COMPACT AND PROGRESSIVE ENCODINGS FOR TASK-ADAPTIVE QUERIES OF SCIENTIFIC DATA

by

Duong Hoang

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2023

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Duong Hoang**
has been approved by the following supervisory committee members:

Valerio Pascucci,	Chair(s)	**06/05/2023** Date Approved
Aditya Bhaskara,	Member	**05/29/2023** Date Approved
Christopher R. Johnson,	Member	**05/24/2023** Date Approved
Robert Michael Kirby II,	Member	**05/23/2023** Date Approved
Peter Lindstrom,	Member	**05/24/2023** Date Approved

by **Mary Hall**, Chair/Dean of
the Department/College/School of **Computer Science**
and by **David B. Kieda**, Dean of The Graduate School.

ABSTRACT

As computational power outpaces bandwidth, handling multi-petabyte datasets, which are easier to produce than to store or access, becomes a challenge. Traditional solutions focus on data reduction through lossy compression or downsampling, but many overlook considerations like the scientific tasks to be performed on the data or the computational resources required. Additionally, there's a lack of a common framework to gauge these techniques' efficacy and costs. This dissertation introduces such a framework and presents novel data reduction techniques offering more holistic considerations and compelling trade-offs.

A novel hierarchical data model that unifies spatial resolution and numerical precision is developed. Most current reduction techniques focus on either resolution or precision, but this new model allows for approximations anywhere in the 2D space of resolution and precision, offering refined approximation in either axis without necessarily refining the other.

A novel progressive streaming framework is proposed to study trade-offs between reducing resolution and precision during common analysis tasks. Using the wavelet transform, both resolution and precision-based reduction techniques can be modeled as different orderings of wavelet coefficient bits. This enables the first-ever demonstration that reducing both resolution and precision is significantly more effective than reducing either alone. It further allows the optimization of bit streams for specific analysis tasks, offering insights about how different tasks fetch data.

Lastly, the new data model is adapted to encode unstructured particle data, a common form of data representation in scientific simulation or imaging. Borrowing techniques from the structured case, it's shown that the odd-even decomposition can significantly reduce reconstruction error while maintaining a constant memory footprint. This results in novel particle hierarchies and optimal traversal heuristics that outperform traditional approaches in progressive decoding quality-cost trade-offs.

To my wife, Trang

CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

For thousands of years, science was largely *empirical*, whereby humans observed and described natural phenomena without a unifying framework. A few hundred years ago, we began developing *theoretical* models to generalize the various empirically observed phenomena into laws of nature. In the last few decades, since the advent of computers, we started using them to *simulate* models too complex to be solved analytically. At the same time, technological advancements have also produced fast instruments that can capture vast amounts of information in real-time. These developments bring a new, *data-intensive* model of science, in which scientists gain insights and breakthroughs by using computers to study data captured by instruments or generated by simulations. Together with theory and experiment, simulations and data analysis from the four pillars of science, with data analysis referred to as the "fourth paragidm" [112].

With the exponential increase in compute power and the increased availability of high-throughput instruments, both simulations and experiments can now produce data at speeds and sizes that overwhelm most scientific workflows. In other words, our ability to produce data has outpaced our ability to process and study them, leading to the so-called data deluge [17]. It is common nowadays for high performance computing (HPC) centers to generate several terabytes or more of data per simulation run, as well as for experiment facilities to capture similar amounts of data per session. This explosion in data generation has been made possible through an exponential increase in computational power, fueled by decades of chip making technology. Memory bandwidth, on the other hand, has not kept pace [191]. Memory devices, such as RAM and solid-state drives, have not seen the same rapid advancements, thus the speed at which data can be transferred to and from memory remains limited. Therefore, it has become overwhemingly challenging to store, manage, and transfer scientific data to serve cutting-edge science. As a result, the rate at which scientific insights can be obtained by studying the generated data using computers has been severly impeded. It is the goal of this work to ameliorate the data deluge problem.

1.1 The role of large data in modern science

To study scientific phenomena, scientists use a variety of tools and methods, of which two of the most important are computer simulations and observations. Simulations use mathematical models to represent the behavior of a system and predict how it will respond to different conditions and inputs, and can be used to study a wide range of phenomena, from fluid dynamics to molecular interactions. Computer simulations enable scientists to better model and understand complex systems at scales difficult to capture in reality, make predictions about future events or the behavior of systems under various conditions, test and validate theoretical models or experimental results, and conduct experiments in ways that may be impossible, unethical, or too costly in the real world. At the same time, large-scale and high-resolution observational data obtained by scanning or imaging devices can also be used to study the behavior of physical systems, such as the distribution of matter in the universe or the behavior of materials under different conditions. Such data enable researchers to see fine-scale details previously inaccessible or difficult to detect, provide comprehensive spatial and temporal coverage of the phenomena being studied across vast geographical areas and over extended periods, as well as improve the validation and calibration of scientific models. Whether through simulations or observations, by collecting data and analyzing it using mathematical techniques, scientists can gain new insights into the underlying structures and behaviors of the systems under study. These insights can then be used to develop new theories and models, leading to a deeper understanding of the natural world and how it works.

To give concrete examples, the study of turbulent fluid mixing and related instabilities, such as Rayleigh-Taylor and Richtmyer-Meshkov [320] through computational methods such as DNS (direct numerical simulation) [203], is essential for understanding complex fluid dynamics phenomena and has wide-ranging applications in various fields, including astrophysics, fusion energy, high-energy-density physics, atmospheric and oceanic dynamics, combustion processes, and geophysics. Combustion scientists study flame dynamics such as the stability of turbulent flames, with the hope to design more efficient fuel with less pollution [310]. Neuroscientists study brain function, which emerges from the coordinated activity of neurons, by visualizing individual fluorescently labeled axons using three-dimensional microscopy image data captured at high resolution and high magnification. Such images of labeled neurons can range in size from gigabytes to terabytes [289, 293]. Climate scientists study Earth's atmosphere and ocean circulation by combining data from satellites, in-water instruments, and computer models to solve fluid flow equations [195,269]. The generated data can be on the order of petabytes [71]. In material science, understanding characteristics of foam built from a network of interconnected ligaments is essential for exploiting the unique properties of open-cell metallic foams and lattices [236]. Here, a data-driven approach using high-resolution CT scans can uncover the relationships between attributes like ligament length, connectedness of junctions, and pore size distribution as ligaments deform or fracture during compressive loading, leading to the development of innovative materials with tailored properties for a wide range of applications across various industries. In cosmology research, dark matter halos are over-dense regions of dark matter particles that serve as the building blocks for cosmic structures and provide the gravitational potential wells where baryonic matter (such as gas and stars) can accumulate and form galaxies. Finding such halos efficiently from high-resolution cosmology simulation data [309] that can reach several terabytes in size [85] is a critical step that provide insights into structure formation, galaxy and black hole evolution, dark matter distribution and gravitational lensing. The National Ecological Observatory Network (NEON) is a large-scale ecological observation facility focusing on capturing and sharing environmental images as well as biological sampling to facilitate a better understanding of complex ecological processes and changes, predict the impacts of environmental change, and develop strategies for managing and conserving ecosystems [134]. A single flight can generate several terabytes of raw data, and considering that the multiple flights per year are conducted across various NEON field sites on a continental scale, the total volume of remotely sensed data generated can easily reach tens to hundreds of terabytes annually.

1.2 Scientific data analysis and visualization

Scientific data analysis involves various techniques and methods to process, interpret, and extract meaningful information from raw data. Major categories include signal or image processing, geometry processing, statistical computation, topological data analysis, machine learning, dimensionality reduction, and data visualization. I briefly summarize these categories below.

- Data visualization represents data in graphical or pictorial forms. Human can process pictures much faster than text or numbers, hence scientific visualization plays a crucial role in understanding, interpreting, and communicating scientific information. Some observational data such as medical scans or aerial images are inherently images. For simulation data, visualization often means mapping the data values of a 2D or 3D field into colors and render the data as if it is being looked at from a virtual camera [72]. Visualizing the data helps researchers better explore the data, identify trends or patterns, and communicate results more effectively.

- Image processing involves both generic, pixel-level and semantic-based, object-level extraction and manipulation methods. Pixel-level procedures include mostly signal processing operations such as filtering, smoothing, sharpening, denoising, etc to help make specific features or details more salient. The semantic-level procedures deal with concepts such as corners, edges, shapes and objects. They make use of the pixel-level procedures to perform object recognition, classification, matching, as well as image segmentation and registration.

- Machine learning is used to uncover patterns, relationships, and insights that may not be readily apparent through traditional analytical techniques. Machine learning techniques can be used for tasks such as classification, regression, clustering, anomaly detection, forecasting, and filling in missing data using a learned model.

- Dimensionality reduction involves techniques that reduce the number of variables or features in a dataset while preserving its essential structure and relationships. Dimensionality reduction can help simplify data analysis, make the data easier to understand, reduce computation time, and minimize the effects of the "curse of dimensionality".

- Geometry processing refers to the techniques and methods used to manipulate, analyze, and process geometric data, which are often represented as points, lines, curves, surfaces, or volumes. The set of geometry processing methods are diverse, including mesh generation, simplification and smoothing, geometric transformations, geodesic computation, segmentation, shape analysis and feature extraction, surface parameterization, and computational geometry.

- Statistical computations are used to understand, quantify, and interpret the underlying patterns, relationships, and uncertainties in the data. Statistical methods help researchers draw conclusions about a population based on a sample (inference), summarize and describe the main characteristics of a dataset (*e.g.,* mean, median, modev, ariance, standard deviation, range), testing hypotheses, handle missing or noisy data, building and validating models, and quantify uncertainty associated with estimates, predictions, and decisions. All of these ensure the validity, reliability, and generalizability of scientific findings.

- Topological data analysis (TDA) leverages topology, the study of shapes and their properties, to extract meaningful information from complex and high-dimensional data. TDA has gained increasing importance in scientific data analysis due to its robustness to noise, invariance to transformations, ability to handle high-dimensional data, multi-scale analysis capabilities, and interpretability. Popular TDA methods in scientific data analysis include persistence homology (capturing the "birth" and "death" of topological features, such as connected components, loops, and voids, as the scale of analysis changes), Morse and Morse-Smale complexes (capturing the relationship between the critical points of a real-valued function and the topology of its level sets to identify and analyze features in data, such as clusters, voids, and ridges), Reeb graph (capturing the evolution of level sets of a real-valued function on a manifold, useful for feature detection, shape analysis, and data simplification), and merge trees (capturing the evolution of connected components of the sublevel sets of a real-valued function, useful for studying the hierarchical structure of data and identify significant topological features, such as clusters and voids).

1.3 Reduction and simplification of scientific data

With exponential increase in computing power, today's supercomputers regularly generate several hundreds of terabytes of data per simulation run. A key challenge of scientific data analysis for large data is limited input/output (I/O) bandwidth. This is because, when processing large datasets, the time taken is often dominated by the time required to transfer the data to and from memory. Two major bottlenecks exist. First, transferring very large datasets between remote sources, as well as accessing such data on remote servers, can be slow due to the limited network bandwidth. Second, even after the data is read and transferred to the destined machine(s), ready for analysis, processing large datasets is also very slow due to the limited memory bandwidth within individual machines. The problem can be alleviated somewhat by distributed processing, however, the computational resources available for data analysis and visualization are usually orders of magnitude smaller than that available for the simulations generating the data. As a result, these bandwidth bottlenecks seriously hamper efficient data analysis and therefore significantly slow down scientific progress.

Several approaches have been proposed to address this challenge, including: using high-performance I/O technologies such as InfiniBand or Fiber Channel, caching the data locally and pre-processing the data before analysis, using fast intermediate storage such as burst buffers, using distributed computing resources to process data in parallel, optimizing the data analysis algorithms so that they run faster and require less data, in-situ analysis that analyzes the data where it is generated, and reducing the size of data through lossless or lossy compression. This work focuses on the last approach: reducing data size through lossy compression. Of the above approaches, lossy compression has the most potential for the largest impact: it is possible to reduce the storage and data movement time by several orders of magnitude, making it possible to visualize and analyze data on consumer computers instead of clusters or supercomputers.

The input data forms the foundation of the analysis that is to be conducted. The finite set of data points form a discrete sampling of space, meaning that the data represents a discrete and limited representation of a larger, continuous phenomenon. Each data point is associated with a vector of real-value quantities, which can represent various physical, chemical or biological properties of the phenomenon being studied. The data points can come from a variety of sources. For instance, they can be discrete samples of a solution to a partial differential equation, which describes the behavior of a physical system in terms of mathematical relationships between variables. On the other hand, the data points can also be discrete samples of an image taken by a scanning or imaging device, such as a CT scan or a microscope image. If we define the *resolution* of a discrete sampling of a dataset as the density of the sampling points (finer resolution means denser sampling), and the *precision* of the same dataset to be the number of bits used to represent the data value at each sample point, then lossy compression of the dataset often means reducing its resolution and/or its precision.

To illustrate the effectiveness of lossy reduction in precision and resolution, consider Figure 1.1 and Figure 1.2, which demonstrate that volume rendering and isocontour extraction can be performed at reduced precision and resolution without significant loss in the results. This is not surprising since volume rendering and isocontour extraction are lossy processes themselves, in that they map the input data into outputs that contain only a fraction of the original information. In fact, most data analysis procedures (*e.g.,* the ones mentioned in Section 1.2 are lossy: while computers and instruments can certainly generate and capture fine-scale phenomena over a large spatial volume or area, in order for humans to comprehend and make sense of the resulting data, it cannot be presented in raw form all at once, but has to be reduced, simplified, or summarized. Data visualization is a typical example: a 3D dataset is rendered to a 2D screen (of typically less resolution) and its samples mapped to color values (of much less precision) for visualization purpose. This process alone involves three forms of data reduction, in dimensionality, precision and resolution. It makes sense therefore, to avoid transferring and loading the entire data into memory before performing analysis. Instead, data reduction should happen before analysis, with each analysis task ideally loading only the subset of data that it truly needs.

To answer scientific questions with required accuracy, not every bit of data is necessary. In particular, for techniques at the end of scientific workflows, such as visualization and data analysis, lower fidelity representations of the data often provide adequate approximations [91, 152, 315]; even during simulation, some loss in precision is often acceptable [32,152]. As a result, several different techniques have been proposed to reduce the size of data. Reducing data size and fidelity primarily takes two forms: reducing resolution (*i.e.,* number of data points) and reducing precision (*i.e.,* the number of bits representing each data value). Generally, both types of techniques transform the original data into multiple "levels", such that meaningful information is disproportionately more concentrated in the first few levels, leaving subsequent levels with inessential information to be discarded. For resolution-based techniques, these levels often manifest in the form of a tree [251], a hierarchical space-filling curve [223], adaptive mesh refinement [19], octrees and other tree structures [250], or a wavelet hierarchy [315]. With precision-based techniques, data samples are often decorrelated to form levels of precision, e.g., with energy concentration transforms [16,69], followed by quantization to truncate away lower order precision levels (e.g., least significant bit planes). Some of the most well-known quantization-based compressors for scientific data are ISABELA [150], ZFP [169], FPZIP [171], SZ [58,164,165,280,323], VQ [209], SQ [128], HVQ [256], and TTHRESH [12]. Traditionally, multiresolution structures have been used to accelerate dynamic queries, e.g., in rendering [151], since discarding data points based on the viewpoint or data complexity can result in significant speed-up. Compression based on quantization, on the other hand, is more common when storing data, where in the absence of other information, treating each sample as equally important is the null hypothesis.

1.4 Problem statement

Most existing data reduction methods are designed for a single purpose, thus they consider very narrow sets of trade-offs. Precision-based compression methods mostly focus on optimizing for data size, most of the time at a single rate, requiring data to be fully decompressed at the finest resolution to be usable. This often means the data can only be decompressed on hardware similar to what was used to compress the data in the first place, significantly limiting the usability of the compressed data. On the other hand, while resolution-based reduction methods often allow level-of-detail access to the data using less powerful hardware, most do not employ very effective precision-based compression schemes, and most also are only suitable for visualization purposes. The end result is that in practice, scientists are often left with no tool to interactively work with terabyte-size data using the conventional hardware that they use everyday, and must resort to using much larger machines with roundtrip times on the order of weeks and often with the indispensable help of the computer scientists.

