DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Volume Data Compression

Maarten Bussler

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Volume Data Compression

Komprimierung von Volumendaten

Author:	Maarten Bussler
Supervisor:	Prof. Dr. Rüdiger Westermann
Advisor:	M.Sc Christian Reinbold
Submission Date:	15.08.2020

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.08.2020 Maarten Bussler

Acknowledgments

I want to acknowledge

- ...Christian Reinbold, my advisor, for his availability, patience and time he put into his helpful feedback,
- ...Dr. Rafael Ballester-Ripoll for his insight into the matter of TTHRESH and the structure of volume data,
- ...my friends and family for always keeping my mood up.

Abstract

Despite extensive research in the field of volume data compression, modern volume visualization faces the challenge to handle both memory and network bottlenecks when visualizing high resolution volume data at a smooth visual feedback rate. Techniques to efficiently compress and decompress volume data are much in demand. TTHRESH by Ballester-Ripoll et al. [BLP19] tackles this problem by providing a lossy compression technique based on the HOSVD and adaptive thresholding of volume elements that offers high compression ratios at the cost of a comparably low approximation error. We hypothesize the possibility of improvement of the achieved compression ratio when incorporating known structures of the data, like the hot corner phenomenon, into the algorithm. Furthermore, we analyze the benefit of combining core truncation as another established compression technique with TTHRESH.

Kurzfassung

Trotz intensiver Forschung im Bereich der Datenkompression steht die moderne Volumengrafik der Herausforderung von limitierter Größe in physikalischen Speicher und Bandbreite gegenüber, die für ein flüssiges visuelles Feedback benötigt werden. Algorithmen für eine effiziente Datenkompression in Hinsicht auf Kompressionsrate, Kompressionsfehler und Kompressiongeschwindigkeit sind gefragt. TTHRESH [BLP19] ist ein Kompressionsalgorithmus speziell für Volumendaten. Die Technik basiert auf der Zerlegung des Volumens durch HOSVD, sowie Reduzierung der unwichtigen Volumenanteile und erreicht so kompetetive Kompressionsraten zu kleinen Einbußen in der Kompressionsqualität. Diese Arbeit verfolgt das Ziel, den vorhandenen TTHRESH-Kompressionsalgorithmus hinsichtlich der aufgebrachten Kompressionsrate weiter zu verbessern. Dabei werden Vorteile von vorhandenen Strukturen der Daten, wie dem 'aktive-Ecke' Phänomen, oder die Kombination mit anderen Kompressionstechnicken analysiert.

Contents

1. Introduction

Volume rendering is a field of computer graphics and data visualization. It constitutes a set of methods and algorithms to render a 2D representation of a discrete 3D scalar data set (*volume data set*). 3D data that is difficult to represent with geometric surfaces (like MRI Scans, fluid dynamics, or gas) is hard to render with conventional methods. Volumetric visualization parses and analyzes such complex data, while allowing experts to reveal complex 3D relationships within the data set. Up-to-date scanner hardware and simulation software advances at a rapid speed and volume data is increasing in size and complexity. Subsequently, scientific and visual computing tasks on volume data are faced with the problem on how to handle such complex data while still providing the fluent interactivity and visual feedback needed for conducting analytic tasks on the data. As resources on physical memory and bandwidth are limited, modern research on volume data and data visualization is confronted with the challenge of efficient data reduction and compression.
Often times volume data is used as the basis for a simulation or is subject to successive calculations. It is thus required to bound the compression error during data reduction on a minimal approximation error. Hence, although lossy and broad compression of volume data offers great reduction in filesize, it is not always an adequate solution. The TTHRESH volume data compression algorithm by Ballester-Ripoll et al. [BLP19] provides a smooth and flexible compression of volume data according to a given target error, while still achieving compression ratios comparable to algorithms with broader approximations of the volume. Tucker decomposition (see Section 2.2) is a popular tensor tool for dimensionality reduction that approximates a given input volume data set T by a set of orthogonal factor matrices U^i and a core tensor B and functions as a basis of TTHRESH. The algorithm then applies adaptive bit-plane coding followed up by run length encryption and arithmetic coding on the volume data and compresses the data by thresholding core coefficients and accuracy of less importance. The goal of this thesis is to improve the TTHRESH compression algorithm in regards of its achieved compression ratio, while preventing a drop in compression quality.
This thesis is structured in the following way: In chapter 2, the theoretical background of the structure of volumetric data as well as the Tucker decomposition as a mean to decompose volume data is introduced. Furthermore, various compression methods, which are then implemented in the volume compression algorithms, are explored. The volume compression methods of core truncation and TTHRESH are analyzed and discussed in more detail in chapters 3 and 4. With this theoretical background we finally try to improve the TTHRESH algorithm in chapter 5. Following are a summary and discussion of the results in chapter 6 and a conclusion of the whole thesis in chapter 7. We provide access to our code on [Bus20].

2. Theoretical Background

2.1. Volumetric Data

To understand the procedure of volume compression it is first important to realize the general structure of volumetric data sets. Many visual effects (like clouds, gas of fluids) are volumetric in nature and are difficult to model with geometric primitives. As a solution, a specific form of representation, the volume data sets, were introduced. They are 3D entities, that do not consist of tangible surfaces and edges. Typically, volume data is represented by a discrete set T of voxels (x, y, z, v) (Fig. 2.1). v represents the value of some attribute (e.g. color, density, heat or pressure) of the data at the 3D position (x, y, z). Voxel-values may be taken from random locations in space, with regularly spaced intervals between the samples on each dimension. As the voxels are defined on a regular grid and are only specified on the discrete grid locations, data structures like 3D arrays (further referred to as *tensors*) are often used to store these data sets.
Volume data is obtained by numerous sampling, simulation and modelling techniques. In the medical field, MRI and CT-Scans acquire data as 2D slices, which are later reconstructed in a volumetric fashion for the purpose of better visualization. Many computational fields, like fluid dynamics, also depend on volume data, as the results of simulations are often visualized 3D volumes for later analysis and interpretation [HJ11].

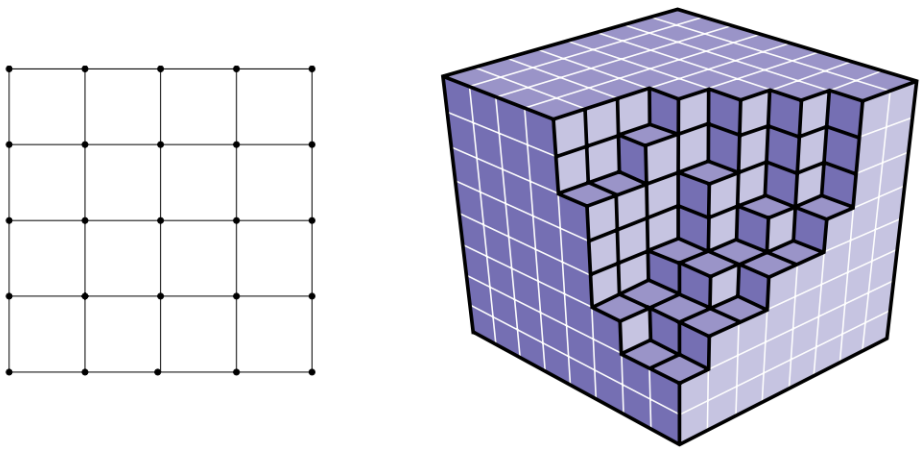


Figure 2.1.: Structure of 3D volume data (right) and discrete 2D voxel-grid (left). One can notice that the values of the voxels are not continuously defined over the whole volume, but only at specific grid locations [Eng+04].

2.2. Tucker Decomposition

Many transform-based compression algorithms for volumetric data rely on some form of data-dependent bases that decompose the data set into smaller approximations. HOSVD (Higher-Order SVD) is one such approximation and is a generalization of matrix SVD for tensors.
According to the *SVD* (Singular Value Decomposition), every 2D matrix A can be represented by its eigenvalues and eigenvectors [De 09]:

$$A = U\Sigma V^T. \tag{2.1}$$

U and V^T are orthogonal matrices that represent the left and right *singular vectors*, Σ is a diagonal matrix that holds the *singular values* σ_i of A. U, V^T and Σ can be computed easily by multiplying A with it's transposed form and solving for either U or V^T:

$$AA^T = U\Sigma V^T * V\Sigma^T U^T = U(\Sigma\Sigma^T)U^T. \tag{2.2}$$

U now holds the eigenvectors of AA^T, while Σ carries the squared eigenvalues σ_i as the singular values of A. Respectively, V holds the eigenvectors of A^TA:

$$A^TA = V\Sigma^T U^T * U\Sigma V^T = V(\Sigma^T\Sigma)V^T. \tag{2.3}$$

Just like with the SVD for 2D matrices, every tensor $T \in \mathbb{R}^{I_1 \dots I_n}$ can be decomposed and represented by a basis core-Tensor $B \in \mathbb{R}^{I_1 \dots I_n}$, holding the eigenvalues, and a set of matrices U^i (factor matrices) of size $I_i \times I_i$ that represent the eigenvectors of the tensor along each dimension (Fig.2.2) [DDV00]. This is called a *Tucker decomposition* of the tensor. A significant compression ratio can be obtained by approximating the tensor and decreasing the size of the basis and the factor matrices with the help of truncation (see Section 2.3.4).
In order to understand the decomposition of a tensor into its core and factor matrices, it is first important to look into the process of multiplying a 3D tensor with a 2D matrix (TTM). We can describe a 3D tensor as a set of 2D matrices (*slice* of a tensor). A single slice of the tensor is then defined by traversing over the elements of the tensor, by iterating along two dimensions (modes) and fixing all others. The 2D matrix rows and columns higher order analogues of a 3D tensor are called *fibres*. Similar to tensor slices, they can be obtained by iterating along a single dimension (mode) of the tensor and keeping all others fixed (Fig. 2.3) [KB09]. The k-mode product of a tensor T with a matrix A multiplies each mode-k fibre of T with A and is denoted as $T \times_k A$ [Zha+14]. Consequently, with the help of the TTM notation, the Tucker decomposition of T can be written as

$$T = B \times_1 U^1 \times_2 U^2 \times_3 U^3. \tag{2.4}$$

By inverting the equation 2.4, the factor matrices U^i characterize a two-way transformation between core B and T

$$B = T \times_1 U^{1^{-1}} \times_2 U^{2^{-1}} \times_3 U^{3^{-1}}. \tag{2.5}$$

It is possible to represent a 3D tensor T as a 2D matrix by ordering all i-mode fibres of T

as columns of the matrix (Fig. A.1). This is called the *i-th mode unfolding* T_i of T [KB09]. *HOSVD* is an efficient and straightforward approach to calculate a Tucker decomposition with orthogonal factor matrices. This is achieved by setting each U^i as the left singular vectors gained from the SVD of the i-th mode unfolding T_i. Finally, B can easily be computed with the help of equation 2.5. It is important to note that a HOSVD decomposition always exists for any form of tensor [BLP19].

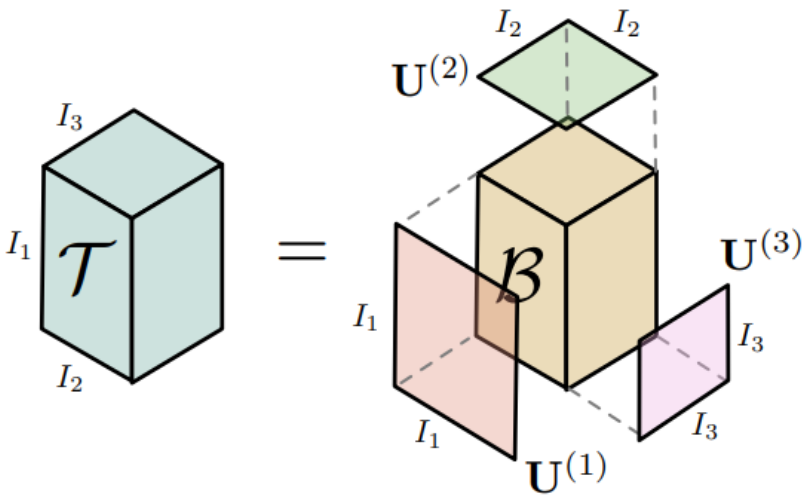


Figure 2.2.: Schema for HOSVD, with the full original tensor on the left and decomposition into core and factor matrices on the right. Adapted from [BLP19].

2.2.1. Properties of Tucker Decomposition

Core and factor matrices constructed by HOSVD provide a number of useful and exploitable properties for use of efficient compression.

- It is important to note that the HOSVD is unaffected by many transformation techniques [BLP19]. Spatially moving the data, padding with zeroes, scaling by some constant, or filtering by upsampling with linear interpolation will result in the same core B, therefore algorithms based in the HOSVD are convenient for data analysis and visualization purposes.
- The generated core B can be structured in such a way that the norms σ_i of the slices of the core are decreasing in magnitude in logarithmic fashion along each axis. [BLP19] The norms in this case act as a generalization of the scalar values in the slice. Thus, the largest core coefficients that contain most of the original tensor's energy are centered around the first element $B(1, 1, 1)$ of the tensor. This upper left corner will further be regarded as the *hot corner* (Fig. 2.4). For an efficient compression of the volume data, special consideration should be given to coefficients situated around the hot corner and the otherwise sparse structure of the core.

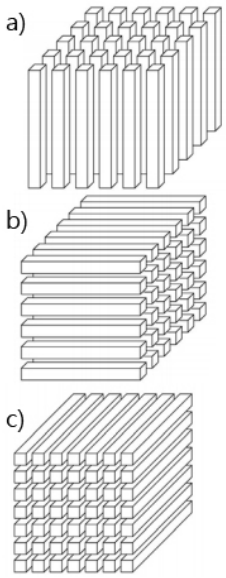


Figure 2.3.: Representation and structure of a tensor according to its a) mode-1 fibres b) mode-2 fibres c) mode-3 fibres. Adapted from [KB09]

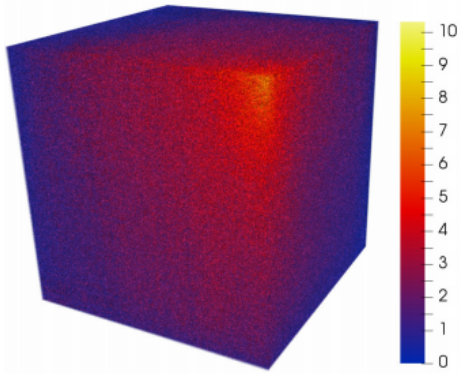


Figure 2.4.: Visualization of the HOSVD core B element magnitudes. All elements were scaled with $x \rightarrow ln(1+x)$ and enhanced with the colormap on the right. As suggested by the hot corner phenomenon, the core elements of largest magnitude concentrate around $B(1,1,1)$. Adapted from [BLP19]

2.3. Data Reduction

Especially in the domain of volume data, interactivity and fluent visual feedback between data set and data consumer is crucial when visualizing data and performing analytic tasks. With recent technological advancements scanners and computers become increasingly powerful, generating very explicit and in-depth volume data. As a result, storage sizes on disk, as well as loading and processing times are multiplying. Data reduction tries to address these I/O-related difficulties. By reducing in-memory sizes of data sets, lower transfer times to and from the disk are achieved.
In general, data reduction techniques can be assigned to one of two groups [Li+18].

