Deterministic Fractals and Image Compression*, Dept. of Computer Science, Univ. of S. Carolina, 1990.

[9] Wikipedia, *Fractal*, http://en.wikipedia.org/wiki/Fractal.

[10] Keogh E., Chu S., Hart D., Pazzani M.: *An Online Algorithm for Segmenting Time Series*, The 2001 IEEE International Conference on Data Mining.

[11] Keogh E.: *Data Mining Large Medical Time Series Database*, http://www.cs.ucr.edu/~eamonn/discords.

[12] Gedikli A., Aksoy H., Unal N. E.: *Segmentation algorithm for long time series analysis*, Springer-Verlag, 2007.

[13] van Bergen A., *Linear regression algorithm*, http://myweb.tiscali.co.uk/vanbergen/linmodel.htm.

[14] Hyndman R.J. (n.d.): *Time Series Data Library*, http://www.robhyndman.info/TSDL.

[15] VÚB Bank web site, http://www.vub.sk.

[16] U.S. Geological Survey Oceanographic Time-Series Data, http://stellwagen.er.usgs.gov.

[17] bzip2 homepage, http://www.bzip.org.

[18] WinRAR homepage, http://www.win-rar.com.

[19] WinZip homepage, http://www.winzip.com.

APPENDIX - CONTENT OF INCLUDED CD

In included compact disc could be found following content ordered according to particular directories:

Experiments - contains the data which were used in experiments chapter ordered according to corresponding sections. List of executed commands, obtained compressed and decompressed data, other output files and excel sheets with obtained results ordered in well-arranged tables could be found here.

Sources - contains sources of created applications. Each application is given as *MS Visual Studio 2008* solution, supplemented with built executable binaries in ***Bin*** folder and with brief informative documents in ***Docs*** folder. In each of these applications emphasis was placed on performing its main tasks, so these applications are very simple.

FractalCompression - C++ solution for console application, where all presented fractal compression algorithms are implemented.

FractalCompressionGUI - C# solution for window application, which allows user to select or insert compression parameters for *FractalCompression* console application.

HuffmanSimulator - C# solution for console application used in experiments to calculate size of integer sequence encoded using *Huffman coding*.

Segmentation - C++ solution for console application used in experiments to process bottom-up segmentation algorithm on given time series.