Furthermore, in many situations, a combination of both resolution and precision reduction may be appropriate. For example, high spatial resolution may be needed to resolve the topology of an isosurface, yet the corresponding data samples may be usable at less than full precision to adequately approximate the geometry. Conversely, accumulating accurate statistics may require high-precision values, yet a lower resolution subset of data points may be sufficient for the task. Toward enabling interactive visualization and analysis of tera-scale datasets, this thesis considers the problem of compact encodings for large scientific data more holistically, since this is really a multi-objective optimization problem. There are multiple challenges to this problem:

- **Multiresolution and multiprecision access.** Multiresolution access refers to the ability to reconstruct low-resolution approximations of the data by fetching a subset of the bits. Not only multiresolution access enables multiscale analysis important for many scientific analyses due to features and phenomena happening at different scales, but it can potentially reduce the amount of data fetched by orders of magnitude by greatly reducing the number of discrete sample points. To fully take advantage of multiresolution access, however, the resolution must be able to vary across space so that it can adapt to the data. Similarly, multiprecision access allows fetching subsets of the bits to reconstruct a low-precision approximation of the data. This is useful since most analysis tasks do not need the precision at which the data was generated and possibly stored. The precision should also be allowed to vary across space to really adapt to features of the data. While precision-based reduction and resolution-based reduction works well on their own, there is great potential for combining both approaches to achieve even better compression. However, precision-based compression often require grouping and quantizing neighboring samples exploit their spatial coherency, while to reduce resolution is to discard neighboring samples to reduce the sampling rate. This inherent conflict is perhaps a reason why most resolution-based reduction methods are not combined with a competitive precision-based quantizer.

- **Progressive access.** This refers to the ability to generate gradually better appoximations of the data by fetching and decoding more bits. This capability is important, since it is difficult to know in advance at what precision or resolution the user scientists would need to perform their tasks under given time and resource constraints. A crude overview of the data for exploration purpose done on a laptop may need very coarse resolution and precision, but a more complex workflow with involved calculations on the data may require approximations with much higher fidelity. Therefore, the ability to refine approximations without re-fetching or re-decoding data is crucial. Progressive encoding is in contrast to *single-rate encoding*, which most lossy compressors for scientific data employ, presumably due to the latter's relative simplicity. With progressive access, scientists are not stuck to a single quality, and also not having to choose this quality from the beginning. When needed, the current approximation should also improve in either precision or resolution independently. This is a challenge in the presence of compression, since many compression techniques often leverage the relative difference in magnitude between data samples at different resolution levels, hence these levels are coupled and compressed together.

- **Speed and memory requirements.** To support progressive updates, certain states must be maintained in memory so that the decoder knows how to resume. Depending on the encoding scheme, these states maintained for progressive access and decoding can be very large, to the point that they can dwarf the actual decoded data. A useful decoder should not only be light in memory but also be fast, while not sacrificing too much compression effectiveness. These two goals are usually in conflict because better compression ratio is often achieved by making the compressor more complex, hence slower. For speed, a crucial property for a decoder running on modern machines is the support for parallelism, *e.g.,* the data is compressed in blocks that can be independently decompressed in parallel. However, this reduces the effectiveness of compression, and often results in blocky artifacts at the block boundaries during reconstruction.

- **Adaptivity to analysis tasks.** Different analysis tasks need different subset of the input data, hence it is important to identify such subsets given an analysis task at hand, to avoid accessing and transferring more data than necessary. Given that a reduction model that combines resolution and precision, there is a need to determine the trade-off between more resolution (and less precision) or vice versa for any given analysis task. There is currently a lack of formal framework to reason about this resolution-precision trade-off; for example, when comparing precision reduction and resolution reduction at the same data size budget of, say, one-third the original, it is relatively easy use a third the number of precision bits, but it is unclear what the equivalent reduction in resolution is. To quantitatively evaluate this trade-off, and in general, to progressively identify the set of bits best for a given analysis task given a size constraint, it is necessary to define an error metric for each task. This is highly nontrivial, since for most tasks there is no agreed-upon standard error metric, especially one that correlates highly with human perception. Even when an error metric is defined, it is unclear how to quickly identify the best bits that will minimize this metric (a problem akin to *rate-distortion optimization* in the lossy compression literature) and then communicate this set from the encoder to the decoder without using a significant amount of extra bandwidth.

- **Subset accesses.** Even when such a subset of bits that best serve an analysis task at hand can be identified, it is difficult to just read these bits from the storage without touch other bits. This is because most storage mechanisms are block-based, so data is read in blocks of at least several kilobytes. As such, the order in which bits are laid out on disk and how they are grouped together to form larger I/O units are crucial to minimize the I/O overhead of fetching unwanted bits. A scientist user may issue very fine-grain queries in resolution and/or precision queries, *e.g.,* a query that asks for specific resolution and precision levels, in which case it is desirable for the data layout to allow reading just those levels from disk. The desired resolution can also vary in dimensions: a query may request for denser data samples along certain dimensions of the data than others. Beside resolution and precision, the user often also wants subsets of data restricted to certain spatial extents (*i.e.,* regions of interest), or certain data value range extents (*e.g.,* isocontour extraction). Since the bits can only be stored in a single layout serving multiple such varied access patterns, the challenge is how to design the layout to serve the common patterns well enough, since obviously it is impossible for a single layout to best serve any access pattern.

- **Local and remote access.** It is rare that large-scale datasets are copied from one storage to another due to the high cost of doing so. Thus, the subdomain, subrange, and on-demand accesses all have to support both efficient local and remote accesses of the data. The main challenges with remote access is that unlike local access, one cannot rely on relatively fine-grained random access to locations inside a file to be efficient, or even supported. For example, many cloud-based object storages only support reading whole files and do not allow querying data chunks within a file. A data layout that divides the data into smaller chunks faces the challenge of choosing a single chunk size that works well for both cloud and local accesses, which may not exist. Cloud access also prevents the layout from solely relying on pointers (or offsets) to locate the data chunks, as some form of address must be used instead. For local access, offsets may still be needed in addition to addresses, however, if a variable-rate compression scheme is used, since the chunks may not all have the same size. Finally, unlike local access, one must also take into account the nonnegligible cost of network latency: it is desirable to not only reduce the request size in bytes, but also to reduce the number of requests.

- **On-demand access.** Most compressors for scientific data require the data to be fully decompressed before use. This severely limits the usability of the compressed data, since the decompression process is often slow and the decompressed data often does not fit in memory. In practice, this means that compression is used purely as a storage-saving solution and not much else. Ideally, decompression should only happen just before the user accesses a particular chunk of data, and should not be run for the whole dataset prior to usage. From the user's perspective, this on-demand decompression should happen transparently in the background without explicit triggers from the user. For performance reason, this often means an in-memory cache is used to store recently decompressed data in case it is accessed again in the future so that the decompression cost is amortized over many accesses. The challenge is, then, how to design this cache so that memory usage and performance are balanced.

1.5 Contributions

To address the above challenges, this thesis makes several technical contributions.

1.5.1 Hierarchical model unifying resolution and precision

Working individually with either precision-based reduction or resolution-based reduction limits the achievable computational gains by maintaining either all bits for a few values or a few bits for all values. I show that both, previously separated, hierarchies can be combined into a single more general hierarchy, called a *precision-resolution tree* — a model that unifies the representation of spatial resolution hierarchies with precision-based data quantization approaches. In this model, adaptive approximations of data are modeled with the classical notion of a *valid cut* of the precision-resolution tree. Making both dimensions of possible data reduction available in a single, unified model lifts the proposed approach out of the 1D reduction spaces, where most existing techniques operate (see Figure 1.3), thus ushering in new opportunities for different mixes of resolution and precision, which have been shown [114] to benefit different types of analysis tasks. In addition, I also analyze the trade-offs associated with different choices in the degrees of freedom of this family of layouts and show how one can reproduce classical encoding schemes or design new ones with the advantage of being able to compare all within the same framework.

1.5.2 Compact encodings and layouts for progressive queries

To realize a precision-resolution requires a compact encoding scheme and a data layout that can efficiently serve data queries needed to perform scientific analysis. Since different analysis tasks need different data subsets, the design of the encoding and layout must be flexible and cannot be optimized purely for compression performance alone. Toward these goals, I formalize a complete family of parameters for practical data layouts that encode such trees in the presence of compression and other practical considerations. My proposed encoding and layout allows the flexibility of arbitrary incremental retrieval of *"chunks"* of data, progressively improving the resolution and/or precision of data. Furthermore, such data chunks can be retrieved without excessive time overhead (e.g., due to complex decoding) or data overhead, such as re-reading of data (e.g., due to a lack of progressivity) and/or reading or decoding of unused data (e.g., due to a lack of random access). I propose not a single data layout, but a new, highly flexible framework that can be leveraged to design novel, mixed-reduction strategies. I also provide an empirical study, including system-level considerations, that allows designing a new encoding scheme for scientific data, which achieves competitive speed, memory usage, and compression rates. These are achieved through choices of the framework's parameters that lean toward high degrees of progressivity and random access in all three domains — *precision, resolution, and space*

— while achieving performance comparable to that of state-of-the-art pure compression techniques. My scheme allows decoding the data progressively while following different precision-resolution trade-off curves decided at read time, per the needs of the application (Figure 1.4).

1.5.3 Optimal progressive streams for data analyses

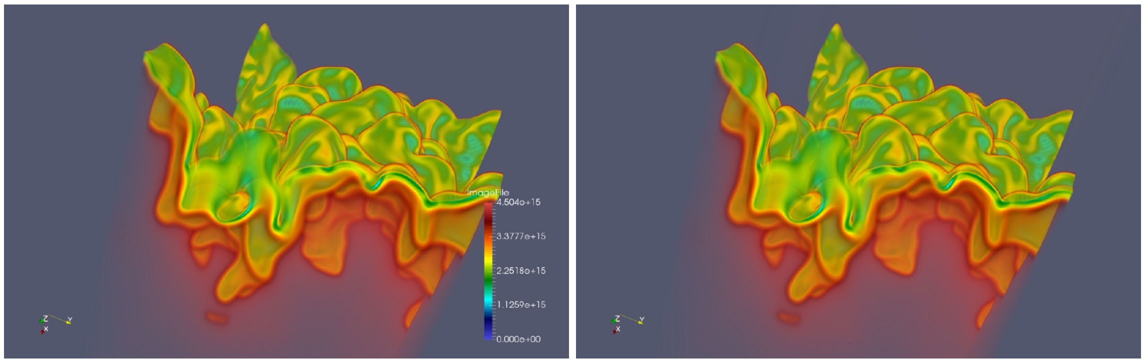
In general, different levels of adaptivity in combinations of resolution and precision may be suitable for different types of analysis and visualization tasks, and for many, these requirements might be data dependent. Consequently, a globally optimal data organization may not exist. Instead, I consider a progressive setting in which some initial data is loaded and processed, and new data is requested selectively based on the requirements of the current task as well as the characteristics of the data already loaded. The result is a stream of bits ordered such that the error is minimized, considering the task at hand. However, although intuitively considering both resolution and precision in the ordering certainly has advantages, it is unclear how much the error could be reduced for a given data budget or how little data could be used to achieve the same error. Furthermore, optimal data-dependent orderings may be especially impractical since they assume perfect knowledge of the data. It is therefore important to understand which of these potential gains are realizable, and address these problems about the suitable bit orderings through extensive, empirical experiments.

To address these issues, I introduce a framework that allows systematic studies of the resolution-versus-precision trade-off for common data analysis and visualization tasks. The core idea is to represent various data reduction techniques as bit streams that progressively improve data quality in either resolution or precision (Section 6.1). I can thus compare these techniques fairly, by comparing the corresponding bit streams (see Figure 1.5). I also provide empirical evidence that jointly optimizing resolution and precision can provide significant improvements on the results of analysis tasks over adjusting either independently. I present a diverse collection of data sets and data analysis tasks and also show how different types of data analysis might require substantially different data streams for optimal results. Finally, I present a greedy approach that gives estimations for lower bounds of error for various analysis tasks. In addition, I also identify practical streams that closely approximate these bounds for each task (Subsection 6.3.1, Subsection 6.3.2, Subsection 6.3.5, and Subsection 6.3.6) using a novel concept called *stream signature* (Subsection 6.2.2), which is a small matrix that captures the essence of how a bit stream navigates the precision-versus-resolution space.

1.5.4 Efficient tree-based progressive compression of particles

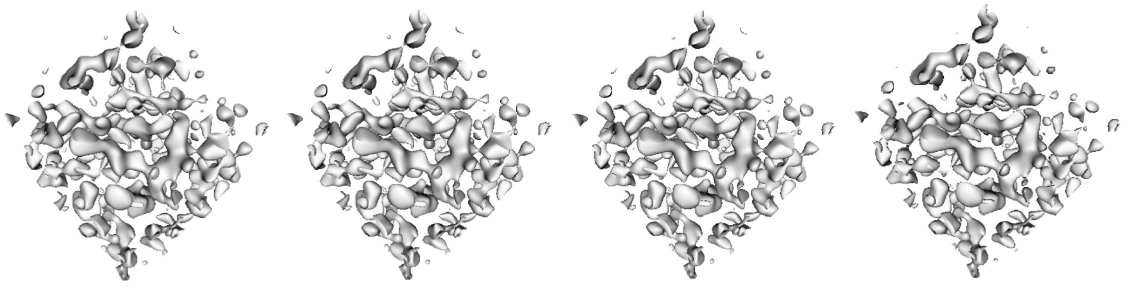
Beside regular grids, a common discrete representation is particles, which are moving points in space that carry attributes. Particles are frequently used in scientific applications, including molecular dynamics [101, 156, 198], fluid dynamics [5, 265, 322], computational cosmology [85, 108, 253], imaging of objects and environments [4, 142, 187], and plasma physics [30]. In a progressive setting, reconstruction quality depends greatly on the order in which the particle position bits are decoded, which also affects the costs of keeping a state in memory for resuming the decompression. Achieving a balance between decoding costs and reconstruction quality often manifests as a choice between (1) spatially limited but complete representation of particles and (2) quantized but uniform coverage of space — or, in a way, between a *depth-first* (DT) and a *breadth-first* traversal (BT) of a particle hierarchy. I explore this trade-off from the perspectives of both tree traversal and tree construction.

At the center of these contributions is a node splitting scheme called *odd-even split*, which I utilize to construct novel hierarchies that can be traversed with asymptotically constant memory footprints to produce high-quality progressive approximations (see Figure 1.6). The *odd-even split* (Section 7.1.1.2), which can be used in conjunction with the standard k-d splits (*i.e.*, splits that create a k-d tree) to selectively convert a DT of a subtree in to BT of the corresponding space. The odd-even splits can be combined with the traditional k-d splits to create *hybrid trees* (Section 7.1.1.3) that allows a low-memory-footprint DT to also have the power of BT (high-quality reconstruction), while being conducive to compression. A novel *adaptive traversal* scheme (AT) (Subsection 7.2.1) can be used to traverse particle trees that allows dynamic guiding of tree refinement, with respect to a given error metric; I propose two such metrics by heuristics. In addition, I introduce *block-hybrid trees* (Subsection 7.1.2), which combine the strengths of both k-d trees and hybrid trees, to be traversed with *block-adaptive traversal* (BAT, Subsection 7.2.2), for improved memory-quality trade-off and error-guided, progressive refinement with random access. For tree node encoding, I propose a *binomial coding* scheme (Subsection 7.3.1) that improves the compression of uniformly distributed particles by modeling the distribution of child node values using the Binomial distribution. Finally, I present an *odd-even context coding* scheme (Subsection 7.3.2) that improves the compression of dense surface data by leveraging the similarity between the two subtrees under an odd-even split.



(a) 64 bits **(b)** 12 bits

Figure 1.1: Volume renderings of a combustion simulation $512 \times 512 \times 256$ dataset, using (a) 64 bits and (b) 12 bits of precision. There are no appreciable differences between the two pictures.



(a) Full resolution, 512^3 **(b)** Resolution 256^3 **(c)** Resolution 128^3 **(d)** Resolution 64^3

Figure 1.2: Isocontours extracted from a combustion dataset at different sampling rates. The information loss is minimal even at the lowest sampling rate (64^3 in d).

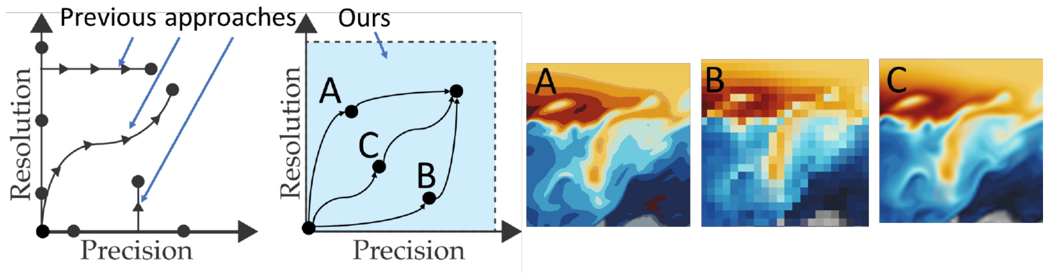


Figure 1.3: Considering precision and resolution as two axes in the space of data reductions. (Left) Previous approaches operate either as fixed points or fixed progression curves in this space. (Middle) The proposed unified tree model supports multiple arbitrary progression curves simultaneously with one data layout, effectively covering the entire space. (Right) Reducing too much precision causes banding (A). Reducing too much resolution results in pixelization (B). This work explores how flexible combinations find better quality at comparable sizes (C).