- **Lossless** compression is achieved by utilizing patterns in the data. No information is intentionally discarded and apart from floating-point rounding errors the reconstructed data after compression evaluates to the same data as the original data.
- **Lossy** compression achieves data reduction by cutting off insignificant values in the data set. Thus, the reconstructed data won't match the original data perfectly, but much higher compression rates than lossless methods can be realized. While these techniques usually try to minimize the introduced error, the amount of information loss differs greatly between applications, and often the exact reduction parameters have to be modulated on a case by case basis.

The following paragraph gives a general overview of the compression techniques that are discussed in this paper. Each technique has different tradeoffs in respect to compression-rate, processing time and introduced error.

2.3.1. Run-length Encryption

Run-length encryption (RLE) is a lossless and simple encryption algorithm that loops through the input data and results in a sequence of consecutive data values in a row (*run*) as output. Compression is achieved by reducing the physical size of these runs in the original data. Runs are encoded in two parts. The first part represents the number of characters stored in the run, the second part is the value of the character in the run. For example, the data (0000111) would be encoded as the sequence [4 0, 3 1]. Context and type of data greatly affect the obtained compression ratios. The more consecutive values in a row, the longer the run and the more space is saved in the resulting compression.
Modified run-length encoding [SW09] is an important variant of the traditional RLE and will be used by later compression algorithms in this paper. Unlike the traditional RLE, the modified version of RLE only works on binary strings and stores distances between bits of value "1". The *distance* of two bits of value "1" is defined as the frequency of value "0" bits between these two bits. The beginning of the bit sequence is interpreted as a bit of value "1". For instance the above data (0000111) would be encoded as [4,0,0] in modified run-length encryption, since the first value "1" bits is distant of four value "0" bits to the first implicit "1" bit at the start of the sequence. Directly thereafter follow two "1" bits without any value "0" bits between them. Modified run-length encoding benefits from sparse data sets, since with a decreasing number of "1" value bits, the total number of encoding symbols in the output decreases, too.

2.3.2. Arithmetic Coding

Arithmetic coding (AC) is an entropy encoding technique. The information content, or *entropy* H of a source S is defined as

$$H = -\sum_{i=1}^{n} p_i \log_2(p_i). \tag{2.6}$$

S is composed of independent and identically distributed symbols a_i with each symbol having a probability of occurrence p_i [HV94]. H gives a lower bound on the expected number of bits needed to represent each a_i and thus indicates the optimal bit representation of the input sequence. Entropy coders remove redundancy from the signal by encoding the symbols a_i as a sequence of codewords c_i that minimizes the average number of bits needed per codeword. As a result, the number of bits needed to represent each symbol in the encoded sequence converges to the entropy H.
Arithmetic coding tries to achieve this optimal bit representation by encoding frequently occurring symbols with fewer bits than lesser occurring symbols. The algorithm takes a series of symbols as input and incrementally builds a floating point number in the range [0, 1] as output. AC relies on a stochastic model to distinguish the symbols it is processing. The task of the model is to give an accurate probability distribution of the symbols in the

input message. This is realized by assigning each specific symbol to a unique segment in the interval [0,1), thereby effectively trying to predict the probability that a specific symbol will appear. The optimality of the resulting encoding process is tied together closely with the accuracy of the model and many different ways of modeling the symbol frequencies exist. Some models are non-adaptive and assign fixed probabilities to all symbols. Others are adaptive and gather statistics about the input file or dynamically evaluate the probability for an event by inspecting the events that preceded it. The decision which model to choose is application based, but for the algorithm to work, both the encoder as well as the decoder have to have identical models [Li+18].
In the *encoding* process the first symbol of the input defines a range of the start-interval [0,1) that was assigned to it according to the model. Following symbols repeat this step by further dividing the interval of the previous step proportional to their own segment defined by the model (Fig. 2.5). After processing the whole input sequence the encoder outputs a floating point number that is contained in the final interval.
The *decoder* reverses the subdivision of intervals. In each step the model detects the symbol, whose assigned interval covers the current value of the message. For further decoding of the following symbols the same narrowing of the interval as in the encoder is performed. As with the encoder, this shrinking of the intervals continues until the end of the file is reached.

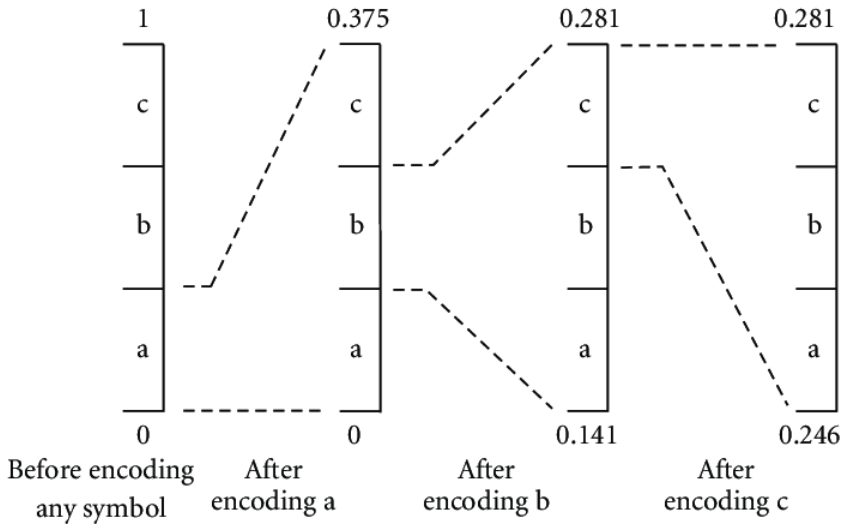


Figure 2.5.: Example encoding of the sequence "abc" with AC. Before encoding, the start-interval is initiated with [0,1) and is set as the current interval. The interval is then divided according to the probability distribution of the encoding symbols. The final interval to encode the input sequence is calculated by successive subdividing of the current interval proportional to the interval segment size of the current encoding symbol [SV10].

Huffman coding is another popular entropy coding technique. Similar to AC, Huffman coding aims to encode frequent symbols with short bit-strings, thereby reducing the physical size of the input data. The main difference between Huffman coding and AC lies in their models,

called the Huffman tree. The Huffman tree is built on the basis of a tree like structure and is used for compression and decompression. In order to generate the tree, frequencies of the input symbols are gathered and a tree leaf-node for each distinct symbol is created. The algorithm then incrementally merges two nodes of the lowest frequencies into a higher level node, whose frequency value is the combined frequency value off the children nodes. The merging process continues until only the root node is left. This way, every node apart from the leaf-nodes provides edges to exactly two children (Fig. 2.6).
The *encoder* labels left and right edges of the tree with the binary values "1" and "0". Each input symbol is then encoded by traversing the Huffman tree from the root node to the corresponding leaf-node and generating an equivalent binary string according to the edge-values. The structure of the Huffman tree guarantees that frequent symbols offer paths of short distances to the root node, thus resulting in shorter codewords [SW09].
Decompressing a Huffman-encoded file first requires rebuilding the Huffman tree from stored frequencies (usually found in the header of the file). A stream of bits is then read from the encoded file and the Huffman tree is traversed accordingly from the root node, until a leaf-node is reached. Finally the symbol of the leaf-node is read and swapped with the bitstream for the decoded file. The procedure repeats, until the end of file is reached.
Although both algorithms seem very similar, both techniques provide different strengths and weaknesses. In general, Arithmetic coding offers more efficient encoding, while Huffman coding boasts superior reconstruction speed [BP16].

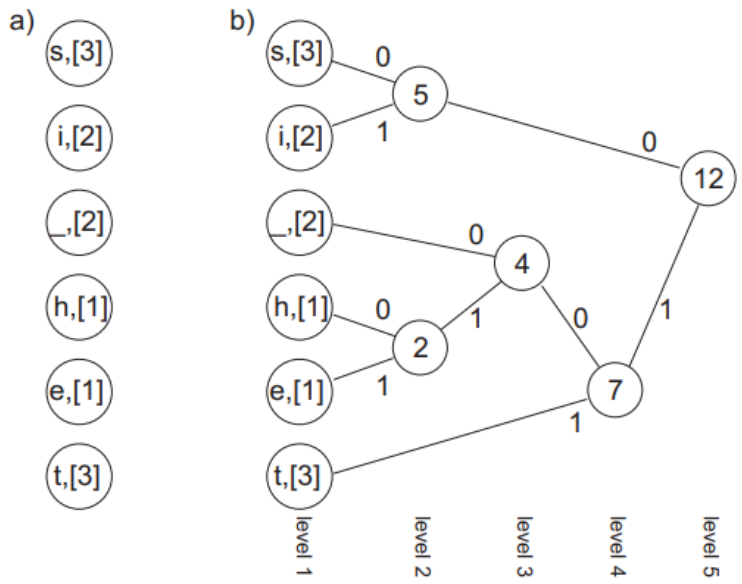


Figure 2.6.: Huffman encoding of the input sequence "this_is_test". In a) the frequencies of the symbols are counted and corresponding nodes generated. In b) the Huffman tree is built and Huffman codes are assigned to the input symbols. Adapted from [SW09].

2.3.3. Quantization

Quantization is a lossy compression technique that limits the complexity of its input data by reducing the precision of the data values [Li+18]. This is typically attained by mapping floating-point values to a finite set of fixed output values (*bins*) that serve as approximations for the input data (Fig. 2.7). For example, rounding of floating-point values to integer values could be seen as a simple form of a quantization process. The device or function that performs the specific mapping of the data is called the *quantizer*, while the difference between the original value and the mapped quantized value (e.g. the rounding error) is referred to as the *quantization error*. Considering the many-to-few mapping of the input data, quantization is a nonlinear and irreversible process. As a consequence it's impossible to recover the exact original value of the input data, since the same output value is assigned to numerous input values.
The input data sets often differ greatly in regard of their value distributions. To this end, and in order to improve the approximation ability of the bins, the mapping scheme can be adjusted to best encompass the underlying data and reflect the uniform or non-uniform value distributions. Quantization is rarely used as-is but forms the basis for numerous complex compression techniques and is usually combined with other compression algorithms to further encode transformed data at the cost of a newly introduced error.

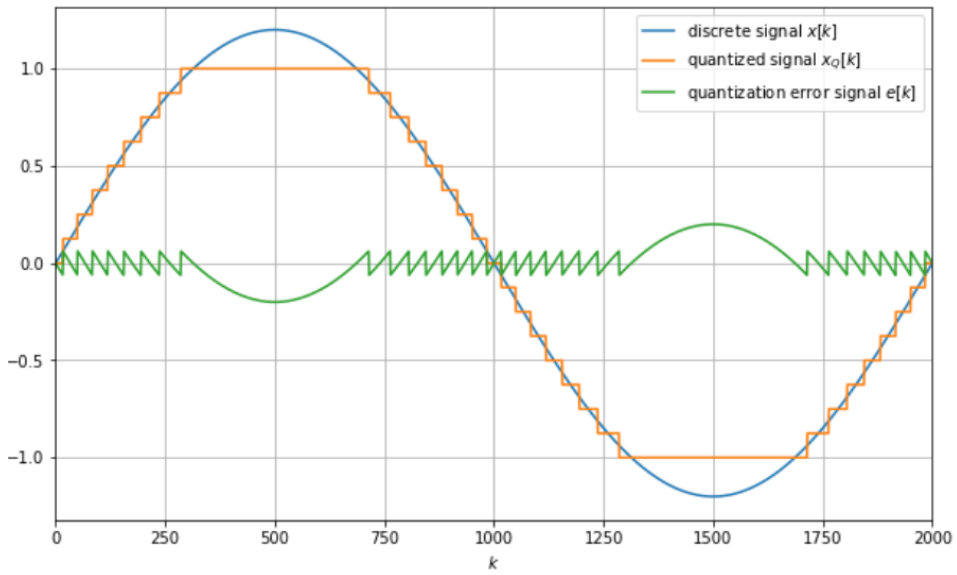


Figure 2.7.: Visual example for the process of quantization. A sine wave (blue) is approximated and encoded with quantization (orange). As each output value (bin) is encoded with 3 bits, there are 8 bins in total. The green plot refers to the quantization error. Adapted from [Bha18].

2.3.4. Truncation

Truncation is a popular and widely used algorithm for lossy data compression. As in quantization, substantial compression ratios are achieved by reducing the precision of the input data. In general, in the field of signal processing, truncation is employed to divide floating-point values into multiple precision levels. As an example MLOC (Multi-level Layout Optimization Framework for Compressed Scientific Data) [Gon+12], a truncation algorithm built for arbitrary and hard-to-predict data layouts, divides double-precision 64-bit values into seven level. The first level holds the first two byte of the data and is constructed out of one bit for the sign, eleven bit for the exponent and three bit for mantissa (Fig. 2.8). Each consecutive level represents an additional byte of the original data. Not required levels are discarded, but each saved byte after the first level increases the precision of the mantissa, thus decreasing the introduced error.

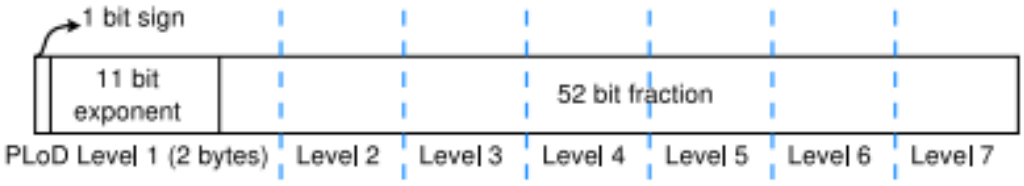


Figure 2.8.: Truncation according to MLOC: double-precision values are partitioned into seven levels, with the first level holding two byte and each consecutive level carrying an additional byte [Gon+12].

In the context of volume data compression the idea of truncation is transformed and applied on 3D data as a whole, not just on scalar values. As characterized in Section 2.1, volume data is structured as a 3D array that consists of multiple 2D matrices (slices). Volume data is often very sparse and eliminating insignificant values of the data set is a common approach used in various compression techniques. The simplest way that comes to mind to realize this goal is to truncate the 3D volume data by discarding whole slices of the data set with the least significant values. Consequently, the missing values are approximated with "0" values in the reconstruction step.
By preprocessing the volume data and decompositing it with the help of HOSVD (see Section 2.2), an efficient categorization of important and insignificant data sections for reduction can be generated. However, recent studies [BP16; BLP19] have proven that thresholding and eliminating the coefficients of the tucker core on a coefficient by coefficient basis yields better results in regard of relative error and compression ratio than the slice-wise truncation of the core. Nevertheless, truncation is an important basis-algorithm that can further be expanded, and together with R.Ballester's latest adaptive thresholding volume data compression algorithm [BLP19] motivated further research in the field of volume data compression.

3. Implementation of Tensor Truncation

As mentioned in earlier chapters, volume compression by truncation is a widespread and extensively studied technique that shall serve as a reference algorithm for comparison of other compression algorithms in this thesis. The hereinafter described algorithm is adapted from [BP16].