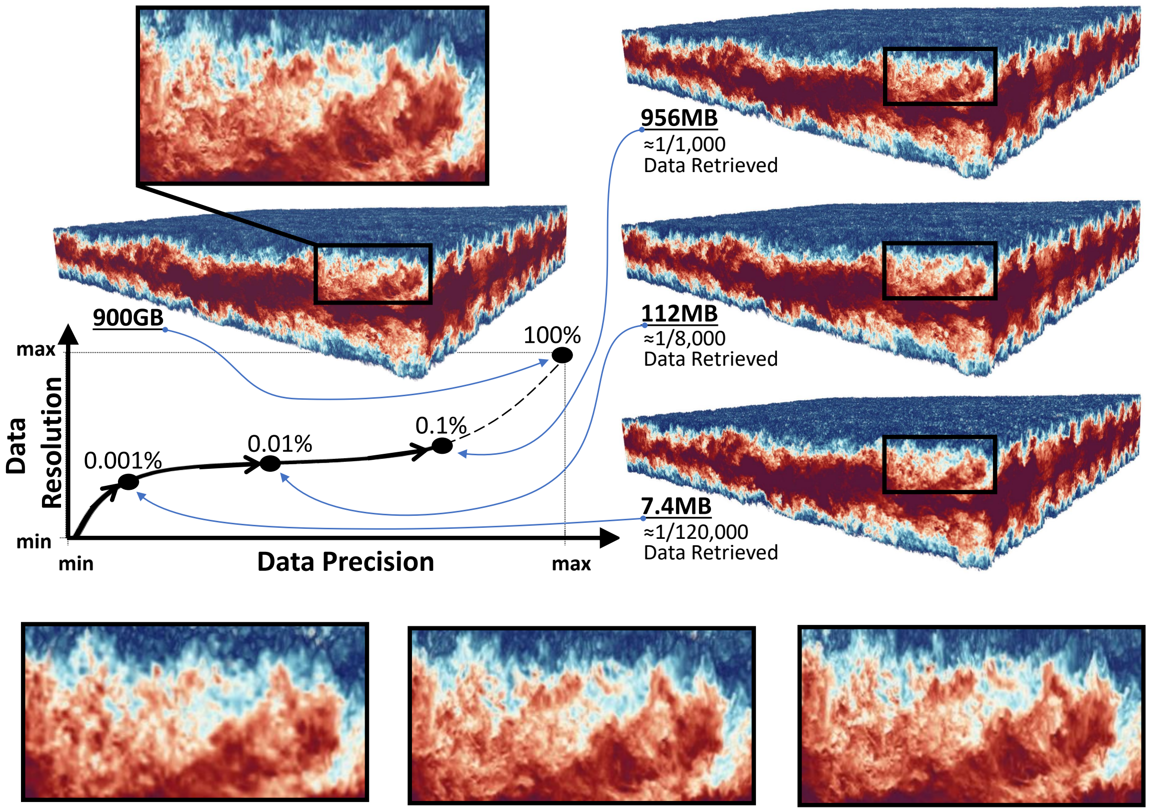


Figure 1.4: In the proposed encoding, each progressive decoding traces a monotonic nondecreasing curve in the precision-resolution space from the origin, 0%, to the full data, 100%. The time to decode the data and RAM used are shown; data retrieved values are inclusive of the preceding points along the curve. Decoding uses one core on a 4-core laptop CPU (2.8 GHz Intel Core i7-7700HQ) and a 122 MB/s consumer hard drive.

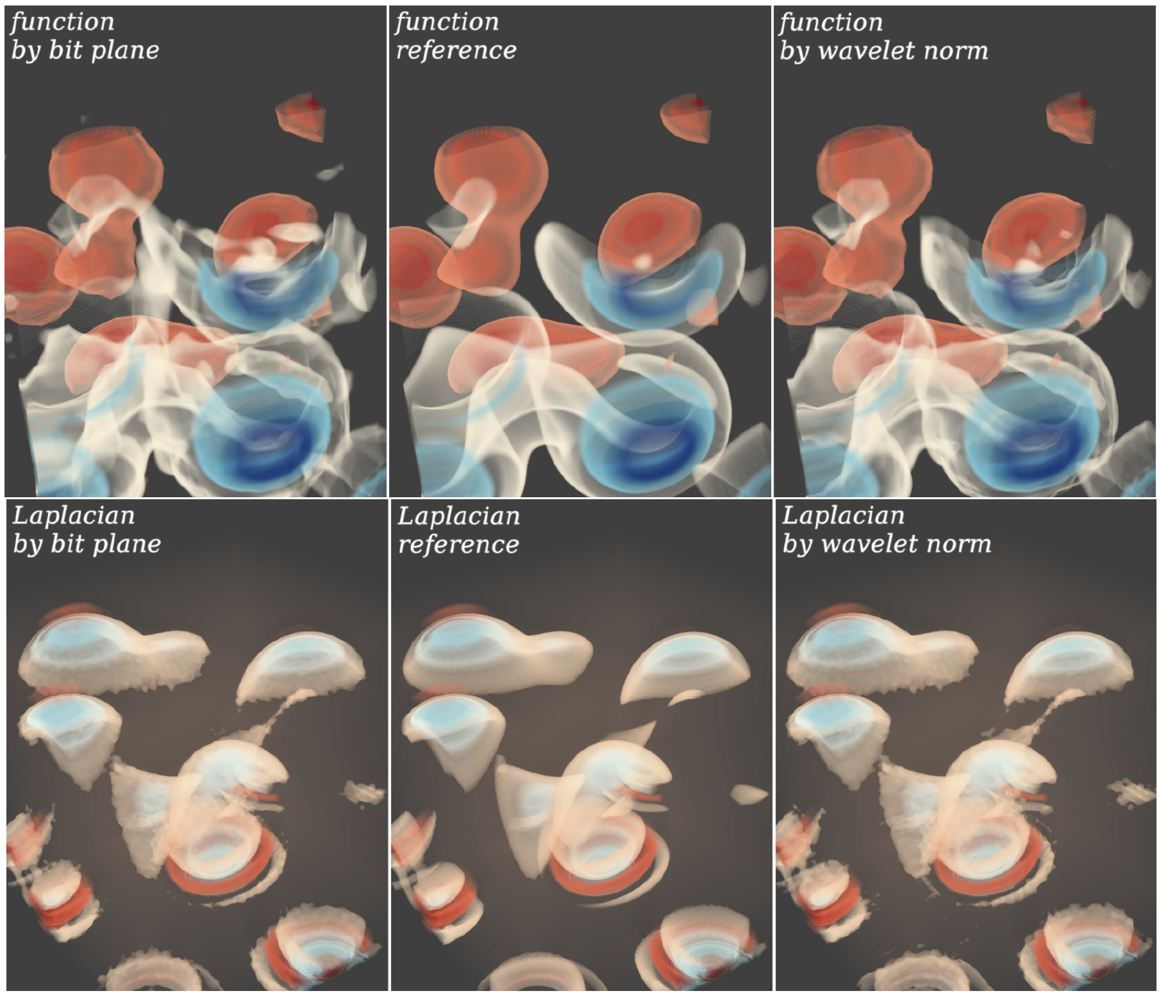


Figure 1.5: Visualization of the *diffusivity* field at 0.2 bits per sample (bps) and its Laplacian field at 1.5 bps, using two of the bit streams studied in this work. Compared to the *by bit plane* stream, the *by wavelet norm* stream produces a better reconstruction of the original function (left, compare white features), and a slightly worse, if not comparable, reconstruction of the Laplacian field (right).

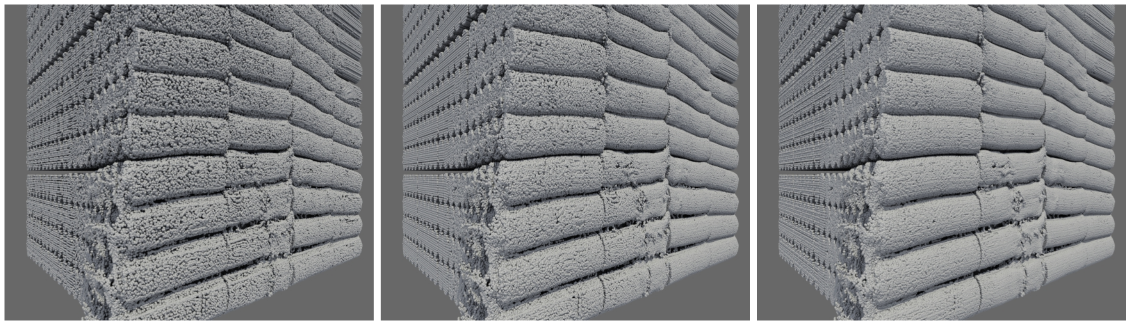

(a) 138×, n = 33M **(b)** 46×, n = 106M **(c)** 23×, n = 215M

Figure 1.6: Three approximations of a *detonation-large* simulation dataset, compressed and then decoded with my block-hybrid tree approach (compression ratio k× and corresponding number of particles, n, given). All three are snapshots of a single progressive decompression process, and all use only 50 MB for decoding.

CHAPTER 2

BACKGROUND

Here, I review the background needed to formally study and quantitatively analyze (lossy) data compression techniques.

2.1 Data compression theory

This section gives the necessary background on *information theory* [260], the theoretical foundation for data compression. We consider the data to compress to be a sequence of *symbols*) $\boldsymbol{s} = \{s_i\}$, represented in some standard binary format. Compression, or *source coding*, is the process of mapping $\boldsymbol{s}$ to a compressed binary sequence $\boldsymbol{b}$, which hopefully requires less storage than $\boldsymbol{s}$. When the original data is needed, $\boldsymbol{b}$ is *decompressed* to obtain a reconstructed sequence $\boldsymbol{s}'$. The compression/decompression process is considered *lossless* if $\boldsymbol{s}$ and $\boldsymbol{s}'$ are identical, otherwise it is considered *lossy*. Our discussion will focus on lossy compression, where the main problem is to minimize the difference between the decompressed sequence $\boldsymbol{s}'$ and the original sequence $\boldsymbol{s}$, often under constraints on the size of the compressed binary sequence $\boldsymbol{b}$.

2.1.1 Information theory

Information theory builds upon probability theory to establish theoretical bounds on the compression of the source sequence $\boldsymbol{s}$. In particular, we model the input sequence of symbols $\boldsymbol{s}$ as being generated by a *random process* (or a *source*) $\boldsymbol{S}$. We use $p_i(x)$ to denote the *probability mass function* (pmf) of S_i. In addition, $p_{i,N}(\boldsymbol{x}_{i,N})$ refers to the joint pmf of N successive samples $\{S_{i+1}, S_{i+2}, \ldots, S_{i+N}\}$. Similarly, $p_{i,N+1|N}(\boldsymbol{x}_{i,N+1|N})$ refers to the *conditional pmf* of S_{i+N+1} given its previous N samples. As is customary in the literature, we only consider *stationary* random processes, where $p_{i,N}(\boldsymbol{x}_{i,N})$ does not depend on i. Thus, every S_i is now just S (all samples are identically distributed), $p_{i,N}(\boldsymbol{x}_{i,N})$ becomes $p_N(\boldsymbol{x}_N)$, and $p_{i,N+1|N}(\boldsymbol{x}_{i,N+1|N})$ becomes $p_{N+1|N}(\boldsymbol{x}_{N+1|N})$. We sometimes use even simpler notations, *i.e.*,

$p(\boldsymbol{x}_N)$ and $p(\boldsymbol{x}_{N+1|N})$, if there is no ambiguity. If, in addition to being identically distributed, the samples $\{S_i\}$ are also independent, then $\boldsymbol{S}$ is called an *independent and identically distributed* (i.i.d), or *memoryless*, source.

2.1.1.1 Entropy

A *code* is a function $\Gamma(\boldsymbol{s})$ that maps an input sequence $\boldsymbol{s}$ to an output binary sequence $\boldsymbol{b}$. We want to be able to *decode* $\boldsymbol{b}$ to get back $\boldsymbol{s}$ (*i.e.,* decompression). Therefore, Γ must be a *uniquely decodeable* code, that is, no two different inputs produce the same output. Assuming that Γ assigns a code γ_k of length $|\gamma_k|$ bits to each letter a_k in the alphabet $\mathcal{A}$, a necessary condition for uniquely decodeable codes is the Kraft-McMillan inequality [140, 193]:

$$1 \geq K = \sum_{a_k \in \mathcal{A}} 2^{-|\gamma_k|} \tag{2.1}$$

The *bit rate* (or expected code length in bits per symbol) for this code is

$$E(|\gamma_k|) = \sum_{a_k \in \mathcal{A}} P(S = a_k)|\gamma_k| \tag{2.2}$$

Using K in Equation 2.1, we can rewrite $E(|\gamma_k|)$ as $-\log_2 K + A + H$, where A is non-negative and $H = -\sum P(S = a_k) \log_2 P(S = a_k)$ bits per symbol. Since $K \leq 1$ and $A \geq 0$, $E(|\gamma_k|)$ is bounded below by H, which is called the (first-order) *entropy* of the stationary random process $\boldsymbol{S}$ (or the entropy of the random variable S). The entropy $H_1(\boldsymbol{S})$ is the lower bound on the achievable bit rate for any code for $\boldsymbol{S}$ that codes each symbol independently of other symbols. If $p(x)$ denotes the pmf of S, *i.e.,* $p(a_k) = P(S = a_k)$, then

$$H_1(\boldsymbol{S}) = H(S) = -\sum_{x \in \mathcal{A}} p(x) \log_2 p(x) \tag{2.3}$$

Entropy is always non-negative, and measures the uncertainty about S; among all random variables with the same alphabet size, L, the uniform distribution ($p(x) = 1/L$) has the maximum entropy $H = \log_2 L$.

We can also define the *joint entropy* and *conditional entropy* of two discrete variables S, T with alphabets $\mathcal{A}_S, \mathcal{A}_T$ and joint pmf $p(x, y)$:

$$\begin{aligned} H(S,T) &= -\sum_{x \in \mathcal{A}_S, y \in \mathcal{A}_T} p(x,y) \log_2 p(x,y) \text{ , and} \\ H(S|T) &= \sum_{y \in \mathcal{A}_T} p(y) H(S|y) = -\sum_{y \in \mathcal{A}_T} p(y) \sum_{x \in \mathcal{A}_S} p(x|y) \log_2 p(x|y) \end{aligned} \tag{2.4}$$

The joint entropy and conditional entropy can be used to define the *N-th order entropy* and *N-th order conditional entropy* of a stationary discrete source:

$$\begin{aligned} H_N(\boldsymbol{S}) &= H(\boldsymbol{S}_N) = -\sum_{\boldsymbol{x}_N \in \mathcal{A}^N} p(\boldsymbol{x}_N) \log_2 p(\boldsymbol{x}_N) \text{ , and} \\ H_{c,N}(\boldsymbol{S}) &= H(S_{N+1}|\boldsymbol{S}_N) = \sum_{\boldsymbol{x}_N \in \mathcal{A}^N} p(\boldsymbol{x}_N) H(S_{N+1}|\boldsymbol{x}_N), \end{aligned} \tag{2.5}$$

where $\boldsymbol{S}_N$ stands for any N successive samples in $\boldsymbol{S}$, with joint pmf $p(\boldsymbol{x}_N)$.

Both $\frac{1}{N}H_N(\boldsymbol{S})$ and $H_{c,N}(\boldsymbol{S})$ are non-increasing functions of N, and both converge to the same limit as N tends to infinity. We define the *entropy rate* of $\boldsymbol{S}$ to be this limit:

$$\overline{H}(\boldsymbol{S}) = \lim_{N\to\infty} \frac{1}{N} H_N(\boldsymbol{S}) = \lim_{N\to\infty} H_{c,N}(\boldsymbol{S}) \tag{2.6}$$

It can be shown that

$$\begin{aligned} H_N(\boldsymbol{S}) &= H_1(\boldsymbol{S}) + \sum_{n=1}^{N-1} H_{c,n}(\boldsymbol{S}) \text{ , and} \\ \overline{H}(\boldsymbol{S}) \le H_{c,N-1}(\boldsymbol{S}) &\le \frac{1}{N} H_N(\boldsymbol{S}) \le H_1(\boldsymbol{S}), \end{aligned} \tag{2.7}$$

with equalities happening when $\boldsymbol{S}$ is i.i.d. In other words, $\overline{H}(\boldsymbol{S})$ provides the lower bound for the bit rate when coding $\boldsymbol{S}$ losslessly, which can be achieved only when infinitely many symbols are coded together ($N \to \infty$). Furthermore, for non i.i.d sources, conditioning on previous symbols and coding many symbols together are better than coding symbols individually.