3.1. Tensor Rank Truncation

The decomposition of a tensor T with the help of HOSVD yields in a core tensor B as well as factor matrices $U^{\{1,2,3\}}$. Similarly to the truncation techniques discussed in Section 2.3.4, the goal of tensor rank truncation is to compress volume data by approximating the tensor by removing dispensable core slices and corresponding columns in the factor matrices. Just as mentioned in Section 2.3.3, the largest core coefficient values tend to concentrate around the hot corner $B(1,1,1)$ of the core. Because of this property, the ideal subset of slices to discard must be chosen from the opposite side of the hot corner [BP16]. As a result only $1 \leq R_n \leq I_n$ core slices along each dimension and corresponding factor columns have to be encoded. This *tensor rank truncation* is illustrated in Figure 3.1 and is inexpensive to construct and reconstruct. Pseudocode to perform core truncation is depicted in Figure 3.2.
Finding good enough choices for R_1, R_2, R_3 that raise the compression ratio while reducing the introduced relative compression error is no trivial task. A practicable solution is to iterate over all truncation possibilities, and picking the best one for a given target error or target compression ratio. Because of the orthogonality of the factor matrices, and $T = B \times_1 U^1 \times_2 U^2 \times_3 U^3$ it follows that $\|T\| = \|B\|$ [BP16], where $\|\cdot\|$ refers to the Frobernius norm: $\|T\| = \sqrt{\sum_{i,j,k} T_{i,j,k}^2}$. Moreover, any error introduced in B passes down to T. This allows for a fast and easy way to compute any introduced relative error, without having to reconstruct the original tensor first:

$$\epsilon = \sqrt{\|T\|^2 - \|B\|^2} / \|T\|. \tag{3.1}$$

Since the relative error ϵ_i for every possible subcore B_i depends on the norm of the subcore, any efficient algorithm to compute all different truncation choices needs to quickly obtain the norm of the different subcores B_i. This is done by utilizing a summed-area table (SAT) that stores at any point (x,y,z) the sum of all values located in the cuboid spanned by the hot corner (1,1,1) and (x,y,z). For a 3D space, this can be done in $O(I_1 I_2 I_3)$ time by computing

$$\begin{aligned} SAT(x,y,z) = B(x,y,z) + SAT(x-1,y-1,z-1) + SAT(x,y,z-1) + SAT(x,y-1,z) + \\ SAT(x-1,y,z) - SAT(x-1,y-1,z) - SAT(x,y-1,z-1) - SAT(x-1,y,z-1) \end{aligned} \tag{3.2}$$

for every coefficient $(x, y, z) \in B$.
The algorithm then evaluates the relative error ϵ and compression factor F for all $I_1 I_2 I_3$ possible truncation choices. This is accomplished by counting the remaining coefficients in the truncated core and factor matrices versus the number of coefficients prior to truncation: $F = (R_1 R_2 R_3 + I_1 R_1 + I_2 R_2 + I_3 R_3)/(I_1 I_2 I_3)$. Subsequently, the resulting list is ordered ascending to F and every best ϵ achieved so far is registered, while all entries that do not improve the current relative error are discarded. This way, only the best ϵ for every F are stored. All solutions for which a better counterpart in terms of both F and ϵ exists, are discarded. In particular, this means that the final list contains no two pairs (ϵ_i, F_i) and (ϵ_j, F_j) where $\epsilon_j < \epsilon_i$ and $F_j < F_i$. According to the target error or target compression ratio a solution of the sorted list is then selected and the core is truncated correspondingly (Fig. 3.3).

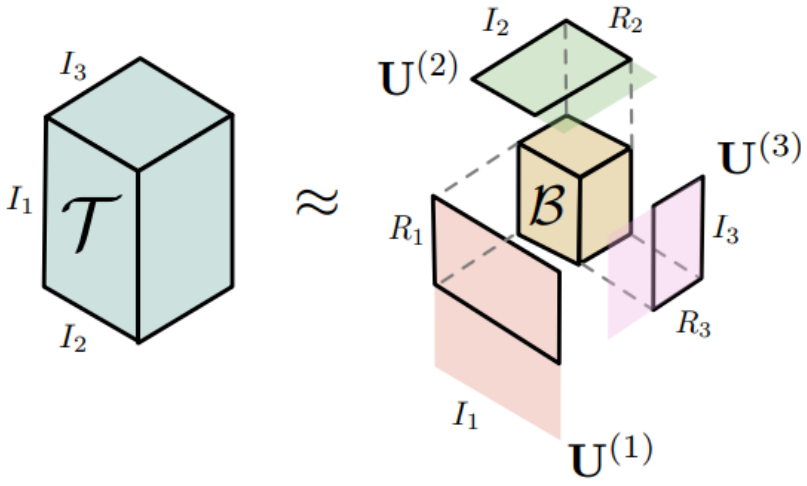


Figure 3.1.: Schema for tensor rank truncation. The original tensor approximated by reducing the size of the core tensor and factor matrices of the HOSVD. Adapted from [BLP19].

3.2. Encoding of Surviving Coefficients by Quantization

The problem of how to efficiently encode any remaining coefficients of the core and factor matrices remains one of the most dominant problems of tensor rank truncation. As mentioned in Section 2.3.3, quantization is a popular method to further compress already transformed data. As discussed in Section 2.2.1, any core produced by the HOSVD can be reordered in such a way that its core slice norms are decreasing in a logarithmic manner. In order to limit the newly introduced quantization error, the mapping scheme is adapted to match the

```
//input: core tensor B, factor matrices U^{1,2,3} and truncation choices R_{1,2,3}
function Truncate(B, U^{1,2,3}, R_{1,2,3})
    B_t = ∅
    for y = 1, ..., R_1 do //truncate core according to given truncation choices
        for x = 1, ..., R_2 do
            for z = 1, ..., R_3 do
                B_t = B_t ∪ B(x,y,z)
            end for
        end for
    end for

    for i = 1, ..., 3 do //truncate factor matrices corresponding to the core
        U^i = U^i.leftCols(R_i)
    end for
```

Figure 3.2.: Pseudocode for truncating a tensor decomposed with HOSVD as described in Section 3.1.

logarithmic probability distribution of the absolute values of the core coefficients [Sut+11]. Every surviving coefficient x of the core and factor matrices is compressed to 9 bits. The first bit is used to save the original sign, while the next 8 bit quantize the absolute value of the coefficient to a value in $[0, 255]$:

$$|x| \to 255 \cdot \log_2(1 + |x|) / \log_2(1 + |x_{max}|). \tag{3.3}$$

, where $|x_{max}|$ refers to the largest coefficient value. Because of its large magnitude and its huge importance for the process of volume reconstruction, the hot corner element $B(1,1,1)$ is excluded from the quantization and stored separately. Figure 3.4 summarizes the examined algorithm steps and gives a general outline on the core truncation compression technique.

```
//input: core tensor B, factor matrices U^{1,2,3} and target error ε_t
function CalculateR(B, U^{1,2,3}, ε_t)
    SAT = BuildSummedAreaTable(B)
    Choices = ∅
    for R_1 = 1, ..., I_1 do //generate F and ε for all truncation combinations
        for R_2 = 1, ..., I_2 do
            for R_3 = 1, ..., I_3 do
                F = (R_1R_2R_3 + I_1R_1 + I_2R_2 + I_3R_3)/(I_1I_2I_3)
                ε = sqrt(||T||^2 − SAT_{R_1R_2R_3}) / ||T||
                Choices = Choices ∪ (F, ε, R_1, R_2, R_3)
            end for
        end for
    end for

    Choices.sort(){Increasing F order}
    Best = ∅
    ε_0 = MAX_VALUE
    for i = 1, ..., Choices.size() do //discard all suboptimal choices
        (F, ε, R_1, R_2, R_3) = Choices.getElement(i)
        if ε < ε_0
            ε_0 = ε
            Best = Best ∪ (F, ε, R_1, R_2, R_3)
        end if
    end for
    r1, r2, r3 = Best.find(ε_t) //get optimal truncation choices
    Truncate(B, U^{1,2,3}, r1, r2, r3)
```

Figure 3.3.: Pseudocode for computing a set C of optimal truncation choices for a tensor decomposed with HOSVD as described in Section 3.1.

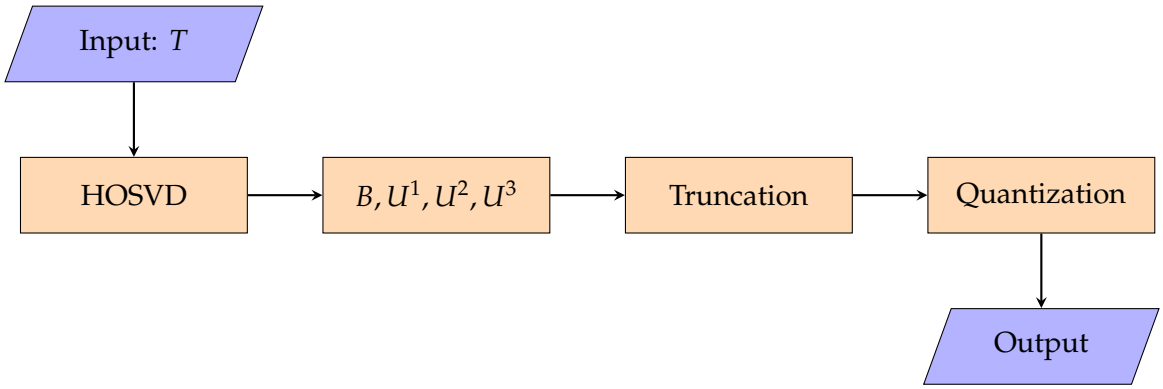


Figure 3.4.: Tensor rank truncation compression algorithm flowchart. The input tensor is decomposed via HOSVD. The resulting core tensor and factor matrices are then truncated and then quantized for encoding. Adapted from [BP16].

4. Implementation of TTHRESH

The paper of Ballester-Ripoll and Pajarola [BP16] pursues the goal of comparing various volume compression approaches with regards to their compression ratio, reconstruction time and magnitude of introduced error. As shown in Section 5.1.2, the authors realize, that, while core truncation techniques achieve lower reconstruction times compared to thresholding algorithms, they fall short in regard of compression ratio and approximation accuracy. Subsequently, Ballester-Ripoll et al. [BLP19] developed *TTHRESH*, a lossy thresholding algorithm, that incorporates low approximation errors with the ability to manipulate compressed data at small costs and dynamic target compression ratios. The algorithm can be split into three main parts:

- First, the three dimensional input tensor T is decomposed into a core tensor B and factor matrices U^1, U^2, U^3 using the HOSVD (see Section 2.2)
- Secondly, the core is converted to a 1D vector, by scaling all C coefficients of the core to 64-bit integers and ordering them in a list. Internally, the list is handled as a $C \times 64$ binary matrix. The matrix is structured in such a way that the binary representations of the coefficients are saved in the rows of the matrix, whereas the n-th column of the matrix (*bit-plane*) holds the n-th significant bit of the coefficient. Thereafter the leftmost most significant columns of the matrix are compressed with RLE followed up by AC (see Sections 2.3.1, 2.3.2). This results in a thresholding of all insignificant coefficients, such that the introduced squared error falls under a given target error.
- Lastly, the factor matrices are compressed in the same manner, using a special importance scaling.

4.1. HOSVD Decomposition

As described in Section 2.2, the HOSVD is used to compute a core tensor B and orthogonal factor matrices U^1, U^2, U^3 by computing the left singular vectors of the unfolded core along each dimension B_i. Since we do not need the right singular vectors for further computations, it is ineffective to calculate each V^{i^T} for each $B_i = U^i \Sigma^i V^{i^T}$. Hence it is far more efficient to evaluate the matrix $\hat{B}_i = B_i \cdot B_i^T$ and acquire the left singular vectors from the eigenvalue-decomposition to form U^i, according to equation 2.2. Additionally, specialized self-adjoint eigenvalue solver can be employed, since $\hat{B}_i$ is defined as a real symmetric matrix, and thus a diagonalization of its eigenvalue decomposition always exists. Because of the orthogonality

of U^i, the right part $\Sigma^i V^{i^T}$ of the SVD can be determined with little computational overhead and is refolded into a tensor, to shape the core B (Fig. 4.1) [BLP19]:

$$\Sigma^i V^{i^T} = (U^i)^{-1} B_i = U^{i^T} B_i \tag{4.1}$$

```
//input: given 3D tensor T
function HOSVD(T)
  B = T
  for i = 1,...,3
      B_i= unfold(B,i) //unfold core into 2D matrix in order to use SVD
      Λ
      B_i = B_i · B_i^T
      U^i = SelfAdjointEigenSolver(Λ B_i) //full eigenvalue decomposition,
      U^i holds the eigenvectors in decreasing order with respect to their
      eigenvalues
      B_i= U^{i^T} · B_i //calculate new core with Σ^i V^{i^T} = U^{i^T} B_i
      B = fold(B_i, i) //fold back 2D core into a tensor for next iteration
  end for
end function
```

Figure 4.1.: Pseudocode for decomposing a tensor with HOSVD as described in Section 4.1.

4.2. Core Encoding

Volume data compression algorithms based on Tucker-decomposition exploit the fact that the core B generated by the HOSVD tends to be very sparse for many real world data sets, and can therefore effectively be approximated by discarding insignificant values. Hence, the core encoding is the part of the compression where most data reduction can be accomplished, but also the part where the primary error for the compressed approximation is introduced. Fortunately, as discussed in Section 2.2.1, any instigated error can be controlled by observing the core-coefficients, since any approximation error in the core transmits directly to the reconstruction of the original tensor. For internal use, a given target relative error ϵ can be converted to a sum of squared errors (SSE) via the following equation:

$$SSE = \epsilon^2 \|T\|^2 \tag{4.2}$$

The decomposition of a tensor T into B and U^1, U^2, U^3 with the help of HOSVD introduces no error to the approximation of the data and from the use of the HOSVD alone, the original tensor could be recomposed without any loss of accuracy. This implies in particular that $\|T\| = \|B\|$ and thus no reduction of the original data size is achieved to this step. By defining

s as the converted target error, the goal of the core encoding now is to reduce the core B in such a way that its approximation $\widetilde{B}$ fulfills

$$SSE(B, \widetilde{B}) \leq s \tag{4.3}$$

Ballester-Ripoll et al. perform data compression with the help of *bit-plane coding*. To this end, the absolute value of each coefficient c of the core is scaled to a 64-bit unsigned integer:

$$c \to \lfloor |c| \cdot 2^{63-\lfloor \log_2(m) \rfloor} \rfloor \tag{4.4}$$

, where $m = \max_{c \in B}(|c|)$ refers to the core's largest element in magnitude. Figuratively, this can be interpreted as scaling each coefficient's absolute value in such a way that the scaled value of the largest core element holds a "1"-bit at the most significant bit position. The signs of the coefficients are stored separately in a dedicated bitmask. The core is then flattened by ordering all scaled coefficient as a sequence and converting them to a binary matrix M for internal use: the 64 bit binary representation $2^{63} \cdot c_{63} + \cdots + 2^0 \cdot c_0$ of each scaled coefficient defines a row of M. The columns of M are further denoted as *bit-planes*, each bit-plane holds bits from the core elements at the same position in the binary representation. Noteworthy, all bits in the same bit-plane are equally important to the overall approximation error, regardless of the coefficient's absolute position in the core. In a greedy encoding strategy, the bit-planes are compressed from the most significant bit-plane $p = 64$ to the least significant $p = 0$, one bit-plane at a time (Fig. 4.2). As the core and as such the binary matrix M is usually sparse (see Section 2.2.1), it is initialized with all values set to 0 at the start of the algorithm. No bits of the core are encoded yet, and the approximation SSE error when reconstructing the compressed data is maximal. Every time a "1"-bit of the core is encoded, information from the core is transmitted to the compression and thus the approximation error is reduced. All processed bit-planes p are traversed from the top to the bottom, and the core encoding stops once the introduced error of the core falls below the target SSE error (see equation 3.5). As a result, only the first $64 \geq p \geq 1$ bit-planes are compressed and in particular all core coefficients with an absolute value $|c| < 2^p$ are discarded and approximated with 0. One can observe an extensive surplus of "0"-bits, as well as continuous "0"-bit runs in the leading bit-planes (see Section 5.1.1), caused by the overall sparsity of the core (except the hot corner phenomenon). Ballester-Ripoll et al. efficiently use this structure to their advantage by deploying *modified run-length encoding* (see Section 2.3.1) to compress the traversed bit-plane. Lastly, the RLE-vectors are encoded again with the help of arithmetic coding (see Section 2.3.2).
Inspection of the bit-planes suggests a classification of each coefficient's bits into two different groups, according to their value distribution:

- *leading bits* describe the coefficients leftmost "1"-bit in addition to all leading left "0"-bits. When traversed along the columns of the matrix M starting from the most significant bit-planes, they tend to form long runs of "0"-bits that exhibit low entropy. Accordingly, RLE+AC can compress these bits effectively, without introducing additional approximation errors.