2.1.1.2 Mutual information

Given two discrete random variables S and T, the "intersection" between $H(S)$ and $H(T)$ is called the *mutual information* between S and T, formally defined as

$$I(S,T) = H(S) - H(S|T) = H(T) - H(T|S) = \sum_{x \in \mathcal{A}_S, y \in \mathcal{A}_T} p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)} \tag{2.8}$$

$I(S,T)$ gives the reduction in the uncertainty of S given the knowledge of T and vice versa (note that when $S = T$, $I(S,S) = H(S)$). Several relations between entropy, joint entropy, conditional entropy, and mutual information exist; I list some of them below (they can all be remembered by thinking of $I(S,T)$ as the intersection of $H(S)$ and $H(T)$, and $H(S,T)$ as their union, as if they were sets).

$$\begin{gathered}
H(S,T) = H(S) + H(S|T) = H(T) + H(T|S) \\
I(S,T) = H(S) - H(S|T) = H(T) - H(T|S) \\
I(S,T) = H(S) + H(T) - H(S,T) \qquad (2.9) \\
I(S,T) \leq H(S) \text{ and } I(S,T) \leq H(T) \\
H(S) \geq H(S|T) \text{ and } H(T) \geq H(T|S)
\end{gathered}$$

The last relations say that conditioning reduces entropy. As before, the *N-th order mutual information* can be defined between two discrete stationary sources:

$$I_N(\boldsymbol{S},\boldsymbol{T}) = \sum_{\boldsymbol{x}_N \in \mathcal{A}_S^N, \boldsymbol{y}_N \in \mathcal{A}_T^N} \frac{p(\boldsymbol{x}_N, \boldsymbol{y}_N)}{p(\boldsymbol{x}_N)p(\boldsymbol{y}_N)} \qquad (2.10)$$

, where $p(\boldsymbol{x}_N, \boldsymbol{y}_N)$ denotes the joint pmf of N successive samples in $\boldsymbol{S}$ and N successive samples in $\boldsymbol{T}$. I will use $I_N(\boldsymbol{S},\boldsymbol{T})$ later when discussing the lower bound on the minimum bit rate required for achieving a desired distortion between the original source ($\boldsymbol{S}$) and its quantized version ($\boldsymbol{T}$).

2.1.1.3 Bit rate bounds for lossless coding

Consider coding a discrete stationary source $\boldsymbol{S}$ with alphabet $\mathcal{A}$ using a code γ that assigns a binary sequence γ_i to each symbol $\{a_i\}$ of $\mathcal{A}$ individually. Let

$$R_1(\boldsymbol{S},\gamma) = \sum_{a_i \in \mathcal{A}} p(a_i)|\gamma_i| \qquad (2.11)$$

denote the *bit rate* (average number of bits per symbol) of this code (the subscript $_1$ indicates that the symbols are coded one at a time). We want to understand how low this bit rate can be as a function of the code γ. Let $\overline{R}_1(\boldsymbol{S}) = \min_\gamma R_1(\boldsymbol{S},\gamma)$. We can bound this minimum rate using the (first-order) entropy of $\boldsymbol{S}$ as

$$H_1(\boldsymbol{S}) \leq \overline{R}_1(\boldsymbol{S}) \leq H_1(\boldsymbol{S}) + 1 \qquad (2.12)$$

The lower bound is achieved if, for all $a_i \in \mathcal{A}$, $p(a_i) = 2^{-|\gamma_i|}$ (*i.e.*, none of the $|\gamma_i|$ bits assigned to code a_i is "wasted").

To reduce the bit rate further, we can treat each group of N samples as a vector, and assign one codeword for each vector. Applying the same bound above to the vector source and dividing by N (to obtain the average bit length per symbol), we have

$$\frac{H_N(\boldsymbol{S})}{N} \leq \overline{R}_N(\boldsymbol{S}) \leq \frac{H_N(\boldsymbol{S})}{N} + \frac{1}{N} \tag{2.13}$$

In the limit, $\lim_{N\to\infty} \overline{R}_N(\boldsymbol{S}) = \lim_{N\to\infty} \frac{H_N(\boldsymbol{S})}{N} = \overline{H}(\boldsymbol{S})$. This means that as N increases, the minimum bit rate can get arbitrarily close to the entropy rate of the source. For conditional lossless coding, we have similar bounds:

$$\overline{H}(\boldsymbol{S}) \leq \lim_{N\to\infty} \overline{R}_{c,N}(\boldsymbol{S}) \leq \overline{H}(\boldsymbol{S}) + 1 \tag{2.14}$$

Compared to the bounds for vector coding, the minimum bit rate for conditioning on previous symbols has the same asymptotic lower bound that is the entropy rate However, the upper bound does not converge to the same entropy rate, but is always 1 bit larger due to the fact that each symbol is still being coded individually.

2.1.1.4 Bit rate bounds for lossy coding

In lossy coding, the reconstructed sequence is not the original sequence $\boldsymbol{s}$ itself, but some approximation $\boldsymbol{s}'$. To quantify the approximation error, we need to define a distance function $d(\boldsymbol{s}, \boldsymbol{s}')$ which tells us how much $\boldsymbol{s}'$ differs from $\boldsymbol{s}$. Let $d(s_i, s'_i)$ denote a (non-negative) distance between two real values s_i and s'_i (common examples for d are the absolute error $|s_i - s'_i|$ and the squared error $(s_i - s'_i)^2$). We define the *distortion* between the original vector $\boldsymbol{s}$ and its reconstruction $\boldsymbol{s}'$ as:

$$d(\boldsymbol{s}, \boldsymbol{s}') = \frac{1}{|\boldsymbol{s}|} \sum_{i=1}^{|\boldsymbol{s}|} d(s_i, s'_i) \tag{2.15}$$

Like with $\boldsymbol{s}$, we model $\boldsymbol{s}'$ as an instance of a random source $\boldsymbol{S}'$ with alphabet $\mathcal{A}'$. To define a distortion between two sources $\boldsymbol{S}$ and $\boldsymbol{S}'$, we average over all possible vectors $\boldsymbol{x}_N$ and $\boldsymbol{x}'_N$ of a fixed length N:

$$d_N(\boldsymbol{S}, \boldsymbol{S}') = \sum_{\boldsymbol{x}_N \in \mathcal{A}^N} \sum_{\boldsymbol{x}'_N \in \mathcal{A}'^N} p(\boldsymbol{x}_N, \boldsymbol{x}'_N) d(\boldsymbol{x}_N, \boldsymbol{x}'_N) \tag{2.16}$$

Unlike in lossless coding where only the rate is of concern, in lossy coding, there is a trade-off between the rate and the distortion: the higher the rate, the lower the distortion and vice versa. Consider a random process $\boldsymbol{S}'$ such that $d_N(\boldsymbol{S}, \boldsymbol{S}') \leq D$ for some given distortion threshold D. Any code for $\boldsymbol{S}'$ needs to have a rate that is greater than or equal to the entropy rate $H(\boldsymbol{S}') = \lim_{N\to\infty} \frac{H_N(\boldsymbol{S})}{N}$, which itself is bounded below by $\lim_{N\to\infty} \frac{I_N(\boldsymbol{S},\boldsymbol{S}')}{N}$. Therefore, the minimum rate R required to achieve a distortion no larger than D is bounded below by the same quantity:

$$R(D) = \lim_{N\to\infty} \inf_{d_N(\boldsymbol{S},\boldsymbol{S}')\le D} I_N(\boldsymbol{S},\boldsymbol{S}') \tag{2.17}$$

$R(D)$ is called the *rate-distortion function*; it is continuous, monotonically decreasing for $R > 0$, and convex. It may seem surprising that we are minimizing the mutual information between $\boldsymbol{S}$ and $\boldsymbol{S}'$ (instead of maximizing this quantity so as to minimize the error between $\boldsymbol{S}$ and $\boldsymbol{S}'$). However, this result should be understood in the sense that, among all the choices of $\boldsymbol{S}'$ with distortion at most D, the one having the least mutual information with $\boldsymbol{S}$ requires the lowest bit rate.

Replacing $I_N(\boldsymbol{S},\boldsymbol{S}')$ with $H_N(\boldsymbol{S}) - H_N(\boldsymbol{S}|\boldsymbol{S}') = H_N(\boldsymbol{S}) - H_N(\boldsymbol{S}-\boldsymbol{S}'|\boldsymbol{S}')$, we obtain the *Shannon's lower bound*:

$$R(D) \ge R_L(D) = \overline{H}(\boldsymbol{S}) - \lim_{N\to\infty} \sup_{d_N(\boldsymbol{S},\boldsymbol{S}')\le D} H_N(\boldsymbol{S}-\boldsymbol{S}'|\boldsymbol{S}') \tag{2.18}$$

This result implies that for an ideal code, the error process $\boldsymbol{S}-\boldsymbol{S}'$ should be statistically independent from the reconstructed source $\boldsymbol{S}'$. It is not possible to determine the Shannon lower bound analytically for most sources and distortion measures. However, for stationary sources and the mean-square-error (MSE) distortion, the distortion $d_N(\boldsymbol{S},\boldsymbol{S}')$ is equal to the variance of the error process $\boldsymbol{E} = \boldsymbol{S}-\boldsymbol{S}'$, and it can be shown that the maximum N-th order (differential) entropy for $\boldsymbol{E}$ with variance σ^2 is $\frac{N}{2}\log_2 2\pi e\sigma^2$. Hence, the Shannon lower bound for stationary processes and MSE distortion is:

$$\begin{gathered} R_L(D) = \overline{H}(S) - \frac{1}{2}\log_2 2\pi e D, \text{ or, equivalently,} \\ D_L(R) = \frac{1}{2\pi e} 2^{2\overline{H}(S)} 2^{-2R} \end{gathered} \tag{2.19}$$

Among sources with the same variance, a Gaussian source requires the highest bit rate (equal to $R_L(D)$) to satisfy the same distortion criterion. For any source, the optimal code that achieves the $R(D)$ bound is one that produces i.i.d. Gaussian errors with variance D. The (differential) entropy of the error source in that case is $\frac{1}{2}\log_2 2\pi e D$, and the lower bound $R_L(D)$ becomes the difference between the entropy of the source and that of the error. If

$\boldsymbol{S}$ itself is an i.i.d. Gaussian process with variance σ^2, the rate-distortion function can be found analytically as

$$R(D) = \frac{1}{2}\log_2 \frac{\sigma^2}{D} \text{ , or, alternatively, } D(R) = \sigma^2 2^{-2R} \tag{2.20}$$

2.1.2 Quantization

Quantization is the most common way to realize lossy coding, where a function Q (the *quantizer*) maps the N-dimensional Euclidean space into a finite set of *reconstruction values* inside the N-dimensional Euclidean space $Q : \mathcal{R}^N \rightarrow \{\boldsymbol{s}'_0, \boldsymbol{s}'_1, \ldots, \boldsymbol{s}'_{K-1}\}$. If the dimension N is equal to 1, Q is a *scalar quantizer*, otherwise it is a *vector quantizer*. K is also called the size of the quantizer Q. Q partitions $\mathcal{R}^N$ into *quantization cells* C_i $(0 \leq i < K)$ such that each cell is associated with a reconstruction vector: $C_i = \{\boldsymbol{s} \in \mathcal{R}^N : Q(\boldsymbol{s}) = \boldsymbol{s}'_i\}$. A quantizer is completely defined by the set of reconstruction values and the associated quantization cells.

We consider the quantization of a sequence of input vectors $\{\boldsymbol{s}_i\}$ that is a realization of a vector random process $\{\boldsymbol{S}_i\}$. Each input vector $\boldsymbol{s}_i$ is mapped onto a reconstruction vector $\boldsymbol{s}'_i = Q(\boldsymbol{s}_i)$. If $\boldsymbol{S}_i$ is stationary, the expected distortion between the input and the output vectors depends only on $\boldsymbol{S}$ and Q:

$$D_Q(\boldsymbol{S}) = E[d_N(\boldsymbol{S}, Q(\boldsymbol{S}))] = \sum_i P(C_i) \int_{C_i} d_N(\boldsymbol{s}, Q(\boldsymbol{s}))\, p(\boldsymbol{s}|C_i) d\boldsymbol{s} \tag{2.21}$$

where $P(C_i)$ is the probability that $\boldsymbol{s}$ falls in cell C_i, and $p(\boldsymbol{s}|C_i)$ is the conditional pdf of $\boldsymbol{s}$ given that $\boldsymbol{s} \in C_i$.

2.1.2.1 Uniform scalar quantization

The simplest type of quantization is scalar quantization with uniform *step size*. That is, the cells C_i become half-open intervals of equal sizes $[b_i, b_{i+1})$ where $\Delta = b_{i+1} - b_i$ is the step size. The interval endpoints $\{b_i\}$ are also called the *decision boundaries*. For general inputs, having a constant step size is not possible, since the finite set of quantization intervals must cover the whole real line from $-\infty$ to ∞. However, many real-world input signals can be bound by an amplitude range $[s_{min}, s_{max}]$, for a dynamic range of $A = s_{max} - smin$. The dynamic range can then be partitioned into K quantization cells with step size

$$\Delta = \frac{A}{K} = 2^{-R} A \tag{2.22}$$

where $R = \log_2 K$ is the bit rate (the number of bits required to specify the quantization cells). A common quantizer of this type is called *Pulse-Code-Modulation* (PCM), where the reconstruction values are placed in the middle of the quantization cells, namely

$$Q(s) = \left\lfloor \frac{s - s_{min}}{\Delta} + \frac{1}{2} \right\rfloor \Delta + s_{min} \tag{2.23}$$

Assuming the source is uniformly distributed between $[s_{min}, s_{max}]$, and that the distortion metric is the mean-square error (MSE), we can rewrite the distortion in Equation 2.21, which is now also the variance of the quantization error (denoted as σ_Q^2), as

$$\sigma_Q^2 = D_Q(S) = E[(S - Q(S))^2] = \sum_i P(s \in C_i) \int_{b_i}^{b_{i+1}} (s - s_i')^2 p(s|C_i) ds \tag{2.24}$$

The above equation can be alytically computed to be

$$\sigma_Q^2 = D(R) = \sigma^2 2^{-2R}, \tag{2.25}$$

where $\sigma^2 = \frac{A^2}{12}$ is the variance of the original source. Equation 2.25 relates the variance of the quantization error to the variance of the original source. The signal-to-noise ratio (SNR) of the quantizer is defined as the logarithm of the ratio between the two variances:

$$SNR = 10 \log_{10} \frac{\sigma^2}{\sigma_Q^2} = (20 \log_{10} 2) R \approx 6.02R \tag{2.26}$$

Thus, every additional bit results in a 6.02 dB gain in SNR – a well-known result in quantization theory [211].

2.1.2.2 Optimal scalar quantization

From the definition of the distortion D (or equivalently signal-to-noise ratio SNR), it is clear that for a source that is not uniformly distributed, the uniform scalar quantizer produces suboptimal expected distortion. A better quantizer should be *adaptive*, that is, it should take into account the actual probability density of the input. Intuitively, we want the quantizer to allocate more output levels where the pdf $p(x)$ is more dense. One effective technique to study such nonuniform quantizer is to model it as a combination of a nonlinear function $F(x)$ (the "compressor"), followed by a uniform quantizer [18]. That is, the input signal is first transformed by $F(x)$, then uniformly quantized. Function F is monotonic, therefore is invertible, and F^{-1} is used on the decompression side. The distortion for the nonuniform quantizer is $D = \frac{V^2}{3N^2}\int_{-V}^{V} \frac{p(s)}{F'(s)^2} ds$ (V, known as the *overload*, specifies the range of the quantizer) [18]. Given an input probability density, this formula can be minimized over all compressor curves $F'(s)$ to give the result that the optimum compressor slope is proportional to the cube root of the pdf $p(s)$, i.e., the optimal compressor function is $F^*(s) = c\int_0^s (p(\alpha))^{1/3} d\alpha$ (c is a constant chosen so that $F(V) = V$).

Alternatively, to obtain an optimal quantizer at a given number of cells, we can minimize the distortion D as a function of both the decision boundaries $\{b_i\}$ and the reconstruction values $\{s'_i\}$. In Equation 2.24, setting both partial derivatives $\partial D/\partial b_i$ and $\partial D/\partial s'_i$ to 0 as in [190], we obtain

$$\begin{aligned} b_i &= \frac{s'_i + s'_{i+1}}{2} \\ s'_i = E(S|C_i) &= \int_{b_i}^{b^{i+1}} p(s|s \in [b_i, b_{i+1}))s ds \end{aligned} \tag{2.27}$$

These two conditions mean that the optimal decision boundaries $\{b_i\}$ are exactly between neighboring reconstruction values (*the nearest neighbor condition*), and simultaneously, the reconstruction values $\{s'_i\}$ are the centroids of their respective intervals (*the centroid condition*). If the distortion metric is not the MSE, the nearest neighbor condition still holds, but the notion of a centroid in the centroid condition depends on the distortion function. For example, if $d(s, s') = |s - s'|$, the centroid of each cell C_i is the median value in C_i. The MSE, however, also equalizes the quantization error among the quantization cells, among other important statistical properties [89,130].

The MSE quantizer for a uniform source is uniform (*i.e.,* the solution given by Equation 2.23), but for a nonuniform source, the best MSE quantizer cannot in general be determined by a close-form formula. In practice, the distribution of the input signal is usually unknown. For such cases, the quantizer can be designed based on a training set of representative input samples. A popular algorithm of this type is the *Lloyd-Max algorithm* [177,190], which iteratively updates both the reconstruction values and the decision boundaries. First, the reconstruction values are found using the centroid condition, then the decision boundaries are found by partitioning the training samples using the nearest neighbor condition. The same process repeats until convergence. For finite inputs, this algorithm is also known as the *k-means clustering algorithm*, independently discovered by Forgy [80].

2.1.2.3 Entropy-constrained optimal scalar quantization

The Lloyd-Max algorithm minimizes the distortion for a given number of quantization cells, assuming that every cell is coded with a codeword of constant length. If the quantization cells are encoded with variable-length codes, however, we need to optimize for the rate and the distortion together. Denote $\bar{l}(s'_i)$ as the average code length assigned to s'_i, then the average bit rate is given by $R = \sum p(s'_i)\bar{l}(s'i)$. We want to minimize R subject to $D \leq D_{max}$: This can be reformulated as an unconstrained minimization problem, using the *Lagrange multiplier* λ, by minimizing $J = D + \lambda R$:

$$J = E[d(S, Q(S))] + \lambda E[\bar{l}(Q(S))] \tag{2.28}$$

If there exist a λ for which J is minimized when $D = D_{max}$, then the corresponding R is a solution for the original minimization problem. If the decision boundaries $\{b_i\}$ are given, the optimal reconstruction values $\{s'_i\}$ are determined by minimizing the distortion D alone; therefore, they are given by the generalized centroid condition (Equation 2.27). It is reasonable to approximate the average rate R by the (first-order) entropy $H(S)$, and set the average code length $\bar{l}(s'_i)$ to $-\log_2 p(s'_i)$. Then, to find the optimal decision boundaries $\{b_i\}$, we assume $\{s'_i\}$ and $\{\bar{l}(s'_i)\}$ are given, and solve for $Q(s)$, using the augmented nearest-neighbor condition:

$$Q(s) = \arg\min_{s'_i} d(s, s'_i) + \lambda \bar{l}(s'_i) \tag{2.29}$$

It can be proven [262] that minimizing $d(s, s'_i) + \lambda \bar{l}(s'_i)$ for each source symbol s also minimizes J in Equation 2.28. The minimization problems to find $\{b_i\}$ and $\{s'_i\}$ can be solved in alternate fashion, until the solutions converge, similarly to how the Lloyd-Max algorithm works in the case of fixed-length codewords. Compared to Lloyd-Max, here there is an additional step to update the average codeword length $\bar{l}(s'i) = -\log_2 p(s'_i)$. Finally, to obtain a solution to the original constrained minimization problem, we perform a search for a λ for which $D \approx D_{max}$. Compared to optimal fixed-length quantizers, which put the decision boundaries in the exact middle of the reconstruction values, optimal variable-length quantizers position the decision boundaries so that low-probability reconstruction values are allocated smaller quantization cells.

2.1.2.4 Asymptotic distortion rate performance of quantizers

In general, for an arbitrary source distribution, it is impossible to obtain analytic rate-distortion functions for optimal quantizers. However, when the rate R is high enough (or, equivalently, the distortion D is low enough), we can obtain closed-form solutions for the distortion-rate function asymptotically (as R tends to infinity). The assumption is that when R is large, the quantization cells are small enough that we can assume the pdf of the source is uniform within each cell, making Equation 2.24 tractable. Below I list several results for different kinds of quantizer:

$$\begin{aligned} D_{pcm}(R) &= \frac{1}{12}A^2 2^{-2R} \\ D_{fixed}(R) &= \sigma^2\epsilon^2 2^{-2R} \text{ where } \epsilon = \frac{1}{\sigma^2}\int_{-\infty}^{\infty}\sqrt[3]{p(s)}ds \\ D_{variable}(R) &= \sigma^2\epsilon^2 2^{-2R} \text{ with } \epsilon^2 = \frac{2^{2H(S)}}{12\sigma^2} \ (H(S) \text{ is the differential entropy of } S) \end{aligned} \tag{2.30}$$

D_{fixed} and $D_{variable}$ are the optimal quantizers using fixed-length [218] and variable-length [90] codewords, respectively. All of the formulas, as well as the Shannon lower bound (Equation 2.19, here denoted as $D_S(R)$ or $R_S(D)$), can be written as $D(R) = \epsilon^2\sigma^2 2^{-2R}$, with different values for ϵ, which depends on the pdf of the source. In terms of SNR,

$$SNR(R) = 10\log_{10}\frac{\sigma^2}{D(R)} = -10\log_{10}\epsilon^2 + R\,20\log_{10}2 \tag{2.31}$$

So, for all high-rate approximations, including the Shannon lower bound, the SNR increases by $20\log_{10}2 \approx 6$ dB per additional bit stored. It is remarkable that the ratio $D_V(R)/D_S(R)$ is constant and is equal to $\pi e/6 \approx 1.53dB$. Correspondingly, the difference in rate is also constant: $R_V(D) - R_L(D) = \frac{1}{2}\log_2(\frac{\pi e}{6}) \approx 0.25$ (bits per sample). In other words, at high rates, the performance of an optimal scalar quantizer is fairly close to the Shannon lower bound, for a difference of 0.25 bits per sample at the same distortions, or an SNR difference of 1.53dB at the same rates.

2.1.2.5 Vector quantization

To close the gap between optimal scalar quantizers and the Shannon lower bound, multiple samples must be quantized together. In vector quantization (VQ) [99], we consider quantizing blocks of N consecutive samples generated by a stationary random process S. Most concepts carry over from scalar quantization. For optimal vector quantization with fixed-length codes, we simultaneously solve two minimization problems which state the necessary conditions for the reconstruction vectors and quantization cell boundaries.

$$\begin{aligned} \text{generalized centroid condition: } \boldsymbol{s}'_i &= \arg\min_{\boldsymbol{s}'} E[d_N(\boldsymbol{S},\boldsymbol{s}')|\boldsymbol{S} \in C_i] \\ \text{nearest neighbor condition: } Q(\boldsymbol{s}) &= \arg\min_{\boldsymbol{s}'_i} d_N(\boldsymbol{s},\boldsymbol{s}'_i) \end{aligned} \tag{2.32}$$

The extension of the Lloyd-Max algorithm to vector quantization is called the *Linde-Buzo-Gray* (LBG) algorithm [166].

For optimal vector quantization with variable-length codes, we again leverage Lagrange multiplier to turn a constrained to an unconstrained optimization problem. The two minimization problems to solve for are:

$$\begin{aligned} \boldsymbol{s}'_i &= \arg\min_{\boldsymbol{s}'} E[d_N(\boldsymbol{S},\boldsymbol{s}')|\boldsymbol{S} \in C_i] \\ Q(\boldsymbol{s}) &= \arg\min_{\boldsymbol{s}'_i} d_N(\boldsymbol{s},\boldsymbol{s}'_i) + \lambda \bar{l}(\boldsymbol{s}'_i) \end{aligned} \tag{2.33}$$

As before, in each iteration, the average codeword length $\bar{l}(\boldsymbol{s}'_i)$ is updated to $-\log_2 p(\boldsymbol{s}'_i)$. This algorithm is known as the *Chou-Lookabaugh-Gray* (CLG) algorithm [38].

Vector quantization (VQ) typically achieves better performances than scalar quantization (SQ) due to three main reasons: space-filling advantage, shape advantage, memory advantage [179]. The space-filling advantage refers to the fact that N-dimensional Cartesian products of intervals (obtained with SQ) are not the best way to partition N-dimensional space into non-overlapping regions so that quantization error is minimized. For example, in 2D, the best shape for “tiling” space is not a rectangle but a hexagon, which can only be obtained with VQ but not SQ. The shape advantage is that VQ can adapt to the pdf of the source by allocating smaller quantization cells to less probable reconstruction values without the need to use variable-length codes. Finally, the memory advantage – perhaps the most important advantage of VQ – is that VQ can exploit correlation among neighboring samples, which is ample in signals such as images and videos.

One major disadvantage of VQ in practice is that the encoder tends to be slow (while the decoder is much faster). This is because to find the correct cell for an input sample, in principle all cells must be examined, and for each cell, a distance involving N pairs of samples must be computed. The total number of operations then is on the order of $N2^{NR}$ with N being the block size of the quantizer. This is also the space complexity required to store the codebook. To reduce this complexity, VQ techniques in practice often impose some structure on the quantization cells, such as requiring them to form a tree or a grid (lattice). The book by Gersho and Gray [89] discusses many such designs.

Taken from [88], the uniform quantizer can achieve an SNR within 7 dB of the best theoretically attainable quantizer. If uniform quantization is followed by entropy coding (of the ouput values), this gap decreases to only 1.5 dB. Without entropy coding, an optimum nonuniform quantizer can be used for an additional 3 dB penalty in SNR. However, these results all assume that the input samples are statistically independent, which is often not true for a wide range of data (e.g., images or simulation volumes). For such data, much better (lossy) compression can be achieved with VQ. In vector quantization, groups of input samples are quantized together to better exploit their correlations. In fact, the Lloyd-Max or k-means algorithm previously mentioned is a form of vector quantization when applied to higher-dimesional data. Although vector quantization often compresses better than scalar quantization, the issue of generating an optimum codebook is non trivial. Algorithms such as k-means can be used, but in general this codebook generation process is slow, so vector quantization is only good for offline compression of data not too large. If scalar quantization is used for correlated input samples, for better compression ratios, the samples should be decorrelated with a linear transform or prediction before quantization.

2.2 Data compression practice

The most common type of scientific data is a d-dimensional scalar field $f : \mathbb{R}^d \rightarrow \mathbb{R}$, sampled on a finite regular grid of sample points. The sample points together form a sequence of real numbers (or more generally, *symbols*) $\boldsymbol{s} = \{s_i\}$. The discrete set of all possible values that each symbol s_i can take on is called an *alphabet* $\mathcal{A}$. Data compression is the process of mapping $\boldsymbol{s}$ to a sequence of bits $\boldsymbol{b}$ such that the length of $\boldsymbol{b}$ (in bits) is less than the length of $\boldsymbol{s}$ in its more "natural" encoding (*e.g.*, storing each sample value in the IEEE754 floating-point format).

It is useful to think of data compression in practice as a two-step process. In the first step, often called *modeling*, $\boldsymbol{s}$ is mapped to an "intermediate" sequence of symbols $\boldsymbol{\beta} = \{\beta_j\}$ (the mapping is not necessarily one-to-one, and can in fact also be the identity map). Each possible value for β_j is called a *codeword* and the set of all possible codewords is called a *codebook*. In the second step, $\boldsymbol{\beta}$ is mapped to the output bit sequence $\boldsymbol{b}$, by assigning a *code*

γ_k (a sequence of bits) to each symbol (or a group of symbols) in $\boldsymbol{\beta}$. From this perspective, compression can happen in two ways. First, we can reduce the number of elements in the codebook so that fewer bits are needed to represent each β_j. Such an approach is often called *quantization*.Alternatively, or in addition, we can carefully assigning binary codes $\{\gamma_k\}$ to the codewords $\{\beta_j\}$ so that the output $\boldsymbol{b}$ is shorter on average (over all possible input sequences). This approach is often called *entropy coding*. Modeling, quantization, and entropy coding are the main building blocks of most practical compression algorithms.

2.2.1 Modeling

2.2.1.1 Linear prediction

Linear prediction assumes an order that the data samples are visited, and predicts the value of the current sample f_i using a linear combination that of k previous samples $f_i \approx \hat{f}_i = \sum_{j=1}^{k} d_j f_{i-j}$. The resulting residual $r_i = f_i - \hat{f}_i$ typically follows a distribution with lower entropy than the original sample distribution (*e.g.,* a Laplace distribution), and therefore can be more effectively encoded with entropy coding. Since the residuals are also small (in magnitude), they can also be effectively quantized without introducing large errors. For efficient prediction, the k weights $d_1, \ldots, d_k$ are typically precomputed. This can be done by assuming that locally $\mathbf{f}$ is well approximated by a function g that can be determined using k known values, *e.g.,* a polynomial of degree $k-1$. For example, the original version of SZ [58] predicts f using degree 0, 1, and 2 polynomials, while Fout and Ma [83] uses cubic polynomials for their lossless compressor.

If the source data $\mathbf{f}$ is multidimensional, so can be the polynomial g. One example is the Lorenzo predictor introduced by Ibarria et al. [124] and used in FPZIP [171], which estimates the value of a scalar field at one corner of a (d-dimensional) hypercube from the values at the other corners. The Lorenzo predictor is exact for fields that satisfy $\partial^d f / \partial x_1 \ldots \partial x_d = 0$. It can also be generalized to higher orders (which use more points for prediction), as done in recent versions of SZ [280] and lossless ZFP [167].

To make data more coherent for better prediction, some techniques employ a sorting step prior to prediction. SQ [128] sorts samples based on their coordinates (in either "original order", breadth-first, or priority-first order), before decomposing the samples into subsets and produce one prediction for each subset using the mean value of the subset. On the other hand, ISABELA [150] sorts the samples using their values and then fit a cubic B-spline through a local window of samples. However, this approach necessitates encoding the sorted order, which can often negate most of the compression gain.

2.2.1.2 Linear transformations using a fixed basis

Linear transformation decorrelates a discrete signal $\mathbf{f}$ by expressing $\mathbf{f}$ as a linear combination of simpler basis functions $\mathbf{f} = \sum c_i \mathbf{u}_i$. The idea is that if the basis functions $\mathbf{u}_i$ are chosen well, the coefficients c_i will be sparse, *i.e.,* most of them are zero or near zero. Oftentimes, to compress image data, the basis functions $\mathbf{u}_i$ are chosen so that they capture varying degrees of details, from high frequencies to low frequencies (*e.g.,* for Fourier-based transforms), or from coarse scales to fine scales (*e.g.,* for wavelets). Then, compression can happen by thresholding or heavily quantizing the coefficients associated with the fine-scale/high-frequency details. Such coefficients are often small in magnitude, thus they can be heavily quantized or discarded while preserving most of the function's energy. If the basis functions are also orthogonal or near orthogonal, these coefficients will also be less correlated compared to the original data samples, and thus they can be quantized with cheap scalar quantization to achieve compression ratios otherwise only possible with the more expensive vector quantization on original data samples.

Perhaps the most well-known orthogonal transform is the Discrete Fourier Transform (DFT), whose basis functions are complex exponentials at increasing frequencies (CITE). For scientific data, the DFT has been used for a limited form of direct rendering from the grid of transform coefficients (the *frequency domain*) [68, 184, 286, 294], leveraging the Fourier projection-slice theorem [161], which equates projection (taking integrals) in the sample domain to the less expensive process of taking slices in the frequency domain. For compression purposes, the closely related Discrete Cosine Transform (DCT) is more popular. The DCT is the DFT applied to the periodic version of the input signal using symmetric extension at the boundary, so that the transform coefficients are real (instead of complex) numbers [271]. Since its introduction, the DCT has been the cornerstone in most image and video compression standards, such as the classic JPEG [298], as well as the more recent AV1 [35], JPEG-XL [8], and HEVC [272].

There are two common criteria to evaluate the efficacy of a linear transform: that of *decorrelation efficiency* (how well the transform decorrelates the input signal) and *coding gain* (how much the transform reduces quantization distortion compared to the input signal) [43]. The maximum decorrelation efficiency and coding gain are obtained through the Karhunen Loève Transform (KLT) transform, whose basis vectors diagonalize the covariance matrix of the input signal. Notable uses of the DCT for scientific data include the work by Fout et al. [82], Yeo et al. [318], and Laurance et al. [154]. While there are other well-known orthogonal transforms beside the DCT in the literature, such as Haar [107], Walsh-Hadamard [7], or Slant [237], the DCT is shown to achieve better decorrelation efficiency and coding gain than the other transforms, and is in fact very close to the optimal KLT for common images [43]. The DCT has also found use in scientific data compression [181]. However, for scientific data (and not everyday images), the ZFP transform [169] which is based on the Gram orthogonal polynomial, has been shown to achieve even better decorrelation efficiency and coding gain compared to the DCT and other transforms.

2.2.1.3 Linear transformations using a learned basis

The transform basis can also be data-dependent. There exist two prominent approaches: sparse coding and tensor decomposition. While both approaches aim to express input data vectors as linear combinations of basis vectors, they do so in different ways. In contrast to data-independent transforms, the data-dependent approaches require storing and transmitting these basis vectors, an overhead that is hopefully compensated for by making the coordinate vectors more sparse.

Sparse coding (or dictionary learning) is a generalization of vector quantization, in which each reconstructed vector is now a linear combinations of basis vectors that form an optimal dictionary D (a matrix where each basis vector is a column). Let $\{y_i\}$ denote the set of training vectors and $\{\gamma_i\}$ be their projections onto the column space of D. The goal is to find a D such that the $\{\gamma_i\}$ are sparse (*i.e.,* most entries are 0; such a dictionary is called over-complete). D and $\{\gamma_i\}$ can be jointly computed by solving the minimization problem: $\min_{D,\gamma_i} \sum_i w_i \|y_i - D\gamma_i\|_2^2$ subject to $\|\gamma_i\|_0 \leq K$, where $\|\gamma_i\|_0$ is the number of zero entries in γ_i and w_i is a weight associated with each sample. If D is fixed, the $\{\gamma_i\}$ can be found by various pursuit algorithms This optimization problem can be solved using the K-SVD algorithm [6], which computes the columns of D one at a time. Notable works that follow this approach include COVRA [91] and its successors [59, 188].