- *trailing bits* refer to all remaining bits that appear to the right of the leftmost "1"-bit. Contrary to the leading bits, they appear to randomly exhibit "0"- or "1"-bits. Hence, RLE+AC is inept to compress these bits efficiently and they have to be stored verbatim.

In order to apply both encoding techniques on the same bit-plane, some way to differentiate between leading and trailing bits is needed. To this end, a *significance map* in the form of a binary mask is utilized. When traversing the bit-planes, the mask records and marks all coefficients whose leftmost "1"-bit have already been encountered. Before endcoding a specific bit, the bitmask is then checked and used to decide on RLE+AC (bit is flagged with a "0"-bit in mask) or verbatim encoding (bit is flagged with a "1"-bit in mask).

4.3. Factor Matrices Encoding

The factor matrices U^1, U^2, U^3 are essentially encoded in the same way as the core (see Section 3.2.2). Although the factor matrices incorporate far fewer elements than the core, each factor matrix is just as influential to the overall approximation error. It follows that it is thus reasonable to spend more bits for the encoding of factor coefficients and the stopping criterion for the bit-plane coding has to be adapted accordingly and should not halt at the same position as determined for the core. In order to achieve a compression quality for the factor matrices similar to the overall core approximation quality, the stopping criterion should not be based solely on a target error, but on the compression ratio and accuracy established during the core compression. One can observe that later bit-planes are not as cheap to encode and do not decrease the introduced error as greatly as the first ones, since the average entropy increases and the significance of the bit-plane is lowered. This behaviour is imitated by the ratio $\alpha_b = \triangle s_b / \triangle S_b = (s_b - s_{b-1})/(S_b - S_{b-1})$ between compression quality (compression SSE s_b) and storage cost (filesize S_b) after receiving b core bits. Since the first traversed bit-planes are cheap to encode the ratio is largest for the first encountered bits. α_b is estimated for the core and used as a stopping criterion in order to put the weightings of the core coefficients and the factor matrix coefficients for the overall compression quality in relation. Encoding of each i-th factor matrices is stopped after b^i bits, once $\alpha_{b^i} \leq \alpha_b$. As the significance of each coefficient and thus the significance of each bit-plane in the factor matrices is higher than in the core, this method results in the compression of more bit-planes in the factor matrices than selected for the core.
Additionally, it is important to note that in the reconstruction of the HOSVD, each column of the factor matrices U^1, U^2, U^3 interacts with one core slice of B. Since the core slices carry vastly different norms and significances (see Section 2.2.1), the matrix columns have different importances for the compression accuracy, and need proper weighting before undergoing bit-plane coding. Along each dimension i, appropriate weighting factors can easily be computed by realizing that the diagonal matrix Σ^i of the SVD $B_i = U^i \Sigma^i {V^i}^T$ holds the core slice norms $\epsilon^i_1 \dots \epsilon^i_{I_n}$. In order to monitor any error introduced by the i-th factor matrix, each j-th column of U^i has to be scaled by its corresponding core slice norm σ^i_j. For later reconstruction of the original tensor, the norms have to be specifically saved.
A summarization of the TTHRESH algorithm can be found in figure 4.3.

```
//input: vector c of C coefficients from the core or factor matrices,
target approximation error s, boolean isCore
function BitMaskEncdoding(c,s,isCore)
    s̃ = |c|² //approximation error is set to maximum
    BM = ∅ //binary mask to record coefficients, whose leftmost "1"-bit has
    already been encountered
    M = scale(c) // C×64 binary matrix, holding all scaled values of c
    for p = 64,...,1 //traversing most significant bit-planes to less significant
        for co = 1,...,C //iterating over coefficients
            if BM[co] == 0 then
                encodeBitRLE(M[co, p])
                if M[co, p] == 1 then //mark leftmost "1"-bit encountered
                    BM = BM ∪ {co}
                end if
            else //leftmost bit of coefficients has already been encountered
                encodeBitVerbatim(M[co, p])
            end if
            update current SSE s̃
            if isCore and s̃ < s then //target compression quality reached
                exit nested iterations
            end if
            calculate current proportion α̃: error reduction performed
            by encoding the last co bits versus the storage cost
            to encode them
            if !isCore and α̃ < α_b then //stopping criterion for factor matrices
                exit nested iterations
            end if
        end for
    end for
    if isCore then
        return α̃ as α_b //save stopping criterion for
        subsequent factor matrix encoding
    end if
end function
```

Figure 4.2.: Pseudocode for compressing core and factor matrices of the HOSVD with bit-plane coding as described in Sections 4.2 and 4.3.

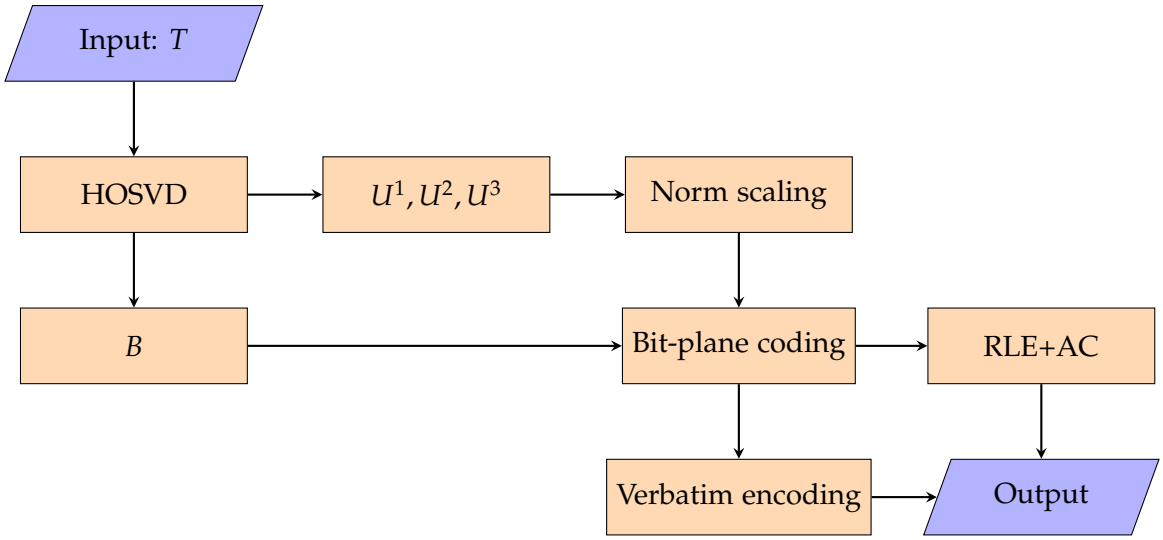
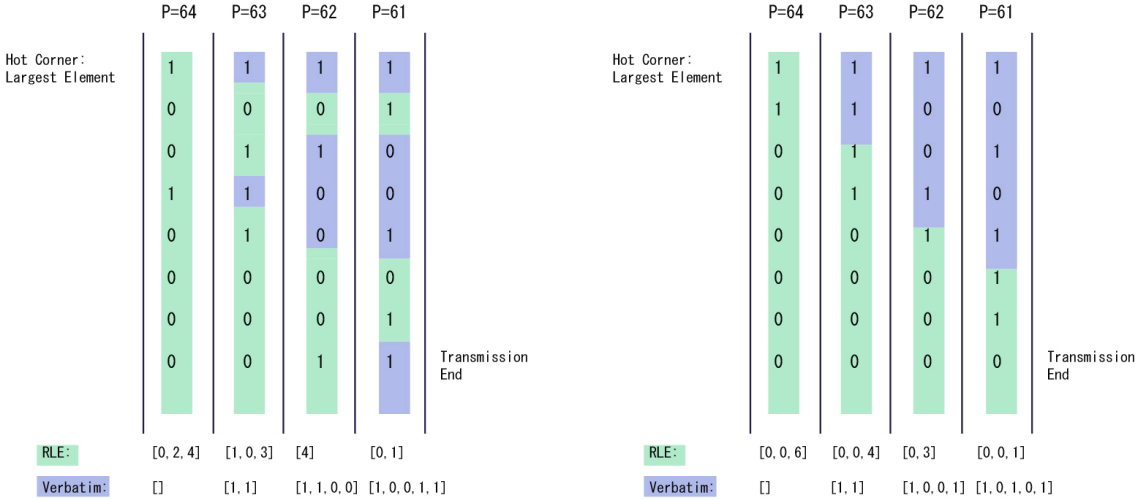


Figure 4.3.: Flowchart depicting the TTHRESH algorithm. The input tensor is decomposed with HOSVD. Both core and factor matrices are encoded with bit-plane coding, but the factor matrices are prior scaled with the core slice norms to apply proper weighting. The bit-planes of the volume elements are either encoded verbatim or with the help of RLE followed up by AC.

5. Results

The main objective of this thesis is to analyze and improve the TTHRESH volume compression algorithm by Ballester-Ripoll et al. in regards of their compression quality and compression ratio [BLP19]. Specifically, the following hypotheses are investigated:
Firstly, the authors of TTHRESH discuss various properties of the HOSVD, but especially the hot corner phenomenon seems to be not utilized to its full extend yet. As discussed in Section 2.2.1, the core calculated by the HOSVD is usually sparse, but can be reordered in such a way, that its core slices are decreasing in norm along the three dimensions. The result of this ordering is the hot corner phenomenon, where all significant coefficients that contain most of the core's energy concentrate around the element $B(1,1,1)$. After ordering of the HOSVD core and factor matrices, the volume data is then encoded with bit-plane coding, which incorporates RLE followed up by AC (see Section 4.2).
In case of AC, the final number of bits needed in the output message will converge to match the number of bits dictated by the entropy of the input, given a sufficiently long input sequence (see Section 2.3.2). This means that it is crucial to generate input messages with a low entropy (vectors that contain mostly the same symbols) in the RLE step of the algorithm in order to obtain good compression ratios with AC. Symbols that appear only a few times in the input sequence can't be encoded efficiently by corresponding codewords and thereby decrease the compression ratio. In the original TTHRESH compression algorithm, the bit-plane encoding traverses the core coefficients as the volume data is stored in memory (*memory ordering*). This thesis assumes that volume data is represented in memory as a cluster of consecutive mode-1/ y-dimension fibres (see Fig. 2.3 a), starting from the top of the leftmost fibre of the foremost mode-1 slice and stretching to the bottom of the rightmost fibre of the hindmost mode-1 slice. By reordering the core coefficients depending to their Manhattan distance to the hot corner, the core coefficients with the highest magnitude will be traversed first for every bit plane. Hence, the resulting RLE-vector will theoretically contain an accumulation of small numbers, since the most significant coefficients with a high probability for an unrecorded leftmost "1"-bit are grouped together (Fig. 5.1). As a consequence, the general entropy of the RLE-encoding decreases. Thus, simply changing the order of the traversed core coefficients in an advantageous way with respect to the hot corner can increase the overall achieved compression rate without negatively affecting the approximation accuracy.

Secondly, linking the thresholding compression approach of TTHRESH with already well-established compression techniques, such as core truncation, could potentially increase the obtained compression ratio. The idea is inspired by Ballester-Ripoll et al. [BP16], who compare thresholding and truncation compression techniques and suggest an hybrid compression algorithm that combines the advantages of both approaches for further research. In the case



(a) (b)

Figure 5.1.: Example of encoding of the core bit planes with RLE (green) and verbatim encoding (blue) until a certain threshold after plane 61 is reached with memory ordering (a) and traversal by Manhattan distance (b). Each bit plane is traversed from the top to the bottom, but the ordering of the core coefficients in the bit planes differs between the two examples. In the process, the significance map was updated from 0 to 7 members. One can observe that the example with the Manhattan ordering yields more RLE-symbols that have the same value, thus carrying a smaller entropy than the example with memory ordering.

of TTHRESH, it stands out that all coefficients with an absolute value $|c| < 2^P$ are thresholded away when stopping at bit plane P. Because of the sparsity of the core, as well as the slice wise structure of the HOSVD components, this means that many core slices and therefore also many scaled columns in the factor matrices hold only 0-values after bit-plane coding. These factor columns and core slices carry no useful information for the reconstruction of the original tensor from the HOSVD. Hence, they can be truncated without any loss of compression quality. Additionally, it is interesting to investigate, if the acquired compression ratio can be further enhanced by cutting away additional core slices, without affecting the thereby introduced approximation error in a significant way.

5.1. Findings

In order to verify or disprove these hypotheses, numerous tests with various volume data sets were conducted. The data sets were acquired from the Visualization Group TU Wien [Fac20]. Compression ratios as well as compression quality for different coefficient orderings and

Table 5.1.: Comparison of the achieved compressed filesize of the (277x277x164) Stagbeetle data set with different approaches to traverse the volume data with the TTHRESH algorithm. "Original size" refers to the original filesize of the uncompressed volume data. "Optimal" marks the filesize of the compressed volume data when ordering the coefficients beforehand and traversing them in decreasing magnitude. "Random" randomly permutes the core elements and functions as a base case to measure possible benefits from advantageous reordering. "in-memory" does not alter the traversal order of the coefficients in any way and the coefficients are iterated in the sequence as the volume data is saved in-memory. "Manhattan distance" traverses them according to their position in the core and distance to the hot corner.

	Original size	Optimal	Random	In-memory	Manhattan distance
Filesize (KB)	24.578	6442	12.890	11.087	11.396

core truncation choices were recorded and analyzed. Special attention was given to the frequency models of AC and the structure of the bit planes, in order to discover advantageous connections between the traversal order of core coefficients, RLE and AC. The above mentioned tests were also performed on a perfectly sorted, as well as randomly sorted core, in order to serve as a reference in order to measure the influence of the core coefficient ordering on the compression ratio of TTHRESH. In order to design an improved hybrid compression algorithm, compression quality, as well as reconstruction speed and achieved filesizes of the TTHRESH and core truncation algorithm are benchmarked.

5.1.1. Taking Advantage of the Core Traversal Order

Table 5.1 suggests that increasing the compression ratio of the TTHRESH algorithm by adjusting the traversal-order of the core coefficients is a reasonable idea. On its own, the compression algorithm reduces the original data's filesize from 24.578KB to 11.087KB, thereby achieving an compression ratio of over 50%. A visual comparison of the original data and the compressed data can be found in Figure A.3. On the other hand, ordering the core coefficients according to their magnitude in decreasing order before passing them to the bit-plane encoding step further enhances the compression ratio to 42% and reduces the needed filesize to 6442KB, without affecting the compression quality in any way. In order to measure the influence of the iterating-order on the AC frequency model and the compression ratio of the algorithm, a base test case with a randomly permuted core is constructed. The base test case features a worse compression ratio than all other ordering strategies, and thereby indicates a connection between compression ratio and core traversal order. As demonstrated by Table 5.2 and as mentioned above, this significant increase in compression ratio can be traced back to the improved frequency model of AC. The magnitude ordering guarantees that a single run of "1"-bits is followed by a run of "0"-bits at the beginning of every bit plane, to sum up all insignificant coefficients without a "1"-bit on that plane. Thus, the overall number of symbols

to encode in the RLE-vector decreases, while the frequency of small numbers (e.g. 0 from the runs of "1"-bits) in the model rises. As such, the entropy of the RLE-vector declines and AC can encode the bit planes very efficiently. In theory, a similar effect could be achieved by traversing the core coefficients with regard to their position in the core, since the coefficient's distance to the hot corner gives a rough estimate about its magnitude. However, a closer look of Table 5.1 gives a different impression: Traversing the core coefficients with respect to their Manhattan-distance to the core does not improve the obtained compression ratio. On the contrary, the required filesize rises from 11.087KB to 11.396KB, an increase of 2.8%. An analysis of Table 5.2 and Figure 5.2 suggest that this difference in compression ratio is based on the different frequency models. Although the Manhattan ordering results in grouping of the core's most significant elements and traverses over elements of higher magnitude than the memory ordering at the start of the iteration sequence (Fig. 5.4), the latter contains more zeroes and ones in its frequency model (Tab. 5.2). A broader look on the frequency models shows that in both models, the smallest RLE-symbols obtain the highest frequencies, while the frequency of longer RLE-runs decreases (Fig. 5.2). Both models converge in a similar fashion, although the Manhattan distance ordering contains a spike around RLE symbol 65, while the memory ordering holds an unusual spike at RLE symbol 276, which will be a key in understanding the difference in compression ratios between both traversing techniques.
Using different weightings of the axis in conjunction with Manhattan ordering does not improve this condition, either. The only traversal refinement that yields an improvement of compression ratio is the alteration of the in-memory interpretation of the volume data, without the use of Manhattan distance ordering. Traversing the volume according to the x-,y- or z-dimension yields different compression ratios (Table 5.3), but the benefit of each traversal option seems to be random and cannot be connect to attributes of the data set (like size of dimensions, or magnitude of core-slice norms in order to find significant or insignificant dimensions) and is thus hard to be taken advantage of, without precomputing the compression results of the different orderings first. Straightforward optimal ordering according to the magnitude of the core elements is also not a feasible option. Unlike with the Manhattan ordering, a sorting of the core coefficients makes the intrinsic reconstruction of the element's original positions impossible, and the permutation order of the elements has to be saved as an additional parameter, in order to restore the original tensor. The surplus of this additional data exceeds the gain in compression ratio of the optimal ordering by far.

From these experiments it can be concluded that it is better to leave the traversal order of the core coefficients untouched and use simple memory ordering in favor of Manhattan distance ordering. The cause of this behaviour can be found in the slice-wise structure of the core. When decompositing a tensor with the help of HOSVD, the resulting core is comprised of slices, whose elements are quite homogeneous in the same slice and bolster a higher absolute magnitude than other slice's elements (see Section 2.2.1). Traversing the core elements as they are stored in memory takes advantage of this structure. For example, the $(277 \times 277 \times 164)$ core of the Stagbeetle test volume set can be ordered in such a way that the norms of its slices are decreasing along the three axes, with the hot corner on $B(1,1,1)$. Using

Table 5.2.: Comparison of the AC frequency models when compressing the(277x277x164) Stagbeetle data set by traversing the volume according to the coefficient's absolute magnitude in optimal decreasing order (a), according to memory ordering (b), or according to Manhattan distances to the hot corner (c). Every input symbol of the RLE-vector (left column) is mapped to its overall frequency in the whole input sequence (right column). For the optimal ordering (a) the sequence of core coefficients guarantees that all "1"- and "0"- value bits are grouped in two consecutive runs for every bit plane, with the run of "1"-bits at the beginning of the plane, followed up by the "0"-bits of the remaining coefficients that are not significant on that plane. Since the modified RLE only considers "0"-bit runs, the run of consecutive "1"-bits at the start of the bit plane results in an accumulation of RLE-symbol 0 for the AC frequency model, while the run of "0"-bits is summed up by the larger RLE-symbols. (b) and (c) cannot guarantee a perfect splitting of their bit planes in two runs of "1"- and "0"-bits and therefore produce broader frequency models with a higher entropy and a worse compression ratio.

RLE symbol	Frequency	RLE symbol	Frequency
0	6291779	12564413	1
6291778	5	12578957	1
6725673	1	12582638	1
7696434	1	12583380	1
8817590	1	12583524	1
9991453	1	12583548	1
11065824	1	12583553	1
11883963	1	12583554	1
12336345	1	12583555	1
12510451	1		

(a)

RLE symbol	Frequency
0	3046294
1	1726143
2	1046085
3	667650
4	419041
5	287737
6	195242
7	139162
8	102364
9	78422
10	59546

(b)

RLE symbol	Frequency
0	2648321
1	1666766
2	1064871
3	663983
4	450738
5	326252
6	229180
7	165130
8	129119
9	98220
10	76716

(c)

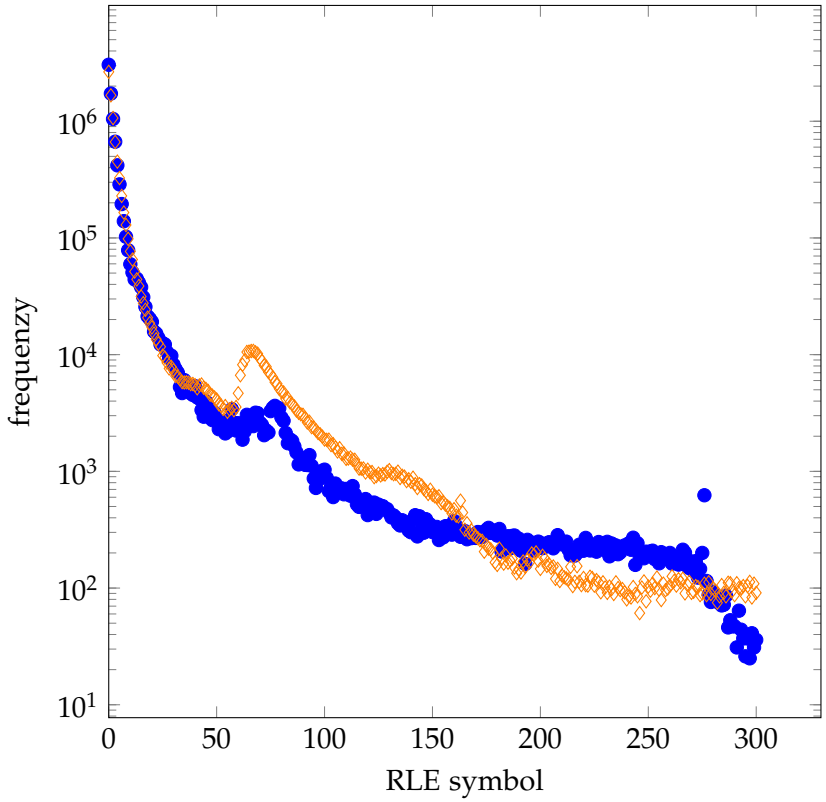


Figure 5.2.: Comparison of a wider view of the AC frequency models for in-memory (blue) and Manhattan distance (orange) traversal of the ($277 \times 277 \times 164$) Stagbeetle data set. The x-axis represents the unique RLE-symbols to be encoded, while the y-axis maps each symbol to its corresponding frequency over the whole RLE-vector. For both models, the frequencies of the large symbols following after symbol "300" converge in the same manner and are cut off to enable a more focused view.

Table 5.3.: Comparison of compression results when approximating the ($277 \times 277 \times 164$) Stagbeetle, ($246 \times 246 \times 221$) Present and ($256 \times 249 \times 256$) Christmas Tree volume data sets with TTHRESH with different orderings along the three dimensions. In case of the y,x,z ordering, all mode-1 fibres of the tensor slices (Fig. 2.3) are traversed from top to bottom, beginning with the leftmost fibre of the slice and iterating the mode 1 slices from the front to the back. As a stopping criterion a target error of $\epsilon = 0,0003$ is given for each compression, which translates to an approximation error of 0,03%. For each depicted test data set connections between compression ratio (a) and maximum slice norm along each dimension (b) were examined. The Stagbeetle and Present data set gain the best compression ratio (marked in bold letters) when traversing their cores according to the magnitude-ordering of their maximal slice norms. Extensive testing with numerous data sets breaks this trend. In this example, the Christmas Tree test set does not provide the best compression ratio when iterating the core according to the dimension with the highest maximal slice norm.

	Stagbeetle	Present	Christmas Tree
Ordering	Filesize (KB)	Filesize (KB)	Filesize (KB)
y,x,z	11.087	16.015	**20.665**
y,z,x	11.085	16.015	20.666
x,y,z	11.078	16.017	20.674
x,z,y	**11.071**	16.015	20.675
z,y,x	11.168	15.958	20.716
z,x,y	11.167	**15.957**	20.715

(a)

	Stagbeetle	Present	Christmas Tree
Dimension	Max norm	Max norm	Max norm
y-dimension	351289	544339	183952
x-dimension	434406	555657	230224
z-dimension	355502	625926	180539

(b)

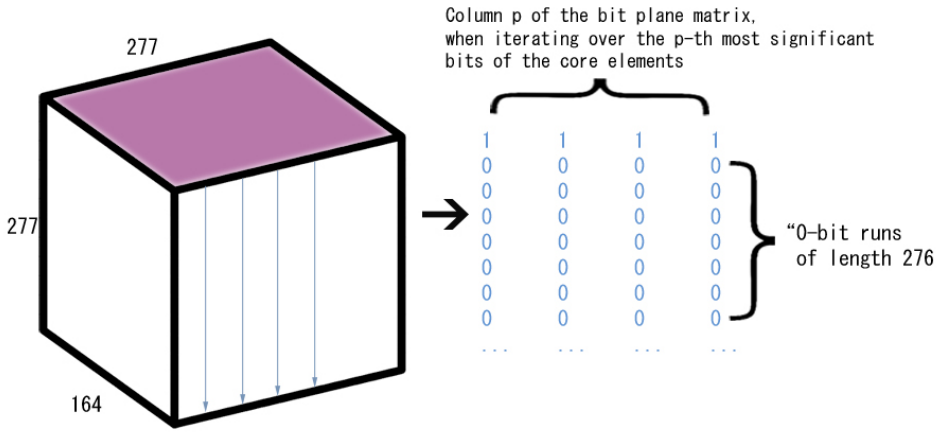


Figure 5.3.: Example for traversing the (277, 277, 164) Stagbeetle data set with memory ordering. The core is structured in such a way that the uppermost slice (violet) holds the core coefficients with the highest magnitude. When iterating column wise in the bit-plane encoding (blue arrows) the RLE may increasingly encounter "0"-bit runs of size 276 (blue number-strings) from the underlying slices, before starting again with the next element from the uppermost slice.

memory ordering (e.g. each mode-1 / y-dimension slice is saved column-wise in-memory), the core elements are iterated perpendicular to a specific dimension (e.g. the y-dimension) (Fig. 5.3). As a consequence, the RLE step will encounter many "0"-bit runs of length similar to the length of that dimension or its multiples. The runs can be efficiently represented as a single symbol by the RLE, while the overall frequency of small RLE-symbols increases, thus developing a low entropy and effective encoding with AC. This effect can also be seen in figure 5.2, as the spike at RLE-symbol 276 of the in-memory graph works against the general downtrend of the frequencies and stems from the described increased probability of "0"-bit runs corresponding to the length of the y-dimension. On the other hand, the Manhattan distance of a core coefficient to the hot corner only gives a rough estimate about the coefficients absolute magnitude, and the core elements won’t be ordered in strictly decreasing magnitude order, like in the optimal ordering. Thus, when using Manhattan distance ordering, the magnitudes of the coefficients will spike seemingly at random and the lengths won’t be structured as nicely, which results in a larger entropy. Figure 5.4 illustrates this uneven magnitude distribution: In the in-memory graph, one can see that the absolute values of the

core elements are overall quite similar. In the graph depicting the Manhattan ordering, huge spikes in core significance of the coefficients can be registered. When iterating over the bit planes of the core, the RLE will thus encounter more uniform runs in the memory ordering than in the Manhattan ordering, which results in a smooth AC frequency model with a lower entropy and a better compression ratio.

Although adjusting the traversal order of the core according to the Manhattan distance of the core elements to the hot corner did not bring the expected gain in compression ratio (Tab. 5.1), the findings of the AC frequency model (Fig. 5.2) are interesting and motivate an alternative approach to utilize the hot corner phenomenon. One can notice that the above described layout of the in-memory ordered tensor data results in spikes in the AC frequency model at the size of the y-dimension, or multiples of it. This observation inspires the use of factorization, in order to decrease the overall entropy of the encoded RLE-vector and thus increase the compression ratio. RLE-symbols $z = x \cdot y$ are factorized and saved into two symbols x and y, where one of the symbols is the mentioned size of the y-dimension, or multiples of it. This way, larger RLE-symbols that only appear a few times in the AC frequency model are removed and replaced with two smaller symbols that appear more often. Because the frequency of these smaller symbols increases as a result of the factorization, it is more effective to encode two symbols in favor of one. Figure 5.5 gives an overview over the needed bits to encode each RLE-symbol via AC. Stemming from the spikes in the AC frequency model, the y-dimension size and and its multiples are able to be encoded more efficiently with less bits, than comparable RLE-symbols in the same magnitude ranges. Hence, the motivation for the use of factorization int the compression pipeline seems to be sound. For example, encoding the symbol "1380" would originally require 22 bits. Clever factorization into "5" (representable in 5 bits) and "276" (representable in 14 bits) would decrease the required bits to 19 bits.
The factorization choices are generated from the size of the y-dimension and all multiples of it, which exist in the AC frequency model. Before encoding the RLE-vectors with AC, every RLE-symbol is checked for good factorization choices, and the choice with the most favorable bit-count is chosen. Furthermore, the factorization is marked with a special symbol in order to reconstruct the original RLE-symbol in the decoding step. It is important to note that as such not all symbols or even all symbols with a low AC frequency are factorized. The factorization of a symbol z into x and y is only beneficial for the AC step, if x and y are symbols of already high frequencies. In essence, this means that only those RLE-symbols are factorized, which have the size of the y-dimension, or one of its multiples, as a possible divider. The amount of RLE-symbols that can thus be efficiently represented by factorization is marginal in comparison to the overall number of encoded RLE-symbols. In case of the ($277 \times 277 \times 164$) Stagbeetle test set, only 1% of all 800000 symbols are able to be factorized in an advantageous way. Since the bit-gain often just influences the decimal range and thus the actual number of bits needed for encoding is not changing, only 5100 bits are saved by factorization of the Stagbeetle test set. Compared to the overall TTHRESH-compressed filesize of 11.087KB of the test set, the factorization of the RLE-symbols has in consequence next to

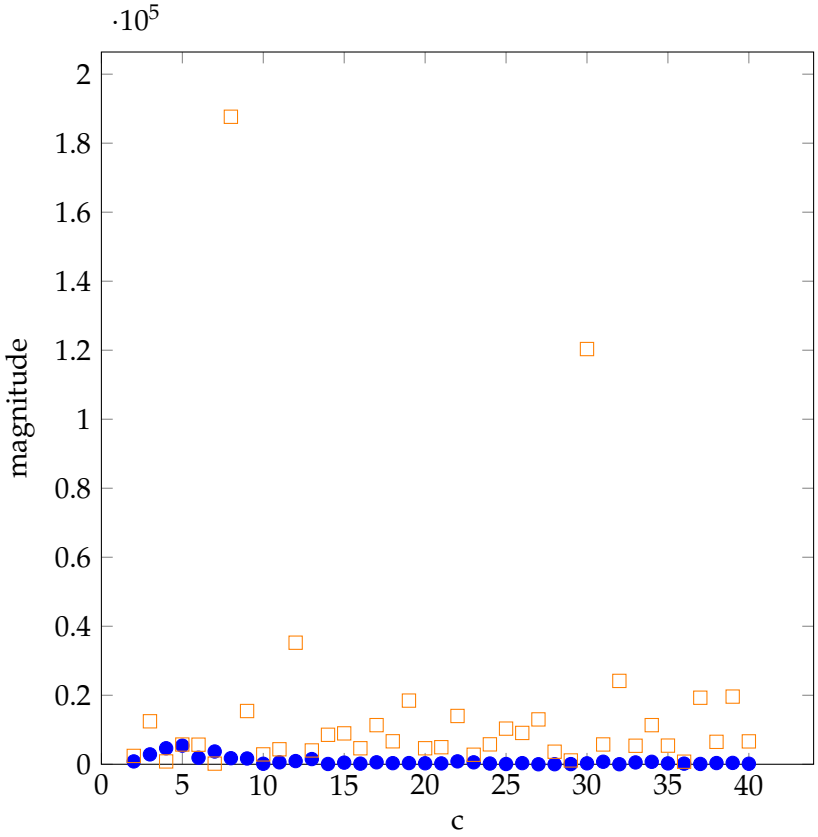


Figure 5.4.: Comparison of the first 40 iterated core elements when using ordering by memory (blue) or Manhattan distance (orange). The first element in both cases is the hot corner element $B(1,1,1)$, which has been cut from this representation. The x-axis represents the core elements c, while the y-axis maps each element to its absolute magnitude. One can see that the Manhattan distance ordering holds overall larger elements than memory ordering in the first 40 entries and includes big spikes in core element magnitude.

no influence on the overall compression ratio.