The other prominent data-dependent approach is based on tensor decomposition. In 2D, the singular value decomposition (SVD) produces the sparsest projection of the input function into an orthonormal basis. It is not used for compression, however, because the basis vectors outweigh the transform coefficients; such a decomposition is only attractive in higher dimensions. In 3D, a volume A is treated as a third-order tensor (a 3D array), with elements $a_{i,j,k}$. The Tucker model [138] decomposes A into a product of a core tensor B with factor matrices U_i, with one factor matrix for each dimension: $A = B \times_1 U_1 \times_2 U_2 \times_3 U_3$. The factor matrices are nonsingular, and the core tensor is of the same size as A (it is possible to force the core tensor B to be smaller than A in which case it is the projection of A into the basis of its factor matrices). This decomposition can be computed using the higher-order singular value decomposition (HOSVD) [55]. For real-world data, the core tensor B tends to be quasi-sparse, and the HOSVD produces core slices that are non-increasing in norm. As a result, lossy compression can be achieved by either truncating core slices with small norms [14, 275, 316], non-uniform quantization of coefficients [13, 274], or bit plane coding with significance map [12].

2.2.2 Quantization

I discussed the theory of quantization in Subsection 2.1.2. Here I review coders that employ quantization in practice.

2.2.2.1 Scalar quantization

Lum et al. [181] use the Lloyd-Max algorithm [177, 190] to quantize DCT-transformed coefficients. Other examples of adaptive quantization in the literature are the well known floating-point compressors ZFP [169] and FPZIP [171], as well as the universal lossy image compressor JPEG [298]. Both ZFP and (lossy) FPZIP employ a form of logarithmic quantization by dropping least significant bits from the mantissa of floating-point values (ZFP quantizes transform coefficients while FPZIP quantizes original values). Because quantization happens for the mantissa, the absolute errors are not uniform but instead are relative to the magnitude of the input values. This is the quantization preferred by scientists, since they often describe the precision of their data as the number of decimal digits in scientific notation. JPEG, on the other hand, deals with integer image data, but also quantizes transform coefficients nonuniformly using a quantization matrix. The matrix is designed so that high-frequency coefficients (which are typically smaller and less important) are quantized more. Popular lossy compressors for scientific data that use uniform quantization are SQ [128] and SZ [58, 280]. SQ quantizes groups of data samples, while SZ quantizes prediction residuals, both to a prescribed input accuracy. In a recent study [132], it has been found that compression ratios can be improved with SZ by adaptively setting the quantization error at user level in the analysis of cosmology simulation data. The features of interest in this type of data (halos, galaxies etc) are separated from the background "noise", so a uniform quantization is not optimal as it introduces similar distortion to both the "signal" and the noise.

2.2.2.2 Vector quantization

Vector quantization was first used for scientific data by Ning et al. [209]. VQ can also be used on transform coefficients or prediction residuals instead of original samples. For example, Lum et al. [181] apply VQ on scalar-quantized DCT transformed coefficients, while Fout et al. [82] transform each block using the KLT transform to produce several resolution levels, and compress each level using VQ. [81] Schneider et al. [256] also use vector quantization but with a simple Haar-like transform that separates each block of 2^3 voxels into one average and seven difference coefficients. Their VQ scheme (hierarchical VQ, or HVQ) is also the first GPU implementation of VQ for scientific data. VQ can also be used to quantize residuals in combination with VQ of original samples [220, 221]. An example of vector quantization for lossy compression of scientific data is HVQ (hierarchical VQ) [256].

2.2.3 Entropy coding

Entropy coding is often the last step in a compression scheme, after data modeling and (optionally) quantization. The problem here is to turn a sequence of input symbols into a binary sequence such that the length of the output in number of bits is less than that of the input. Many well-known schemes exist for entropy coding, they can roughly be characterized by whether they operate on individual symbols or groups of symbols and whether they output fixed-length or variable-length codes. In practice, the random variables $\{S_i\}$ are rarely independently distributed, and even if they are, it is often not possible to achieve an average code length close to the entropy by coding the symbols $\{s_i\}$ individually (since the code length for each symbol has to be a whole number). Therefore, $\boldsymbol{s}$ is often treated as a sequence of blocks $\{\boldsymbol{s}_i\}$, each of which is mapped to a codeword $\boldsymbol{b}_i$. A block $\boldsymbol{s}_i$ has L_i symbols, and is mapped to a codeword $\boldsymbol{b}_i$ of K_i bits. That is, $\boldsymbol{b}_i = \gamma(\boldsymbol{s}_i)$, $|\boldsymbol{b}_i| = K_i$, $|\boldsymbol{s}_i| = L_i$, $\sum K_i = K$ and $\sum L_i = L$. The lengths of the blocks and their corresponding codewords can either be constant or vary across blocks, leading to four types of mapping: fixed-to-fixed (*e.g.,* ASCII codes), fixed-to-variable (*e.g.,* Huffman codes [123]), variable-to-fixed (*e.g.,* Tunstall codes [252]), and variable-to-variable (*e.g.,* arithmetic codes [312]). Most entropy coding approaches take advantage of the statistics of the input data (with *context/memory* or otherwise) to optimally assign binary codes to symbols. Below I review the most well-known entropy codes and where they are used in the scientific data compression literature.

2.2.3.1 Huffman coding

Huffman coding [123] is a fixed-to-variable coding scheme that assigns codes of variable lengths to individual letters $\{a_i\}$ of an alphabet $\mathcal{A}$. It can be constructed from a binary tree, where each internal node has two children, to which we associate the values of 0 and 1. The tree itself is built bottom-up by first assigning each letter a_i (with probabilities $p(a_i)$) to a leaf node, then recursively grouping the two least frequent nodes under the same parent, until there is only one root. The binary code for each letter a_i is then obtained by traversing top-down from the root to the corresponding leaf. Huffman coding achieves compression by assigning shorter codes to symbols with higher probabilities. It can be shown that the average Huffman code length is within one bit of entropy: $H(S) \leq \texttt{Huffman(S)} \leq H(S) + 1$. However, when $H(S)$ is small, this one-bit overhead can be relatively large. JPEG [298] uses Huffman coding to compress the quantized DCT-transformed coefficients. For scientific volume compression, Huffman coding has been used to compress prediction residuals [58, 84, 165, 280] as well as transform coefficients [106, 176, 246, 308, 318]. Komma et al. [139] compares different entropy coding schemes and general-purpose compressors, including Huffman coding, for lossless compression of CT and X-Ray scans.

2.2.3.2 Arithmetic coding

To achieve bit rates closer to the entropy, even when the symbols are independent, it is necessary to “amortize” the coding cost over multiple symbols, for which one common technique is *arithmetic coding*. To see how arithmetic coding works, let us rename the input sequence $\boldsymbol{s} = \{s_0, s_1, \ldots, s_{N-1}\}$ as $\boldsymbol{s}_N$ to indicate that it has N symbols, and define a relation < (less than) between two sequences $\boldsymbol{s}_N$ and $\boldsymbol{s}'_N$ to mean that there exists $n < N$ such that $\boldsymbol{s}_{n-1} = \boldsymbol{s}'_{n-1}$ and $s_n < s'_n$. Arithmetic coding associates a sequence $\boldsymbol{s}_n$ with a half-open range $I_n = [c(\boldsymbol{s}_n), c(\boldsymbol{s}_n) + p(\boldsymbol{s}_n)) \subset [0,1]$, with $p(\boldsymbol{s}_n) = P(\boldsymbol{S}_n = \boldsymbol{s}_n)$ and $c(\boldsymbol{s}_n) = P(\boldsymbol{S}_n < \boldsymbol{s}_n)$ being the probability and cumulative probability mass functions, respectively, for the sequence $\boldsymbol{S}_n$ of random variables $\{S_0, S_1, \ldots, S_{n-1}\}$. We can relate $c(\boldsymbol{s}_n)$ with $c(\boldsymbol{s}_{n-1})$ by noting that $c(\boldsymbol{s}_n) = P(\boldsymbol{S}_n < \boldsymbol{s}_n) = P(\boldsymbol{S}_{n-1} < \boldsymbol{s}_{n-1}) + P(S_n < s_n \mid P(\boldsymbol{S}_{n-1}) = P(\boldsymbol{s}_{n-1})) = c(\boldsymbol{s}_{n-1}) + p(\boldsymbol{s}_n)c(s_n \mid s_0, s_1, \ldots, s_{n-1})$. Similarly, $p(\boldsymbol{s}_n) = P(\boldsymbol{S}_n = \boldsymbol{s}_n) = P(\boldsymbol{S}_{n-1} = \boldsymbol{s}_{n-1})P(S_n = s_n \mid \boldsymbol{S}_{n-1} = \boldsymbol{s}_{n-1}) = p(\boldsymbol{s}_{n-1})p(s_n \mid s_0, s_1, \ldots, s_{n-1})$. These formulas give a recipe for iteratively updating the range I_n associated with $\boldsymbol{s}_n$ as n increases (these ranges are in fact nested *i.e.,* $I_n \subset I_{n-1} \subset \ldots$). Each range I_n can be encoded by the binary code of any number within that range. This (theoretical) procedure is known as *Elias coding*, while arithmetic coding (or *range coding*) refers to a family of practical approaches [186,222,244,312] that implements Elias coding using finite precision arithmetic. Compared to Huffman coding, arithmetic coding is often slower but achieves bit rates closer to entropy, while also not requiring the source statistics to be known prior to coding (it can be computed on the fly). Arithmetic coders can also more easily exploit correlation among symbols to reduce code size, a technique known as *context coding*. CABAC of H.264 [185] and EBCOT of JPEG2000 [281] are two well-known binary context coders. FPZIP [171] uses range coding to compress prediction residuals for floating-point scientific data.

2.2.3.3 Universal codes

In many cases, scanning the source $\boldsymbol{s}$ to obtain the needed statistics for Huffman or arithmetic coding is impractical, since $\boldsymbol{s}$ may be too large or compression has to be done on-the-fly. In such cases, *universal codes* that do not depend of source statistics such as unary, Elias γ and δ [70], Golomb [95], Rice [243], or Exponential-Golomb [283] codes can be faster and more convenient. These codes are designed to compress positive integers by allocating, to different extents, fewer bits to smaller numbers. The unary code $\alpha(n)$ of a positive integer n comprises of $n-1$ zero bits followed by a one bit. The natural binary expansion $\beta(n)$ of $n > 0$ is the L-bit sequence $\{b_0, b_1, \ldots, b_{L-1}\}$ such that $\sum 2^{(L-1-i)b_i} = n$.

Unlike α, the binary code β is not uniquely decodeable. The Elias γ code fixes this issue by using $\alpha(L)$ to specify the length L of $\beta(n)$, *i.e.,* $\gamma(n) = \alpha(L)\beta'(n)$ (β' is β without the first bit, which is always one and thus is redundant when prefixed with α). This technique of combining β with either α or γ is quite common. For example, the Elias δ code uses $\gamma(L)$ in place of $\alpha(L)$, *i.e.,* $\delta(n) = \gamma(L)\beta'(n)$. Golomb codes (used in JPEG-LS [304]) are parameterized on a number b and has two parts: n div b in unary, then n mod b in binary, *i.e.,* $\texttt{Golomb}_b(n) = \alpha(n \text{ div } b)\beta'(n \bmod b)$. Exponential-Golomb codes (used in H.264/AVC [133]) replace the first part with $\gamma(n \text{ div } b)$. Rice codes are a special case of Golomb codes where b is a power of two, and thus admits very fast encoding/decoding by simple bit shifts. The different codes are optimal for different probability distributions of input symbols. The unary code α is optimal for integers that occur with probability $P(n) = 2^{-n}$. For γ and δ codes, the optimal probabilities are $P(n) = 1/(2n^2)$ and $1/(2n \log_2^2 n)$, respectively. Golomb codes are optimal for coding *geometric* distributions produced by a Bernoulli process, *i.e.,* $P(n) = (1-p)^{n-1}p$ (this is the probability of $n-$symbol runs of a binary symbol occuring with probability p).

2.2.3.4 Lempel-Ziv coding

The above universal codes are good options when the source sequence $\boldsymbol{s}$ can be well modeled by a single global probability distribution. However, many sources in practice are better modeled by a combination of *local* distributions instead (*i.e.,* neighboring symbols exhibit higher correlation compared to distant symbols). In such cases, local methods such as Lempel-Ziv coding [306, 324, 325], interpolative coding [201], or group testing [116] often perform better. Lempel-Ziv (LZ) coding, or *dictionary coding*, builds a local codebook from previously seen symbols, and replaces a subsequence of symbols with a codeword that is a pointer to where it has occured. Previous symbols tend to make a very good codebook (or dictionary), especially for text-bases sources, since prior text tends to be in the same language and style as current text. The different variants of LZ-based methods mostly differ in how the input is parsed, how the dictionary is stored, and how the pointers are represented. While LZ77 [324] uses pointers of type $(\texttt{offset}, \texttt{length}, \texttt{next symbol})$ to refer to any phrase in a sliding window of some length, LZ78 [324] uses only $(\texttt{code}, \texttt{next symbol})$ to refer to a previously seen phrase in a codebook represented as search tree (or *trie*) that is built gradually by adding new phrases as the coder encounters them. LZW [306] removes the need to include the (`next symbol`) by including all symbols of the alphabet into the codebook in the beginning. The pointers in LZ-based methods can be encoded using entropy coding methods such as Huffman coding, as done in DEFLATE [56]. LZ-based methods work well on text-based sources, but their use in image and scientific data compression is limited, mainly because repeating patterns are rarer for data in the latter category. Nevertheless, DEFLATE is used in the lossless image format PNG [28] and is supported by both HDF5 [79] and ADIOS [174, 178], two well-known file formats for scientific data.

2.2.3.5 Interpolative coding

Like universal codes, interpolative coding [201, 202] is a nonstatistical method designed to encode a list of nonnegative integers. It constructs a binary tree where each leaf is assigned one integer in the list, and a parent node is assigned the sum of its children. It then encodes the root as well as all left-child nodes of the tree in a top-down manner, achieving compression due to the fact that if n is the value at a parent node, then at most $\lceil \log n \rceil$ bits are needed to encode a child value. Intuitively, the higher-order bits of the leaf values are stored once in their common ancestor nodes, resulting in compression. Interpolative coding is especially effective when the input integers form clusters since more bits can be shared, and can often achieve better compression ratios than universal codes [311]. It does so while retaining the main benefit of universal codes in that it avoids the need to gather source statistics required by statistical codes such as Huffman. A related idea is tournament coding [284], where a parent node is assigned the maximum value of its two children instead of the sum. Interpolative coding has been used successfully for compressing the position of mesh vertices [57] and of particles [113] in 3D.

2.2.3.6 Group testing

Given a binary string, run-length coding encodes runs of the same bit (either zero or one) using a single number. *Group testing* is a generalization of this idea, where same-value bits can be grouped together and replaced by a single bit even if they do not form a run. Given a nonempty set K of bits, of which some bits are one (significant) and the rest are zero (insignificant), a group test can output either 0 (to indicate that all bits of K are insignificant) or 1 (to indicate that at least one bit of K is significant). If the output is 0, we know that all bits of K are 0 and K needs not be tested further. If the output is 1 and $|K| > 1$, K is split into two roughly equal size parts K_1 and K_2 and the same group test can recurse on each of these parts until all bits of K are coded. This procedure results in a code equivalent to the Golomb code [116]. Group testing originates from the problem of identifying soldiers infected with syphilis using the least amount of tests. The idea is to pool together blood samples of several men, and if syphilis is rare among the population, then one test that returns negative for the combined blood sample is enough to exempt the whole group. This problem was first studied formally by Dorfman [63].

In designing a group testing algorithm, the optimal choice of $|K|$ depends on p, the probability of a bit being significant. Intuitively, we prefer each outcome of a group test (either 0 or 1) to have a probability of approximately 0.5. Thus, $|K|$ should be chosen so that the probability of $|K|$ zero bits occuring is 0.5, or $p^{|K|} = 0.5$. Also important are the way groups are form and how a group is split. An algorithm can be *adaptive* if the number of significant bits is unknown in the beginning. Algorithms to solve the group testing problem are presented in the book by Du and Hwang [65]. In the context of compression, group testing is often used for compressing transform coefficients in images. Since these coefficients usually exhibit spatial coherence, often also across levels of a hierarchy, they can be grouped and tested for significance together. Group testing is common in wavelet-based coders, where a group is often defined as a subtree of wavelet coefficients and testing is done one *bit plane* at a time.

Examples of embedded bit plane coders include EZW [261], SPIHT [249], SPECK [227], SBHP [39], ECECOW [317], and GTW [116]. Some wavelet coders use Golomb codes [183,213] to encode wavelet coefficients. Although not a wavelet coder, ZFP [169] also uses group testing to encode bit planes of transform coefficients. Finally, group testing can also be used on quadtrees to code binary images, or on octrees to code 3D positions of particles, as done by MPEG [257].

Although most resolution-based techniques stress progressivity, here the majority of techniques adopt a single-error, write-once-read-once approach, where compression and decompression happen only at a predetermined quality. This approach requires the user to choose between reducing too much at write time or decoding too much at read time.

2.3 Discrete wavelet transforms

The *discrete wavelet transform* is a linear transform that decomposes an input signal $\boldsymbol{x}$ of length N into two parts: the *trend* $\boldsymbol{s}$ signal and the *detail* signal $\boldsymbol{d}$, each of length $N/2$. The idea is that $\boldsymbol{s}$ captures the essence of $\boldsymbol{x}$ but at half the resolution, while $\boldsymbol{d}$ captures the local variance in $\boldsymbol{x}$, which, when added to $\boldsymbol{s}$, recovers the original signal $\boldsymbol{x}$. In the context of data compression, $\boldsymbol{d}$ can then be either discarded or otherwise heavily quantized while most of the energy of $\boldsymbol{x}$ is preserved by $\boldsymbol{s}$. In the literature, wavelets can be presented from the filter bank or lifting viewpoints. Here, I present wavelets from the lifting viewpoint.

2.3.1 The lifting scheme

Consider a real-valued signal $\boldsymbol{x} = [x_0, x_1, \ldots, x_{N-1}]$. To obtain the detail signal $\boldsymbol{d}$, assuming that $\boldsymbol{x}$ is reasonably smooth, we can *predict* each odd sample x_{2k+1} to be the same as the previous even sample x_{2k}, and record the difference in $\boldsymbol{d}$. To do so, we first split $\boldsymbol{x}$ into an odd and an even part:

$$[\text{split}]\ \boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_{even} \\ \boldsymbol{x}_{odd} \end{bmatrix} \tag{2.34}$$

where $\boldsymbol{x}_{even} = \boldsymbol{x}_{\{2k\}}$ and $\boldsymbol{x}_{odd} = \boldsymbol{x}_{\{2k+1\}}$. Then, the detail signal $\boldsymbol{d}$ is the difference between $\boldsymbol{x}_{odd}$ and $\boldsymbol{x}_{even}$: $\boldsymbol{d} = \boldsymbol{x}_{odd} - \boldsymbol{x}_{even}$. The trend signal can then be $\boldsymbol{x}_{even}$ itself: $\boldsymbol{s} = \boldsymbol{x}_{even}$.

Another reasonable set of rules is for $\boldsymbol{s}$ to be half the sum of the odd and even subsignals: $\boldsymbol{s} = \frac{1}{2}(\boldsymbol{x}_{even} + \boldsymbol{x}_{odd})$, and $\boldsymbol{d} = \boldsymbol{x}_{even} - \boldsymbol{s}$, or, equivalently, $\boldsymbol{d} = \frac{1}{2}(\boldsymbol{x}_{even} - \boldsymbol{x}_{odd})$ and $\boldsymbol{s} = \boldsymbol{x}_{even} - \boldsymbol{d}$. In lifting terminology, the step that computes $\boldsymbol{d}$ is called the *predict* step, while the step that computes $\boldsymbol{s}$ is called the *update* step. In matrix form, we have

$$\begin{aligned} \boldsymbol{d} &= \frac{1}{2}\begin{bmatrix} I & -I \end{bmatrix}\begin{bmatrix} \boldsymbol{x}_{even} \\ \boldsymbol{x}_{odd} \end{bmatrix} \text{ (predict)} \\ \boldsymbol{s} &= \begin{bmatrix} I & -I \end{bmatrix}\begin{bmatrix} \boldsymbol{x}_{even} \\ \boldsymbol{d} \end{bmatrix} \text{ (update)} \end{aligned} \tag{2.35}$$

Putting both transformations in a single matrix T_a (the subscript $_a$ is for analysis), we have

$$\begin{aligned} \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{d} \end{bmatrix} &= \underbrace{\begin{bmatrix} I & -I \\ 0 & I \end{bmatrix}}_{U}\begin{bmatrix} \boldsymbol{x}_{even} \\ \boldsymbol{d} \end{bmatrix} = \underbrace{\begin{bmatrix} I & -I \\ 0 & I \end{bmatrix}}_{U}\underbrace{\begin{bmatrix} I & 0 \\ I/2 & -I/2 \end{bmatrix}}_{P}\begin{bmatrix} \boldsymbol{x}_{even} \\ \boldsymbol{x}_{odd} \end{bmatrix} = \\ &\begin{bmatrix} I/2 & I/2 \\ I/2 & -I/2 \end{bmatrix}\begin{bmatrix} \boldsymbol{x}_{even} \\ \boldsymbol{x}_{odd} \end{bmatrix} = \underbrace{\begin{bmatrix} I/2 & I/2 \\ I/2 & -I/2 \end{bmatrix}}_{T_a}[\text{split}]\ \boldsymbol{x} \end{aligned} \tag{2.36}$$

The matrices P and U are for the predict and update steps, respectively. In general, lifting involves chaining together potentially several predict and update steps. In this simple case, $T_a = UP$. The lifting steps can also be easily inverted to form the inverse wavelet transform, captured by the synthesis matrix $T_s = T_a^{-1} = [\text{merge}]\ U^{-1}P^{-1}$. The columns of T_s are called the *wavelet basis functions*.

2.3.2 Biorthogonal wavelets

The above transformation is called the *Haar* wavelets (CITE), a member of the *Daubechies* dbN wavelet family [52]. Daubechies wavelets are orthogonal and have compact support, making them useful in practice. A variety of wavelets exist, each with different sets of properties that make them suitable for different uses. Such properties include: orthogonality, length of the support, symmetry, number of vanishing moments, and regularity of the wavelet. Unfortunately, some of these properties are mutually exclusive. For example, Haar is the only orthogonal wavelet with compact support and is symmetric; other wavelets in the DbN family are smoother (have more vanishing moments) but are not symmetric. To obtain symmetric wavelets with compact support, we must forgo the orthogonality requirement. The results are *biorthogonal* wavelets [47].

An example of biorthogonal wavelets is the CDF(2, 2) wavelets, with 2 and 2 referring to the number of vanishing moments for the low-pass and high-pass filters. For CDF(2, 2), instead of predicting an odd sample to be the same as the preceding even sample, we predict it to be the average of two neighboring even samples. In other words, we have

$$\begin{aligned} \boldsymbol{d} &= \boldsymbol{x}_{odd} - \frac{1}{2}(\boldsymbol{x}_{even} + S^{-1}\boldsymbol{x}_{even}) \text{ (predict)} \\ \boldsymbol{s} &= \boldsymbol{x}_{even} + \frac{1}{4}(\boldsymbol{d} + S\boldsymbol{d}) \text{ (update)} \end{aligned} \tag{2.37}$$

with S being the shift operator *i.e.,* $\boldsymbol{x}_{n+1} = (S^{-1}\boldsymbol{x})_n$ The $\frac{1}{4}$ factor is o that $\boldsymbol{s}$ and $\boldsymbol{x}$ have the same average value. The corresponding predict and update matrices are

$$P = \begin{bmatrix} I & 0 \\ -\frac{1}{2}(I + S^{-1}) & I \end{bmatrix}, \; U = \begin{bmatrix} I & \frac{1}{4}(I+S) \\ 0 & I \end{bmatrix} \tag{2.38}$$

2.3.3 Multiresolution transforms

The above matrices are for *one-scale* wavelet transforms. To perform multiscale decomposition of the input signal $\boldsymbol{x}$, we recursively apply the same analysis transformation, but only on the trend signal. If we consider the input signal $\boldsymbol{x}$ as the trend signal on some resolution level, say, 3 (*i.e.*, $\boldsymbol{x} = \boldsymbol{s}_3$), then:

$$\begin{aligned}
\begin{bmatrix} \boldsymbol{s}_2 \\ \boldsymbol{d}_2 \end{bmatrix} &= W_a^{(3)} \boldsymbol{s}_3 \\
\begin{bmatrix} \boldsymbol{s}_1 \\ \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \end{bmatrix} &= \begin{bmatrix} T_a^{(2)} & \mathbf{0} \\ \mathbf{0} & I^{(2)} \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_2 \\ \boldsymbol{d}_2 \end{bmatrix} \\
\begin{bmatrix} \boldsymbol{s}_0 \\ \boldsymbol{d}_0 \\ \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \end{bmatrix} &= \begin{bmatrix} T_a^{(1)} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & I^{(1)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I^{(2)} \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_1 \\ \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \end{bmatrix}
\end{aligned} \tag{2.39}$$

Assuming k levels of transform are performed, we can concatenate the transformations at all levels into a single analysis matrix $W_a^{(k)}$, namely

$$\begin{bmatrix} \boldsymbol{s}_0 \\ \boldsymbol{d}_0 \\ \cdots \\ \boldsymbol{d}_k \end{bmatrix} = W_a^{(k)} \boldsymbol{s}_k \tag{2.40}$$

To reconstruct $\boldsymbol{x} = \boldsymbol{s}_k$ from the trend and detail signals at level k, we transform with the synthesis matrix $W_s^{(k)}$ which is the inverse of $W_a^{(k)}$.

$$\boldsymbol{s}_k = W_s^{(k)} \begin{bmatrix} \boldsymbol{s}_0 \\ \boldsymbol{d}_0 \\ \cdots \\ \boldsymbol{d}_k \end{bmatrix} \tag{2.41}$$

The columns $\boldsymbol{h}_i$ of W_s are the *wavelet basis functions* at scales $0, 1, \ldots, k$. The input signal $\boldsymbol{s}_k$ is effectively decomposed into a linear combination of the wavelet basis functions (the components of the trend and detail vectors are called the *wavelet coefficients*). For example,

$$\boldsymbol{s}_3 = \boldsymbol{s}_0[0]\boldsymbol{h}_0 + \boldsymbol{d}_0[0]\boldsymbol{h}_1 + \boldsymbol{d}_1[0]\boldsymbol{h}_2 + \boldsymbol{d}_1[1]\boldsymbol{h}_3 + \boldsymbol{d}_2[0]\boldsymbol{h}_4 + \boldsymbol{d}_2[1]\boldsymbol{h}_5 + \boldsymbol{d}_2[2]\boldsymbol{h}_6 + \boldsymbol{d}_2[3]\boldsymbol{h}_7 \tag{2.42}$$

Such a decomposition is called the *multiresolution representation* of the input signal $\boldsymbol{x} = \boldsymbol{s}_3$. Regardless of the number of resolution levels, the number of wavelet coefficients is always the same as the number of original data samples. The 1D wavelet basis functions on the same resolution level are simply translated versions of a single basis function. For example, in Equation 2.42, $\boldsymbol{h}_2$ and $\boldsymbol{h}_3$ are both associated with $\boldsymbol{d}_1$, meaning they are translated versions of some basis function $\overline{\boldsymbol{h}}_1$. Similarly, $\boldsymbol{h}_4, \boldsymbol{h}_5, \boldsymbol{h}_6, \boldsymbol{h}_7$ are all translated versions of the basis function $\overline{\boldsymbol{h}}_2$.

2.3.4 Multidimensional transforms

When the input signal $\boldsymbol{x}$ is defined in higher dimensions, we can take the tensor product (Kronecker product) of the 1D basis functions to form multidimensional basis functions. For example, in 3D, ignoring translations, each basis function will have the form

$$\overline{\boldsymbol{h}}_{ijk} = \overline{\boldsymbol{h}}_i \otimes \overline{\boldsymbol{h}}_j \otimes \overline{\boldsymbol{h}}_k \tag{2.43}$$

where $0 \leq i, j, k \leq L$ with L being the number of transform passes in each dimension. The Kronecker product $\otimes$ just means $\overline{h}_{ijk}(x, y, z) = \overline{h}_i(x)\overline{h}_j(y)\overline{h}_k(z)$.

The set of wavelet coefficients whose corresponding basis functions are translated versions of the same function $\overline{\boldsymbol{h}}_{ijk}$ is called a *subband*. Effectively, the multidimensional wavelet transform partition the original set of samples into a set of subbands. The subbands capture different combinations of average-detail across the dimensions. For example, one subband may contain fine details along x, but average information along y and z, while another subband contains fine details along y and average information along x and z. There is always one subband that captures average information along all dimensions – this subband represents the original input signal. A nice property of defining multidimensional basis functions as Kronecker products of 1D basis functions is that the transform can be done along one axis at a time.

However, it is not always the case that all combinations of $\{i, j, k\}$ exist. Which combinations of $\{i, j, k\}$ exists depend on how the multiresolution transform is done in higher dimensions. For example, if L transform passes are performed in x, followed by L passes in y and L passes in z, each time done on the whole domain, then indeed every combination of $\{i, j, k\}$ exists. However, if the transform passes in x, y and z are done in an interleaved manner, and next-level transforms are performed only on the coarsest-resolution subband, then the only combinations of $\{i, j, k\}$ that exist are ones in which $\max(i, j, k) - \min(i, j, k) \leq 1$. These two ways of transform are the most common in practice, the former is called the standard decomposition, while the latter is called the nonstandard decomposition [23]. The nonstandard decomposition is slightly more efficient to compute. Additionally, all basis functions in the nonstandard decomposition has "square" supports (same lengths in all dimension), while some basis functions in the standard decomposition have nonsquare support. The choice of subband decomposition depends on the application, although in the data compression literature, the nonstandard decomposition is more popular.

CHAPTER 3

PREVIOUS WORK

This chapter reviews previous work in the literature on lossy encodings of image and particle data.

3.1 Encodings for image data

Lossy compression approaches approximate an input sequence (of, say, image pixels) with another sequence $\boldsymbol{s}'$, before encoding $\boldsymbol{s}'$ using entropy coding. The size of $\boldsymbol{s}'$ can either be the same or smaller than the size of $\boldsymbol{s}$. In the former case, compression can still happen if the values in $\boldsymbol{s}'$ are expressed using a smaller alphabet. Examples of this case include the scalar quantization methods discussed in Subsection 2.2.2. In practice, methods that reduce the alphabet, but not the length of $\boldsymbol{s}'$, are said to be reducing the data's *numerical precision* (*i.e.,* reducing the number of bits required to specify each data sample). In contrast, methods that reduce the length of $\boldsymbol{s}'$ but not necessarily its alphabet, are said to be reducing the data's *spatial resolution*. In this case, each data sample in $\boldsymbol{s}'$ is often an average value that serves as a surrogate for a block of samples in $\boldsymbol{s}$. From this point of view, precision-reduction corresponds to scalar quantization, while resolution-reduction corresponds to vector quantization. In either case, the quantization can either be *uniform* (data-indepdendent) or *adaptive* (data-dependent), with the common trade-off that data-dependent methods tend to compress better at the expense of computational complexity.

For reasons that are perhaps historical, in the literature, precision-reduction methods and resolution-reduction methods are treated as disctint paradigms, often studied by different communities: the former by data compression researchers and the latter by computer graphics researchers. These communities work with different assumptions and priorities: data compression deals with relatively smaller datasets and prioritizes data quality, wheares computer graphics deals with larger datasets and prioritizes speed through smaller data sizes. However, we will see that combining both paradigms allows us to find better quality-size/speed tradeoffs, which is crucial for handling today's terabyte-scale scientific datasets. After all, (vector) quantization is about reducing both the alphabet (precision) and the sequence length (resolution). Intuitively, when either is reduced to a certain degree, it is wasteful not to reduce also the other.

If the difference $\boldsymbol{d}$ between $\boldsymbol{s}$ and $\boldsymbol{s}'$ is kept, $\boldsymbol{s}$ can be recovered from $\boldsymbol{s}'$ and $\boldsymbol{d}$. Furthermore, $\boldsymbol{s}'$ itself can be recursively quantized, effectively decomposing the original $\boldsymbol{s}$ into a sequence of signals

3.1.1 Precision-based encodings

Other approaches [169,227,249] provide an additional degree of flexibility: the ability to specify a desired precision during decompression. Many of these approaches are used in combination with wavelets or other transforms, encoding the coefficients one bit plane at a time. Progression in precision is achieved by sorting the bit planes in decreasing order of significance, thus decoding the bit stream gives a progressive best effort approximation. For effective compression, however, a bit plane will often span multiple resolution levels, which complicates decoding when only a subset of the levels are desired. Most embedded coders encode the transform coefficients one bit plane at a time, using some form of group testing [116]. The ZFP compression scheme [167] encodes transform coefficients by bit plane, in order of decreasing significance. By partitioning the domain into $4 \times 4 \times 4$ independent blocks, ZFP supports fixed-rate compression, random access to the data, localized decompression, and fast inline compression. Extensions of ZFP allow it to vary either the bit rate or precision spatially over the domain, albeit at fixed resolution [170].