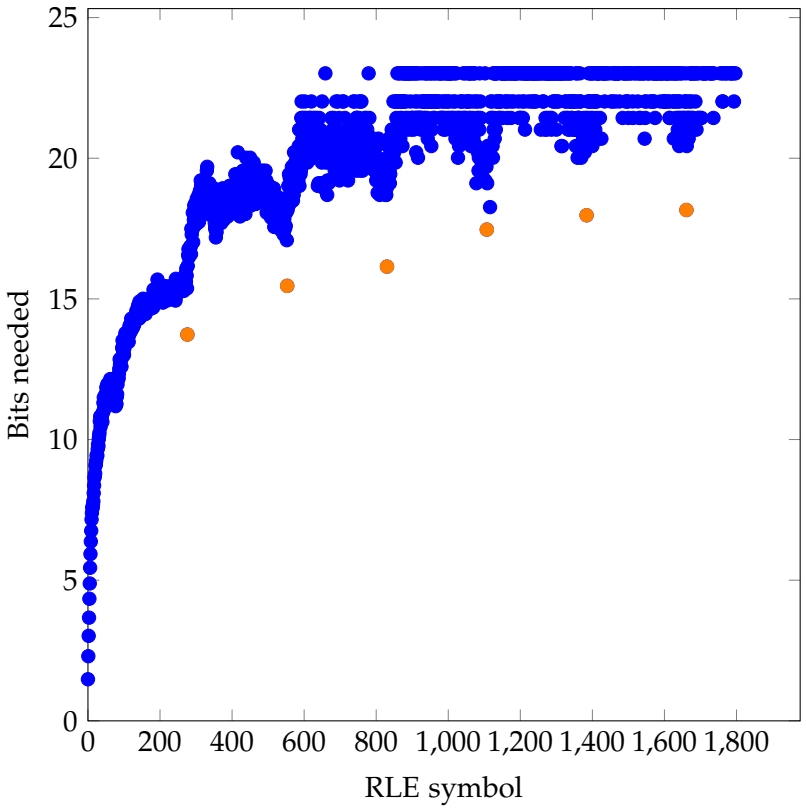


Figure 5.5.: Overview of the needed bits (y-axis) to encode each of the first 1800 RLE-symbols (x-axis) of the ($277 \times 277 \times 164$) Stagbeetle test set. As the RLE-symbols are encoded with AC, the needed bits for a symbol A are calculated according to the information value of the entropy: $\text{Bit}(A) = -\log_2(p(A))$, where $p(A)$ refers to the probability of A occurring in the RLE-sequence. The orange dots refer to multiples of the y-dimension (277).

Table 5.4.: Bench-marking of encoding and reconstruction speed in ms for the TTHRESH, core truncation and TTHRESH-Truncation hybrid algorithm when compressing and decompressing the ($277 \times 277 \times 164$) Stagbeetle test set. The average values of 6 tests conducted on different dates and times of the day are depicted below.

	TTHRESH	Truncation	Hybrid
Encoding (ms)	10.240	7.643	10.255
Decoding (ms)	8.639	4.906	8.414

Table 5.5.: Comparison of core truncation and TTHRESH in regards of their compression ratio and quality. All tests were conducted on the (277,277,164) Stagbeetle test set. Row "Original" refers to the original volume data, without any compression, while "TTHRESH" and "Truncation" point to the compression results when approximating the original data with said techniques. The "Control sequence" is a sequence of volume values read out after compression of the volume to examine the real approximation error. The sequence is scaled to integers for a better readability and does not reflect the exact values after reconstruction of the compressed tensor. Although TTHRESH and Truncation arrive at similar compression ratios, TTHRESH obtains a better approximation of the original data.

Compression Method	Compressed filesize	Control sequence	Approximation error
Original	24578KB	(284,1066,1530,754)	0%
TTHRESH	11.087KB	(283,1063,1526,753)	0,03%
Truncation	10.730KB	(257,978,1438,698)	8%

5.1.2. TTHRESH Truncation Hybrid

Comparing compression of volume data by core truncation (Section 3) against core thresholding (Section 4) yields the same results as observed in [BP16]: While core truncation boasts better encoding and decoding speed (Table 5.4), thresholding achieves better compression quality than truncation for similar compression ratios (Table 5.5). This thesis values compression quality over reconstruction speed, thus TTHRESH is better suited as a basis for further improvement of the achieved compression ratio with a TTHRESH-Truncation hybrid compression algorithm. TTHRESH combines the use of HOSVD with RLE and AC in the bit-plane coding step of the algorithm, in order to achieve a smooth approximation of the volume data (see Fig. 4.3). After the bit-plane coding, any coefficients with an absolute value $|c| < 2^P$ are reduced to 0 when stopping at bit plane P. By precomputing the stopping plane P, the TTHRESH-Truncation hybrid can determine any core slices and scaled factor matrix columns that would be affected by this thresholding. If all elements in a thresholded core slice or scaled factor matrix are reduced to 0, the whole slice or corresponding factor matrix bears

no useful information for the reconstruction of the original volume data. By combining core truncation with the already established TTHRESH, these core and factor elements (further referred to as *zero factor columns and core slices*) can be cut off from the encoding process, and are just intrinsically assumed with value 0 for the reconstruction of the tensor. Thereby the number of reconstruction steps and overall amount of volume data to be saved is reduced. This data reduction results in slightly higher reconstruction speed (Table 5.4) and theoretically in an improved compression ratio. It is also compelling to investigate the option to truncate additional insignificant core elements beyond the zero factor columns and core slices, without severe worsening of the overall approximation quality. Apart from the clipping of said factor columns and core slices, the hybrid algorithm continues with bit-plane coding as described in Section 4.2. As a reference, Figure A.2 depicts a flowchart of the TTHRESH-Truncation hybrid algorithm.

Tests with the hybrid compression technique reveal a marginal increase in compression ratio compared to the original TTHRESH compression algorithm (Table 5.6). Although considerable parts of the core were cut off (about 10% in case of the ($277 \times 277 \times 164$) Stagbeetle test set), the advantage in compressed filesize falls short of expectations and only an improvement of about 1% from cutting off the zero factor columns and core slices can be realized. Moreover, the gain in compression ratio from truncation declines drastically with broader approximations of the original volume data and decreasing overall compressed filesize (Fig. 5.7a, Tab. 5.6).
The truncation of additional significant core data performs in a similar way. When answering the question on how much of the overall hybrid compression error should be contributed by the TTHRESH target error and the truncation error, one has to consider a criterion that measures the influence of the TTHRESH- and truncation error on the compressed filesize and approximation quality (Fig. 5.6). The ratio α plots the overall compression error versus the compressed filesize and is calculated for different TTHRESH- and truncation error pairs. A high α indicates a high compression ratio achieved by a small introduced error. Even though the TTHRESH- truncation error pairs peak in a similar way, no intersection points can be found. This means that a broader approximation in the TTHRESH-step has a higher influence on the compression ratio with a smaller overall approximation error than the truncation of additional core slices. Although the additional truncation of core elements results in further improvement of compression ratio, the thereby introduced error to the overall approximation quality scales faster than with using a comparable TTHRESH target error. This uneven scaling makes it impossible to achieve a better hybrid compression ratio by shifting to a lower TTHRESH target error and a higher truncation error. Thus, the cutting of additional significant core slices at the cost of an additional truncation error is suboptimal compared to compression with a higher TTHRESH target error, since with broader TTHRESH a similar compression ratio with a better compression quality can be achieved.
However, it is possible to use a list of best truncation choices, similar as mentioned in Section 3.1, to further reduce the core in such a way that the truncation error won't be noticeable in comparison to the overall target error ϵ given by the TTHRESH-base. Figure 5.7 illustrates

Table 5.6.: Comparison of obtained filesizes when compressing the (277,277,164) Stagbeetle data set with TTHRESH or TTHRESH-Truncation hybrid, using different approximation errors to achieve different compression ratios. If the test set is compressed with a larger allowed approximation error, the accomplished compression ratio rises in both cases. Although the hybrid method performs slightly better (about 1%) the advantage of using the TTHRESH-Truncation hybrid declines, as the filesize after compression decreases. In case of the TTHRESH-Truncation hybrid, 12, 55 and 4 slices along the y-, x- and z-dimension of the core and factor matrices were identified as zero factor columns and core slices (amounting to about 10% of all core slices) and were cut off respectively.

Approximation error	filesize TTHRESH	filesize Hybrid	Hybrid advantage
0%	20.050KB	19.885KB	165KB
0,03%	11.087KB	11.005KB	82KB
0,3%	7.050KB	7.013KB	37KB
3%	3.230KB	3.222KB	8KB

this connection between TTHRESH target error and additional hybrid truncation error. One can notice that a more generous TTHRESH target error enables the use of a broader hybrid truncation error, without causing major differences in compression quality (Fig. 5.7b). Figure 5.7a depicts the gain in filesize of the hybrid algorithm when truncating further core slices in addition to the zero factor columns and core slices. Even though the hybrid's advantage in filesize decreases with a broader volume approximation by the TTHRESH-algorithm, the previous observation enables the use of a more generous truncation of the core in case of a higher TTHRESH approximation error, thereby countering this downwards trend. Figure 5.6 suggests that the broad truncation of additional significant core slices in inefficient compared to the use of a higher TTHRESH target error. Nevertheless, a small enough truncation error corresponding to the TTHRESH target error can be contained in such a way that no significant drop in compression quality is introduced, while additional core slices are truncated. Therefore, this method is suggested as a pure improvement over simply truncating the zero factor columns and core slices. Figure A.4 depicts the rendering results of the TTHRESH-Truncation hybrid algorithm when applying different additional truncation errors. As described above, a larger truncation error results in visible rendering artifacts that worsen the overall compression quality, while a controlled and small enough truncation error is not causing any drop in approximation value.

The above mentioned overall poor improvement of the TTHRESH-Truncation hybrid can be traced back to the bit-plane coding of the TTHRESH algorithm. In TTHRESH, core and factor elements are not encoded as singular elements (like in quantization), but are merged as RLE-runs in the bit-plane coding. Truncating the core and thereby reducing the number of core elements does thus not decrease the overall number of encoding symbols of the

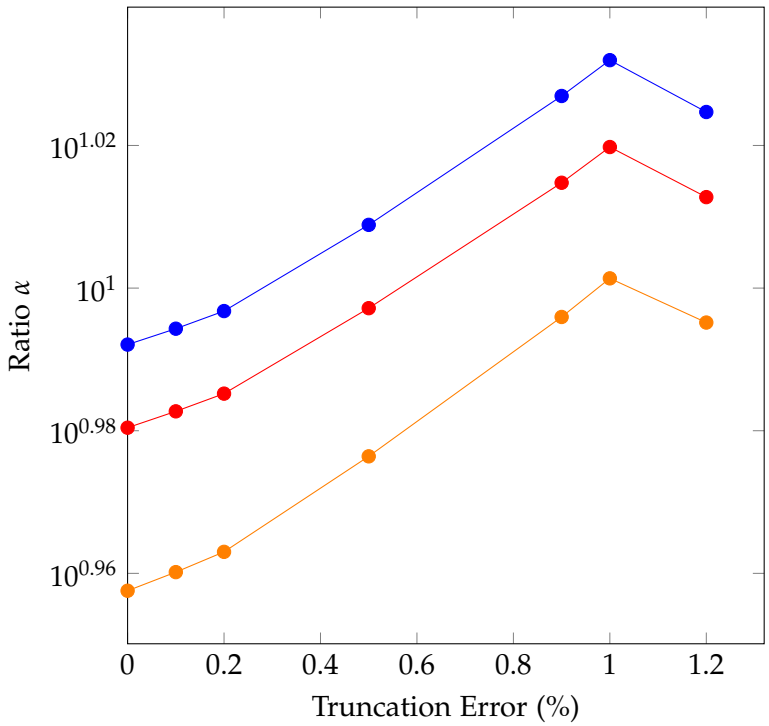
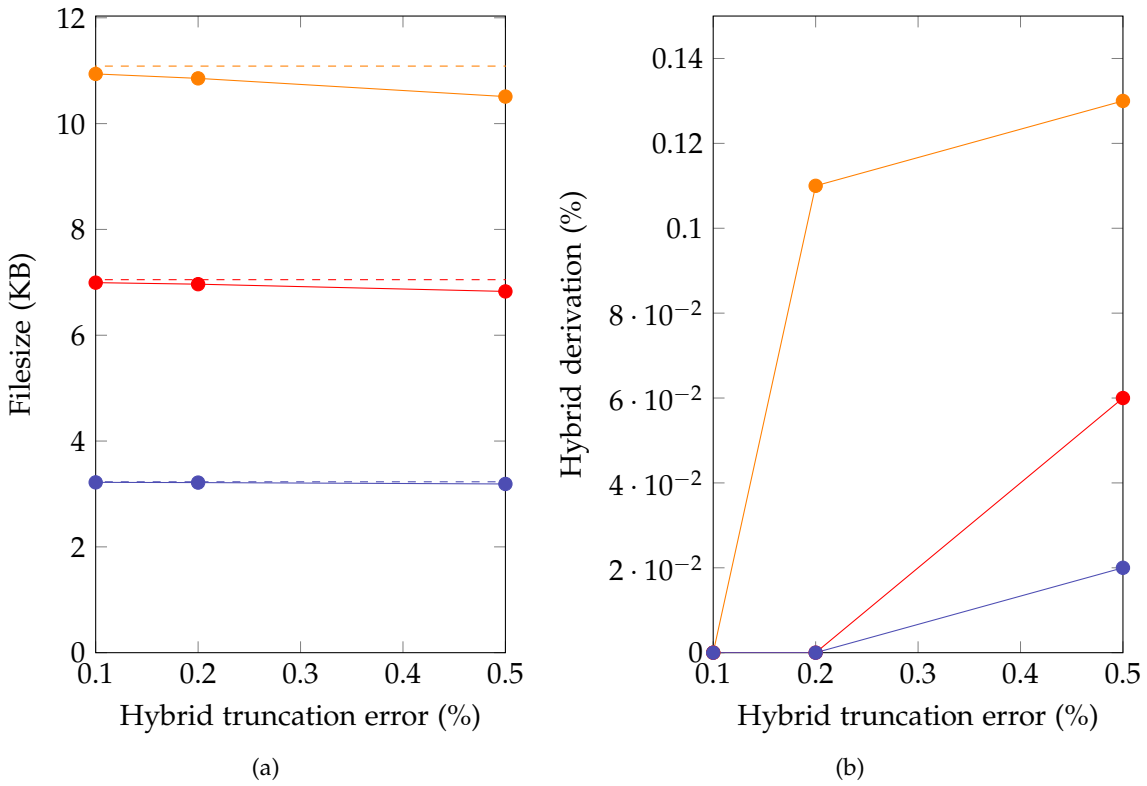


Figure 5.6.: Influence of the TTHRESH target error and truncation error on the overall approximation error ϵ_O and compression filesize S. Numerous correlations between TTHRESH target error (0.03% orange, 0.04% red, 0.05% blue plot) and truncation errors (x-axis) are analyzed. The ratio α on the y-axis plots the overall hybrid compression error vs. the compression filesize and is obtained with the formula $\alpha = (100 - \epsilon_O)/S$. Thereby a higher α indicates a better compression ratio with a smaller overall compression ratio. Although the ratios α peak around a truncation error of 1% for all analyzed TTHTRESH target errors, none of the plots provide intersection points. This suggests a higher scaling and influence of the TTHRESH target error on the achieved compression quality and ratio compared to the truncation error.



(a) (b)

Figure 5.7.: Figure (a) represents a comparison of gained filesizes (y-axis) when introducing an additional error (x-axis) with truncation in the TTHRESH-Truncation hybrid method compared to compression ratios achieved by the original TTHRESH algorithm. For each TTHRESH target error (0,03%: orange, 0,3%: red, 3%: blue plots), three truncation choices with different truncation errors were analyzed. The plotted lines represent the achieved filesizes when compressing the ($277 \times 277 \times 164$) Stagbeetle data set with TTHRESH, the continuous lines refer to compressed filesizes after approximation with the hybrid method, given the TTHRESH target error and additional truncation error. Figure (b) depicts the difference in compression quality between approximation with pure TTHRESH and the hybrid technique (y-axis) when using a specified truncation error (x-axis) on top of the already given TTHTESH target error. The same volume set as in (a) was used for the tests and the same color-coding in regard of the TTHRESH target error applies.

Table 5.7.: Comparison of compressing the ($277 \times 277 \times 164$) Stagbeetle test data set with an TTHRESH approximation error of 0,03% and no additional truncation error in case of the hybrid technique. "Verbatim size" compares the number of entries of the verbatim-vector of the bit-plane coding, "RLE size" the number of elements in the RLE-vectors and "'1'-bits RLE encoded" the number of '1'-value bits encountered during RLE. All three values are the same for both techniques.

Algorithm	Verbatim size	RLE size	"1"-bits RLE encoded
TTHRESH	5,29266 e^7	8,4911 e^6	8,49112 e^6
Hybrid	5,29266 e^7	8,4911 e^6	8,49112 e^6

RLE-vector (see Tab. 5.7). Small core elements are reduced to "0"-values in the bit-plane coding step and can be efficiently compressed with RLE. This translates to the compression of the factor matrices, since corresponding factor columns to the core slices are scaled and reduced to "0"-values. Truncation of the negligible core slices and factor columns does thus not eliminate the costly core elements with "1"-value bits on the bit planes from the encoding process and hence hardly improves the compression ratio. Furthermore, the truncation of the zero factor columns and core slices does also not benefit the verbatim-encoding of the bit-plane coding, since only slices with insignificant norms are cut off. These slices exclusively hold core elements of small magnitudes, which would not be classified as significant by the significance map and are thus not encoded verbatim (Tab. 5.7). Instead, the frequency and length of runs in the RLE-step of the hybrid algorithm are altered, which results in a modified AC frequency model (see Section 5.1.1). An analysis of the AC frequency models (Tab. 5.8, Tab. 5.9) of the hybrid technique and original algorithm reveals the difference in probability distribution between both models: Although both models hold the same number of overall input symbols, the hybrid frequency model holds less unique input symbols than the frequency model of the original TTHRESH algorithm (3255 vs. 3579 unique symbols). The small magnitude symbols of the RLE-vector that appear the most in both models, boast a slightly higher frequency in the hybrid technique model. Overall, symbols that appear only once or twice in the whole RLE-vector are represented less in the frequency model of the hybrid approach. As a result, the input symbols of the hybrid's frequency model are distributed more evenly and the entropy of the RLE-vector declines, thereby ensuing a more effective compression.

Table 5.8.: First ten values of the frequency model of the AC step when compressing the(277x277x164) Stagbeetle data set with TTHRESH (a) or hybrid of TTHRESH and truncation (b). Each symbol from the RLE-vector (left column) is mapped to its frequency of appearing over the whole input vector (right column)

RLE symbol	Frequency
0	3046294
1	1726143
2	1046085
3	667650
4	419041
5	287737
6	195242
7	139162
8	102364
9	78422
10	59546

(a)

RLE symbol	Frequency
0	3052637
1	1738943
2	1062244
3	682511
4	431535
5	298272
6	202725
7	147266
8	111265
9	85548
10	67113

(b)

Table 5.9.: Comparison of frequency models of the TTHRESH (a) and TTHRESH-Truncation hybrid (b) in regards of their frequency distributions. All frequencies of the RLE-symbols were analyzed and ordered according to their occurrences in the frequenzy model in descending order. RLE-symbols that occur only a few times in the RLE-vector have a low frequency, which appears overall the most in the frequency models. On the other hand, recurring symbols tend to have unique high frequencies that emerge only a few times in the models. In general, the hybrid technique boasts slightly less RLE-symbols with a low frequency (e.g. 1726 vs 1971 1-frequencies), thereby reducing its entropy.

Frequency	Occurrence in model
1	1971
2	368
3	197
4	115
5	67
6	53
7	42
8	43
9	34
10	25

(a)

Frequency	Occurrence in model
1	1726
2	353
3	177
4	105
5	68
6	59
7	39
8	40
9	34
10	25

(b)

6. Discussion

As described in Section 5, the main goal of this thesis is to analyze and improve different popular volume data compression algorithms, mainly the TTHRESH algorithm by Ballester-Ripoll et al. [BLP19]. By making use of the hot corner phenomenon as well as advantageous combination of two compression algorithms, we hope to enhance the gained compression ratio, without decreasing overall compression quality. As discussed in Section 5.1.1, it is quite challenging to improve the TTHRESH compression ratio by adjusting the traversal order of the core according to the hot corner phenomenon alone. Although the reordering of the core coefficients according to their Manhattan distance to the hot corner seems promising at first glance, the thereby occurring spikes in core element magnitude of the traversal sequence result in a higher entropy and worse compression ratio than traversing the volume data according to its slice-wise structure.
Changing the underlying arithmetic coding to other entropy coders, like Huffman coding, does also not improve the achieved compression ratio. Entropy coders try to achieve the optimal bit representation of the encoding sequence as indicated by the entropy, but various encoding techniques perform different on individual data sets. Generally, Huffman coding produces the best results when the probabilities of occurrence in the frequency model are powers of ½. On the other hand, AC offers more efficient encoding in general. AC is flexible in that it provides efficient encoding with any probability function, source alphabet or encoding alphabet and is in fact nearly optimal [MM03] and is thus better suited for our approach on volume data compression.
When comparing TTHRESH to other popular volume data compression algorithms, like core truncation, one can notice that TTHRESH is not designed to boast high encoding and decoding speeds, but offers a very fine target error granularity. Since the TTHRESH compression can stop at any arbitrary part of the bit-planes, very smooth approximations of the original data according to the given target error with comparable compression ratios to core truncation can be accomplished. This motivates the use of TTHRESH as a basis of a TTHRESH-Truncation hybrid algorithm for further improvement of the compression ratio at consistent compression quality. However, adding core truncation to TTHRESH as described in Section 5.1.2 does only marginally increase the achieved compression ratio and thus falls short of our expectations. While considerable parts of the core are truncated according to the reduction of zero factor columns and core slices, an improvement of only 1% in compression ratio can be practically realized. The cause of this minor improvement of hybrid compression ratio can be traced back to redundancy between the TTHRESH thresholding and the truncation of zero factor columns and core slices. As the overall compression quality of TTHRESH and TTHRESH-Truncation hybrid is supposed to stay the same, only insignificant core coefficients are removed from the compression during truncation. In particular, the number of overall encoding symbols of

the RLE-step of the TTHRESH-part does not change. Instead, the core truncation influences the frequency of the RLE-symbols, which results in a smoother AC-frequency model and a slightly lower entropy and thus compression ratio. One has to note, however, that the reduction of the core as the consequence of core truncation also reduces the overall amount of reconstruction steps and reconstruction time of the volume data, since the values of large parts of the compressed data are intrinsically known to the algorithm and do not have to be decompressed. As a result the hybrid approach boasts slightly better reconstruction time of the original data, which can be helpful for many analytic visualization purposes.

7. Conclusion

In this thesis, we implemented and analyzed the TTHRESH- as well as core truncation volume compression algorithm. Approaches to improve the TTHRESH fell short of our expectations and did only marginally improve the achieved compression ratio and reconstruction time. Firstly, the proposal to adjust the volume traversal order in an profitable way by incorporating the hot corner phenomenon did not increase the acquired compression ratio. Compared to the natural in-memory traversal of the HOSVD core, the altered traversal by Manhattan distance to the hot corner did disturb the HOSVD core's natural slice-wise structure and thus increased the overall entropy of the encoded message. Our second approach with the truncation of insignificant zero factor columns and core slices did reduce the tensor and the amount of volume data. However, this truncation had no influence on the overall amount of actual encoding data and is thus not very beneficial for improving the compression technique. There still exist multiple open issues and methods that were not tested in this thesis but could potentially benefit the TTHRESH compression ratio.

7.1. Outlook

We encountered numerous problems during the improvement of the TTHRESH-algorithm. Although the changing of the traversal order of the HOSVD core according to the hot corner phenomenon is promising from a theoretical point of view, the thereby introduced spikes in core coefficient magnitude make it quite hard to improve the gained compression ratio. Instead, we propose the slice-wise structure of the HOSVD core as a possible basis for future improvements of the TTHRESH-algorithm. To that effect our approach to factorize the RLE-symbols according to the size of the tensor dimensions does also not enhance the compression ratio, as too few symbols are affected by this factorization. A possible alternative to the factorization of RLE-symbols is the replacement of RLE-symbols according to the dimensional sized of the volume data: Even if the RLE "0"-bit runs along the bit-planes of the TTHRESH bit-plane encoding are interrupted by a "1"-value bit and do not form a smooth run to match the size of the traversed dimension (in case of the ($277 \times 277 \times 164$) Stagbeetle data set runs of size 276), the runs tend that sum up according to the dimensional size of the traversed core slice. For example for the ($277 \times 277 \times 164$) Stagbeetle data set, many consecutive RLE-runs along the bit-planes can be arranged into groups to sum up to 276. The last RLE-symbol of the group can then be removed and is replaced with just the instruction for the decoder to sum up to 276. This way, a RLE-symbol of low frequency may be replaced with a RLE-symbol with high frequency, which results in an overall lower entropy for the encoding message and a more efficient compression by AC.

A. Figures

A.1. Example of Tensor Unfolding

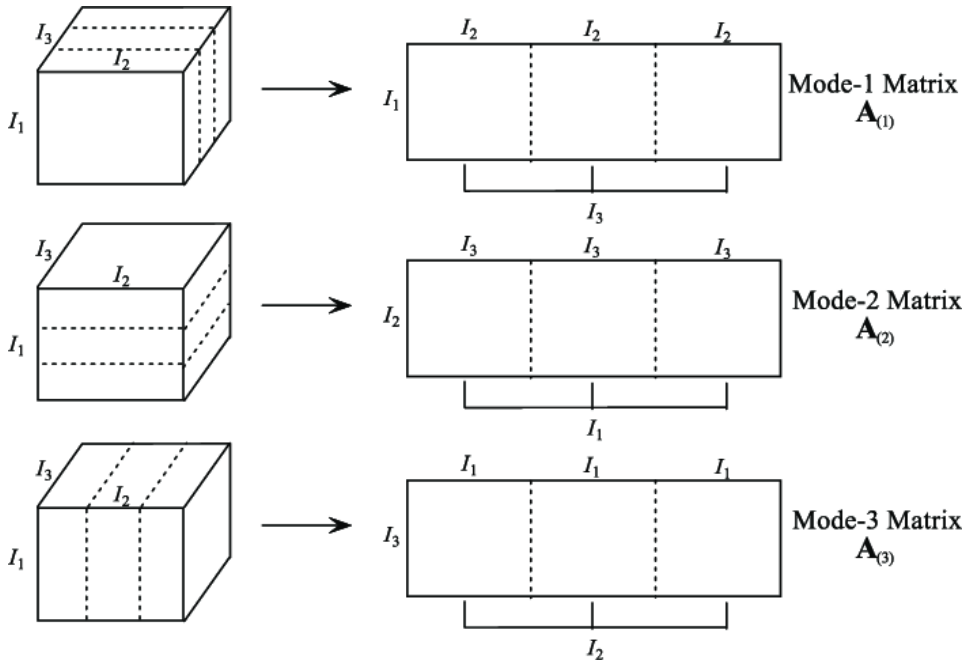


Figure A.1.: Example of unfolding the $(I_1 \times I_2 \times I3)$-tensor T into the three mode-i matrices by ordering the i-mode fibres of the tensor as the columns of the matrices. Thus, the mode-1 matrix A_1 is of size $(I_1 \times I_2 I_3)$, the mode-2 matrix A_2 of size $(I_2 \times I_3 I_1)$ and the mode-3 matrix A_3 is of size $(I_3 \times I_1 I_2)$. Adapted from [Qia+09].

A.2. TTHRESH-Truncation Hybrid Flowchart

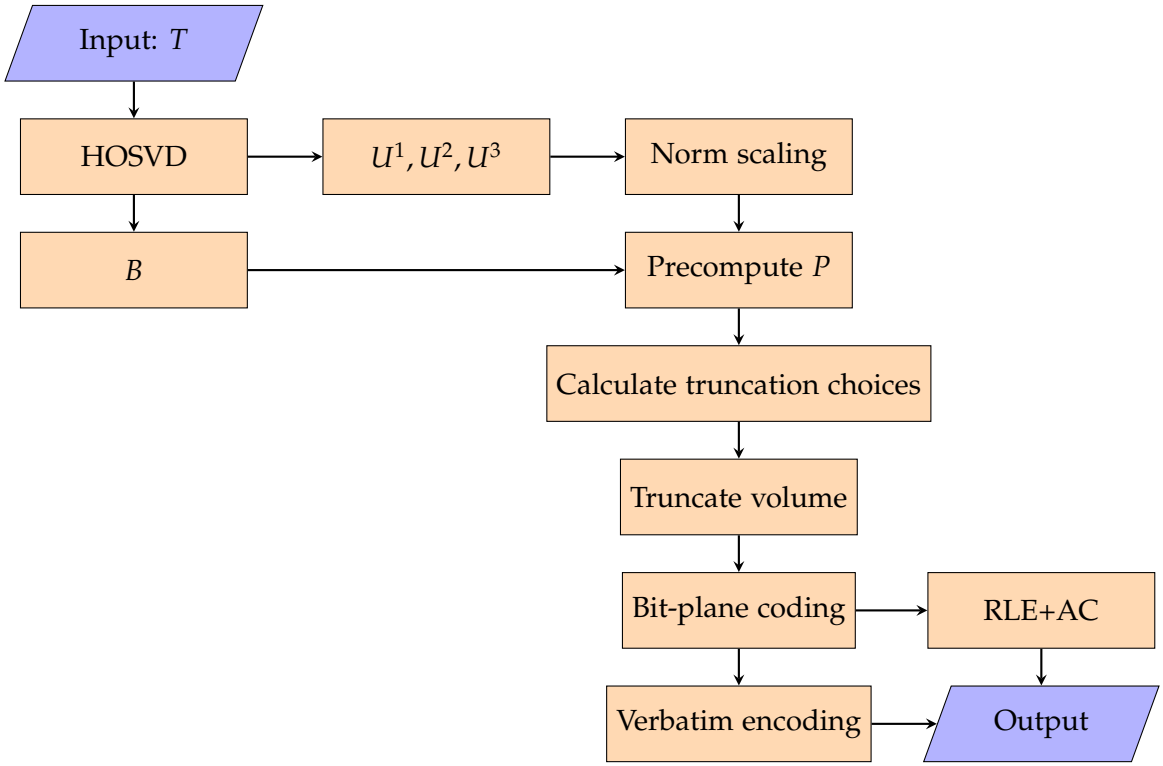
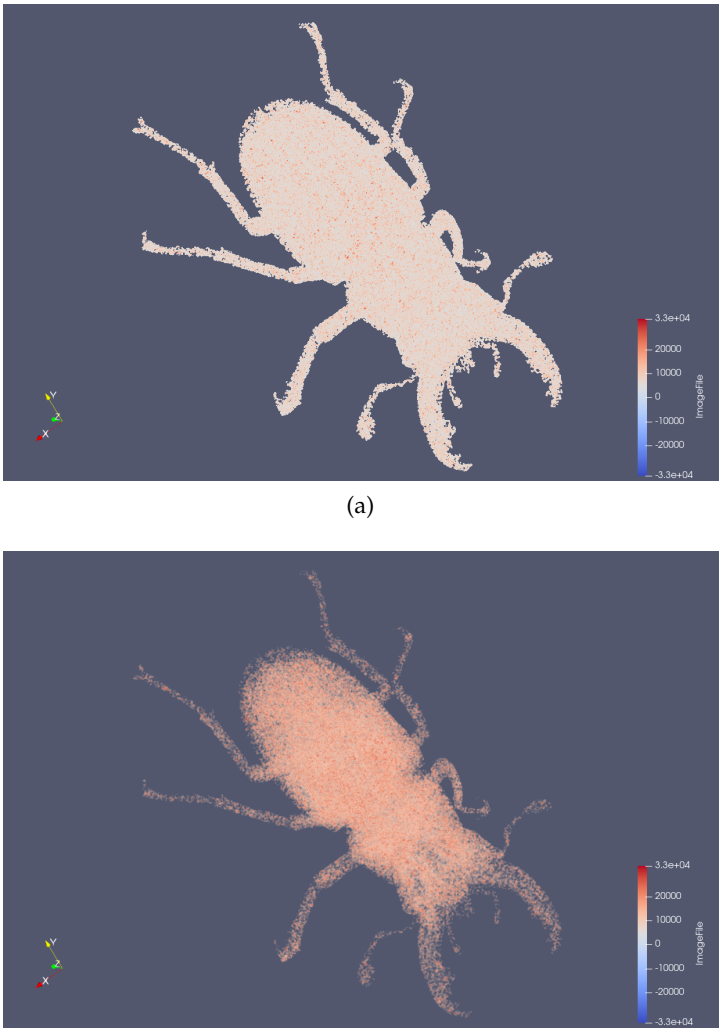


Figure A.2.: Flowchart depicting the TTHRESH-Truncation hybrid algorithm. For the most part, the algorithm is similar to the original TTHRESH algorithm described in Section 4. The sections "Precompute P", "Calculate truncation choices" and "Truncate volume" deviate from the original algorithm. As described in Section 5.1.2, the precomputation of the stopping plane P allows to asses the factor columns and core slices, which would hold only zero-values after bit-plane encoding. The truncation choices are then calculated in such a way that only the described zero factor columns and core slices are truncated from the volume and the overall amount of volume data is reduced. After the truncation the hybrid algorithm continues as the original TTHRESH.

A.3. Rendering Results of TTHRESH and TTHRESH-Truncation Hybrid



(a)

(b)

Figure A.3.: Rendering of the ($277 \times 277 \times 164$) Stagbeetle test set with ParaView [Aya15]. On a) the original uncompressed volume set is depicted (filesize 24.578 KB). On b) the compressed file is illustrated, when encoding the testset with TTHRESH with a target error of 0.03%, realizing a filesize of 11.087 KB.

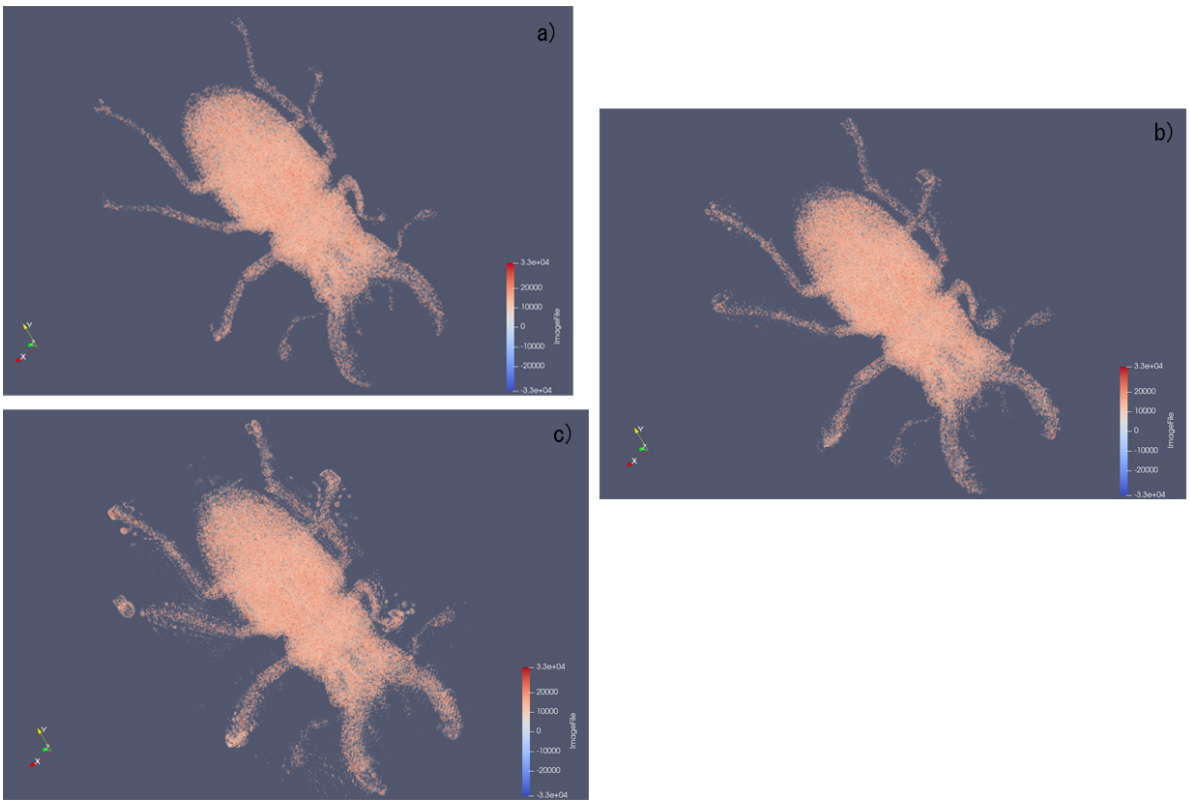


Figure A.4.: Rendering of the ($277 \times 277 \times 164$) Stagbeetle test set with ParaView [Aya15] when employing the TTHRESH-Truncation hybrid algorithm with a TTHRESH target error of 0.03%. On a) the algorithm is applied with an additional truncation error of 0.1% (filesize 10.939 KB), on b) an additional truncation error of 0.5% (filesize 10.525 KB) is used and on c) the algorithm is utilized with an additional truncation error of 1% (filesize 9.928 KB). One can notice that a) is visually equivalent to the rendering result of the pure TTHRESH algorithm (Fig. A.3b). On b) first minor rendering artifacts around the beetle's legs appear and in c) major artifacts on the stag beetle's whole body emerge.

List of Figures

List of Tables

Glossary

Bit-plane coding Compression technique that traverses over the HOSVD core elements and iteratively compresses the same bit from the element's binary representation (bit-plane), starting from the most significant bit-plane. 1, 18–21

Entropy Information content that describes the degree of randomness and uncertainty in a message. When encoding, the entropy gives a lower bound on the expected number of bits needed to represent each symbol of the input message. A low entropy suggests that the message can be encoded efficiently. 6, 7, 18, 19, 22, 23, 25, 26, 29, 30, 32, 38, 40, 41

Fibre Higher order analogues to matrix rows and columns of a tensor. Obtained by iterating over the tensor elements and fixing all but a single index along the dimensions. 3, 22, 28, 44

HOSVD Higher order single value decomposition: Decomposes original tensor T into core tensor B and factor matrices U^i. 3, 4, 10, 12–17, 19–23, 25, 33, 43

Hot Corner Element $B(1,1,1)$ of the HOSVD core. The core elements with the highest absolute magnitude concentrate around this element. 4, 5, 11, 13, 18, 22, 24–26, 29–31, 41, 43

Memory ordering Representation of a tensor's elements as stored in-memory. This thesis assumes that a 3D tensor is represented in memory as a cluster of consecutive mode-1/ y-dimension fibres, starting from the leftmost fibre of the foremost mode-1 slice and stretching to the rightmost fibre of the hindmost mode-1 slice. 22, 23, 25, 26, 29, 30

Run Sequence of the same consecutive data values in a row that is compressed to a single symbol by RLE. 6, 18, 24–26, 29, 30, 35, 38

Slice 2D section of a 3D tensor. Obtained by iterating over the tensor elements and fixing all but two indices along the dimensions. 3, 10, 11, 19, 21–23, 25, 28, 29, 33–35, 38, 41, 43

Tensor Higher order arrays of dimension $N \geq 1$. In this thesis, we refer to a tensor as a multiarray of dimension 3. 1–5, 11–13, 15–17, 19, 21, 23, 25, 28, 30, 33, 34

TTHRESH Lossy volume compression algorithm by Ballester-Ripoll et al. [BLP19]. 1, 16, 19, 21–24, 28, 30, 33–43, 45, 49

Zero factor columns and core slices Core slices and corresponding factor matrix columns that contain only coefficients with the value "0" after TTHRESH thresholding by bit-plane coding and core slice norm scaling. 34, 35, 38, 41, 43, 45

Acronyms

AC Arithmetic coding. 6, 7, 16, 18, 19, 21, 22, 24–27, 29, 30, 32, 33, 38, 39, 41, 42, 48, 49

MLOC Multi-level Layout Optimization Framework for Compressed Scientific Data. 10

RLE Run-length encryption. 6, 16, 18, 19, 21–27, 29, 30, 32, 33, 35, 38–40, 42

SAT Summed-area table. 11

SSE Sum of squared errors. 17–19

SVD Singular Value Decomposition. 3, 4, 17, 19

TTM Tensor times matrix product. 3

Bibliography

[BLP19] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola. "TTHRESH: Tensor compression for multidimensional visual data". In: *IEEE transactions on visualization and computer graphics* (2019).

[Bus20] M. Bussler. *Bachelor thesis TTHRESH code on Github*. `https://github.com/Bussler/bachelorTTHRESH`. Accessed: 2020-08-11. 2020.

[HJ11] C. D. Hansen and C. R. Johnson. *Visualization Handbook*. Academic Press, August 30, 2011.

[Eng+04] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. "Real-time volume graphics". In: *ACM Siggraph 2004 Course Notes*. 2004, 29–es.

[De 09] L. De Lathauwer. "A survey of tensor methods". In: *2009 IEEE International Symposium on Circuits and Systems*. IEEE. 2009, pp. 2773–2776.

[DDV00] L. De Lathauwer, B. De Moor, and J. Vandewalle. "A multilinear singular value decomposition". In: *SIAM journal on Matrix Analysis and Applications* 21.4 (2000), pp. 1253–1278.

[KB09] T. G. Kolda and B. W. Bader. "Tensor decompositions and applications". In: *SIAM review* 51.3 (2009), pp. 455–500.

[Zha+14] Z. Zhang, G. Ely, S. Aeron, N. Hao, and M. Kilmer. "Novel methods for multilinear data completion and de-noising based on tensor-SVD". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 3842–3849.

[Li+18] S. Li, N. Marsaglia, C. Garth, J. Woodring, J. Clyne, and H. Childs. "Data reduction techniques for simulation, visualization and data analysis". In: *Computer Graphics Forum*. Vol. 37. 6. Wiley Online Library. 2018, pp. 422–447.

[SW09] M. Stabno and R. Wrembel. "RLH: Bitmap compression technique based on run-length and Huffman encoding". In: *Information Systems* 34.4-5 (2009), pp. 400–414.

[HV94] P. G. Howard and J. S. Vitter. "Arithmetic coding for data compression". In: *Proceedings of the IEEE* 82.6 (1994), pp. 857–865.

[SV10] M. Sinaie and V. T. Vakili. "Secure arithmetic coding with error detection capability". In: *EURASIP Journal on Information Security* 2010.1 (2010), p. 621521.

[BP16] R. Ballester-Ripoll and R. Pajarola. "Lossy volume compression using Tucker truncation and thresholding". In: *The Visual Computer* 32.11 (2016), pp. 1433–1446.

[Bha18] J. Bhattacharjee. *fastText Quick Start Guide*. Packt Publishing, July 26, 2018.

[Gon+12] Z. Gong, T. Rogers, J. Jenkins, H. Kolla, S. Ethier, J. Chen, R. Ross, S. Klasky, and N. F. Samatova. "MLOC: Multi-level layout optimization framework for compressed scientific data exploration with heterogeneous access patterns". In: *2012 41st International Conference on Parallel Processing*. IEEE. 2012, pp. 239–248.

[Sut+11] S. K. Suter, J. A. I. Guitian, F. Marton, M. Agus, A. Elsener, C. P. Zollikofer, M. Gopi, E. Gobbetti, and R. Pajarola. "Interactive multiscale tensor reconstruction for multiresolution volume visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2135–2143.

[Fac20] T. W. Faculty of Informatics. *Visualization Group Data sets*. `https://www.cg.tuwien.ac.at/research/vis/datasets/`. Accessed: 2020-08-07. 2020.

[MM03] D. J. MacKay and D. J. Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[Qia+09] X. Qiao, R. Xu, Y.-W. Chen, T. Igarashi, K. Nakao, and A. Kashimoto. "Generalized N-Dimensional Principal Component Analysis (GND-PCA) based statistical appearance modeling of facial images with multiple modes". In: *Information and Media Technologies* 4.4 (2009), pp. 999–1009.

[Aya15] U. Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitware, 2015.