Most work that explores the precision axis can be found in wavelet coders. Wavelet coefficients in corresponding regions across subbands can be thought of as belonging to a "tree". The embedded zerotrees (EZW) coder exploits the observation that in such trees, "parent" coefficients are often larger in magnitude than their "children". EZW therefore locates trees of wavelet coefficients that are insignificant with regard to a series of thresholds and encodes such a tree with one single symbol. The thresholds are typically set at the bit planes, starting from the most significant one. In this way, the data can be progressively refined in precision during decompression. The SPIHT coder [249] improves on EZW by locating more general types of zero trees [36]. SPECK [227] extends SPIHT to also exploit spatial correlations among nearby coefficients on the same subband.

3.1.2 Resolution-based encodings

Another common data encoding paradigm is multilevel representations, in which the data is decomposed into multiple “levels” (most often either resolution levels or precision levels). Such encodings allow data to be partially decoded to obtain data approximations, by decoding a subset of the levels. More bits can be decoded on-demand to obtain progressively better approximations. This approach allows users to work immediately with decoded data, and having their approximations refined over time, without having to wait for the whole data to decode, which can take a long time.

Most resolution-based approaches subdivide the space to form a tree, where branches denote a spatial subdivision in one or more dimensions. How the dimensions are treated (simultaneously [151,205] or independently [77,314]) defines the shape of the tree. At coarse levels, data samples are obtained through some form of weighted averaging [29,151,238]. These trees often duplicate data for lower resolutions and therefore incur overheads in progressive data loading. Fine-level nodes can be discarded based on a threshold, thereby storing only sparse trees [22,49,78,93,109]. However, such approaches primarily support visualization and not numerical analyses.

Another group of techniques use data-dependent basis transforms [12–14,82,91,274,275], expressing the data as a linear combination of multi-dimensional basis functions forming levels akin to resolution. Common image and video compression methods also take this approach using variants of the discrete cosine transform [35, 272, 298], but with data-independent bases, thus trading quality for speed during encoding. These approaches are often limited in reconstructing coarse approximations, since often their “resolution levels” and the actual data samples in the original grid have no direct correspondence. Therefore, the (inverse) transform must be done at full resolution, and only subsequently can redundant samples be discarded, which is costly. Some approaches circumvent this limitation by constructing an octree before the transform [59,91,188]. Still, these approaches provide limited or no progression in precision.

Another notable approach, the IDX file format [146, 149], supports fast decoding by rearranging grid samples into a spatially coherent hierarchical space-filling curve [96,223]. Because the rearrangement is efficient, IDX and techniques derived from it have been shown to scale to very large data sets [145,224,245,273]. In this work, I show that IDX's hierarchy [223] is a specific type of wavelet hierarchy, just without data filtering. Lacking data filtering means IDX does not have an interpolation basis and the coherency needed for effective compression.

A very common scheme to generate a tree-like hierarchy is to construct low-resolution copies of the data from higher resolution ones through downsampling. Examples include Gaussian and Laplacian pyramids [29] and mipmaps [151,238]. The data is often stored in blocks on each resolution level. To save bandwidth, low-resolution versions of distant blocks can be streamed during rendering. However, these methods increase storage requirements, making them unsuitable for very large data. Recent multiresolution techniques save storage by adapting to the data in such a way that different regions are stored with different resolutions. A very popular approach is sparse voxel octrees [49,93] and variations [22,78,109]. Smooth-varying regions are stored at coarser octree levels, which significantly reduces storage. During rendering, blocks of samples are streamed from an appropriate resolution, determined by how far the queried samples are from the eye/camera. A sparse, multiresolution hierarchy can also be built using other trees such as B+ tree [205] and kd-tree [77,314], or space-filling curves [96,223], which reorder data samples to form a hierarchy without any filtering steps or redundant samples. Low-resolution levels are constructed via subsampling, which, unfortunately, is prone to aliasing problems.

Other multiresolution approaches reduce data by transform-based compression. For example, COVRA [91] constructs an octree of bricks (consisting of blocks) and learns a sparse representation for the blocks in terms of basis blocks. Similarly, Fout et al. [82] transform each block using the KLT transform to produce several resolution levels, and compress each level using vector quantization [209]. Schneider et al. [256] also use vector quantization but with a simple Haar-like transform that separates each block of 2^3 voxels into one average and seven difference coefficients. Other examples of transform-based hierarchies include works that use the Tucker decomposition [13,14,274,275], which decomposes the input data (stored as a tensor) into n matrices of basis vectors and one core tensor. Tensor decomposition works for higher dimensional data and can achieve high compression ratios, albeit at the price of a costly transform step.

The precision-based level of details (PLoD) scheme proposed in MLOC [96] truncates floats by dividing a double-precision number into several parts. MLOC includes a multiresolution scheme based on Hillbert curves, but this scheme (based on resolution) and the PLoD scheme (based on precision) are exclusive.

3.1.3 Adaptive-resolution encodings

Tree-based hierarchies, such as k-d trees [77, 314] and octrees [129], are among the most popular spatial-subdivision schemes due to their simplicity. Octrees, in particular, have found widespread adoption across diverse domains. They are especially useful when the data contains sparse details, so the smooth-varying regions can be stored at coarser octree levels, thereby reducing storage, *e.g.*, using sparse voxel octrees [22, 49, 93]. Recent approaches have made modifications to traditional octrees to leverage modern computational architectures. For example, OpenVDB [205] increases the tree branching factor to ensure that sibling nodes are stored contiguously in a cache-friendly layout; SPGrid [259] stores octree levels separately as sparse and nonoverlapping grids, taking advantage of virtual memory handling capacities in modern operating systems. Similar to SPGrid, I store per-level vertices separately and aggregate them on demand, but use hash tables instead of the OS's virtual memory to handle sparsity.

Adaptive mesh refinement (AMR) [19, 67] is another popular class of multiresolution schemes, especially for simulations. In structured AMR, each resolution level consists of a set of nonoverlapping uniform grids. As the grids can be placed arbitrarily, fine-resolution grids can be used to quickly resolve fine details. An AMR mesh can be either vertex-centered or cell-centered, depending upon where the data points are stored. Although the cell-centered approach is more common, visualizing the resulting mesh requires preprocessing steps, such as remeshing and stitching [204, 303] and *ad hoc* interpolation [175, 299].

Wavelets provide a rigorous framework for multiresolution decomposition that is also amenable for data reduction and, thus, is commonly used in visualization frameworks for large data [162, 163, 287, 315]. There also have been works that study data simplification and approximation using wavelet-based subdivision schemes [20, 100], as well as representing multilinear functions using a minimal number of mesh elements to reduce memory footprints [172, 305].

Linsen et al. [172] subdivide cubes into simplices and use linear interpolation for function reconstruction. Weiss and Lindstrom [305] later demonstrated that using multilinear interpolation produces superior quality meshes with respect to approximation error. Bhatia et al. [25] extends Weiss and Lindstrom's approach by utilizing rectangular cuboidal cells as opposed to cube-shaped cells (of a standard octree hierarchy), thereby significantly reducing memory requirements. Both of these work use linear B-spline wavelets [47], since multilinear interpolants are at the foundation of many visualization techniques [11, 41, 180, 208]. Sparse grids [87, 321], a common solution for circumventing the curse of dimensionality when solving partial differential equations, also form a piecewise multilinear multiresolution basis. Wavelets [52] are the most common transforms that support fast multiresolution decoding. Each transform step separates the input samples into equal halves: (1) *low-pass* coefficients representing a coarser grid, and (2) *high-pass* coefficients containing fine details absent in the first half. This transform can be recursively applied to produce a hierarchy of coefficients capturing details at multiple scales. Wavelet transforms are local and fast, and the coefficients are highly compressible [37, 39, 227, 249, 261, 282], and thus they are used in various data reduction systems [45, 162, 239, 315]. The multiresolution nature of the wavelet transform also makes it useful for, e.g., level-of-detail visualizations [105, 106, 125, 207, 241, 256, 287, 307]. Nevertheless, most of these systems do not take advantage of precision-based reduction, or do so only as a final lossy compression step with no progression. In contrast, my unified tree seamlessly consolidates resolution and precision.

Transforms that use fixed bases avoid such high computation cost at the expense of slightly less effective compression. Perhaps the most popular transform that uses a fixed basis is the (discrete) wavelet transform (DWT), which constructs a hierarchy of resolution levels via low and high bandpass filters. The transform is recursively applied to the lower resolution band, resulting in a hierarchy of "details" at varying resolution. The DWT is merely a change of basis that does not increase the data size. Furthermore, the wavelet basis functions are defined everywhere in space, requiring no special interpolation rules when given some arbitrary subset of wavelet coefficients. One disadvantage of the DWT is the random access cost, which is not constant time, although there has been work to develop acceleration structures to speed up local queries [305].

Besides offering a multiresolution decomposition, enabling data streaming with level-of-detail support, wavelet coefficients are especially amenable for compression, through thresholding or entropy compression. In storing and visualizing scientific data, wavelets (with compression) are used in a wide variety of systems [45,162,315] and applications, such as volume rendering with level-of-detail [105,106,125,207,307], turbulence visualization [287], and particle visualization [241].

Most wavelet-based techniques employ tiling of wavelet coefficients in individual subands to facilitate random access and spatial adaptivity in resolution. For example, the VAPOR toolkit [162] incorporates a multiresolution file format based on wavelets to allow data analysis on commodity hardware by storing individual tiles in separate files to allow loading of the region of interest. However, like most multiresolution work, only the resolution control is leveraged. The precision axis, which can potentially further reduce data transfer, is left unexplored.

3.1.4 Quality levels

JPEG2000 [282] allows for the selection of a small set of quality levels (computed at compression time) that are optimal combinations of resolution and precision in L_2 norm. This approach has disadvantages when applied to scientific data, since the preselected quality levels are quite limited, with little control over how those are achieved. I instead explore how the precision-resolution space can be navigated flexibly. JPEG2000 is also designed for imagery and not scientific data and as such does not support high-precision data. In addition, it is not concerned with large out-of-core data and therefore does not optimize for disk I/O. Overall, its optimization is tailored for visually appealing images and not necessarily the best for achieving scientific tasks, may require using very different kinds of error metrics.

The most relevant approach to the target of this work is VAPOR [46, 162], a data visualization toolkit that uses wavelets for compression and multiresolution access. VAPOR also exposes a set of predetermined quality levels at compression time. At read time, VAPOR can selectively fetch wavelet coefficients as the user increases the quality level and/or resolution level, reducing I/O and computation overheads. However, the quality levels correspond to the number of wavelet coefficients to decode and are not a direct control over data precision. This, together with the fact that only a few quality levels are supported (by default 4), means that the control over data quality is quite limited.

Schemes that allow progressive data access in both resolution and precision include SBHP [39] and JPEG2000 [264]. Both partition each subband into blocks and code each block independently, in bit plane order. By interleaving compressed bits across blocks, one can construct a purely resolution-progressive or a purely precision-progressive stream, or anything in between. JPEG2000 has found use in the compression of scientific data, e.g., by Woodring et al. [315]. Since most JPEG2000 implementations are limited to integer data, the authors apply uniform scalar quantization to convert floating point data to integer form. Even though JPEG2000 supports varying both resolution and precision, most works do not explore this capability but focus only on setting bit rate. In general, efficiently leveraging both axes of data reduction has not been well studied.

3.2 Encodings for particle data

In this section I give an overview of the literature on particle (point cloud) data management and compression.

3.2.1 Particle hierarchies

One of the most common ways to introduce structure to a particle dataset – to facilitate compression – is to impose a spatial hierarchy (a tree) on the particles. Many state-of-the-art compressors follow this approach, where the tree can be one of many types, *e.g.,* binary trees [92], quadtrees [255], octrees [9,86,118,120,157,173,194,216,219,232,254,257], k-d trees [42,57], and bounding-volume hierarchies [248]. An octree where each node stores the occupancy of its children is by far the most common approach. A hierarchy helps compression in two ways. First, the higher position bits are "distributed" into coarser tree levels and shared among particles in the form of coarse tree nodes. Thus, in finer nodes, one needs to store only the lower order bits for the particles within, possibly with truncation [117,121]. Second, regions with no particles (empty space) are quickly identified and carved away, further reducing the number of bits needed to accurately locate particles — a key property that helps both compression and rendering [253,296,297].

Although a tree naturally provides a progressive coarse-to-fine structure, from which representative particles can be decoded and viewed [85,248], some techniques generate levels of detail through subsampling [92,118,245,258,288,313], which requires no data duplication at coarse levels, and is often faster to compute. Random subsampling [92,258,288] may seem a reasonable choice, but leads to suboptimal compression because the bounding volumes for coarse particle subsets are not easily bounded. This is not the case with the lazy wavelet inspired *odd-even* subsampling, which exactly halves the bounding volume at each level. Wavelet-based downsampling is common for compressing mesh vertices [135,182,290]. When a mesh is not readily available, connectivity can be introduced by building a graph [34], local graphs [263,302], or a resampled signed distance field [141] from the particles. Instead, I use a regular grid, which is simple and fast to compute.

3.2.2 Error-guided tree construction and traversal

Minimizing approximation error can be cast as a (hierarchical) clustering problem, where, at each level, particles are clustered and represented with points chosen to minimize some error metric [75,98,117,118,168,226,231]. More data-adaptive hierarchies reorder child nodes based on their predicted occupancy [120], or make planes of k-d divisions adaptive to local variations [42]. The trade-off between quantization (imprecise particles) error and discretization (low particle count) error has been studied both in theory [137] and practice [158,291] for triangle meshes, where refinement heuristics are given based on geometric distortion measures, including a progressive reconstruction that ranks octree nodes by a priority value [233].

My *adaptive traversal* instead assumes no connectivity information and works on generic particle data. For reconstructing point-sampled geometry, DT has been shown to be memory efficient whereas BT gives better progressive reconstruction [27]. In fact, BT is by far the more preferred traversal order in the literature. However, I show that the reconstruction quality of DT can be vastly improved through my *odd-even* decomposition of space. Finally, some studies have focused on task-based error metrics for point clouds beyond PSNR [9,64]. My *block-adaptive traversal* also facilitates a user-specified error heuristic at decoding time independently of how the data are encoded.

3.2.3 Large-scale and out-of-core techniques

Techniques that handle large data usually organize the data into blocks, so that each block can be randomly accessed and decoded independently as needed [253,258]. Multilevel hierarchies that treat subtrees as blocks are also not uncommon [31,66,85,136], but previous approaches traverse both the coarse-level tree and the fine-level subtrees (blocks) using BT, which restricts the traversal to a *single progressive order*, where blocks are traversed one by one with potential memory reuse in between. In contrast, by using DT within the blocks, my *block-hybrid trees* allow for *simultaneous, independent, and progressive decoding* of all blocks, not one at a time. This approach provides excellent computational gains because thanks to DT's low memory footprint.

3.2.4 Modeling for compression

For effective compression, techniques often assume some model for the particle data. The model can be prescribed, using *e.g.,* local planes [94,210,219,226,254,267,302], higher order surfaces [76, 255], self-similarity of patches [60, 122], grid-based or graph-based transforms [199,242,285], or learned from training data [64,102,119,206,300]. The model can also be statistical [86,153,194], which often means using a frequency histogram to drive an arithmetic coder [200]. It is also common to sort particles to introduce coherency, either with a graph-based traversal [103,197] or by directly using particle coordinates [131,143,212,279]. My *odd-even context coding* assumes a statistical model but is unique in that it relies on self-similarity between subsampled versions of the same point set, which is an idea not previously explored.

CHAPTER 4

HIERARCHICAL MODEL UNIFYING RESOLUTION AND PRECISION

Consider discrete samples of a scalar field defined on a regular grid. The samples can be defined in three domains simultaneously: the original image domain, a tree domain, and a linear storage domain (see Figure 4.1). The image domain is important for spatial display and computations. For example, data filtering and feature extractions happen in this domain. The tree domain is important for reasoning about data approximations and apdative refinements. In this domain, approximations are often formed by pruning tree nodes toward the leaf level, and adaptive refinements corresponds to selecting which subtree to include. The storage domain is important for reasoning about reading and writing data samples from and to computer memory, which, unlike the image domain, is inherently linear. Here, the ordering of the samples matter, since memory I/O happens not in individual but rather blocks of samples.

An *id* function map sample coordinates in the image domain into the linear storage domain, *i.e.*, $id(x, y, z) = i$ means the sample at coordinates (x, y, z) in 3D is stored at index i in memory or on disk. The *id* function is one-to-one, meaning no two samples share the same index. If we consider the domain of *id* to be not the entire R^d but only restricted to where the input field is defined, then this function is not necessarily onto, meaning there can be slots in the linear storage where no samples Similarly, every sample maps one-to-one to a node in the tree. To determine which node, a *lvl* function returns the tree level where the node belongs, and a *parent* function returns the node's parent.

There are of course many ways to define such functions, depending on the objective. The goal here will be that these functions map a linear read of the storage into a reasonable progressive top-down refinement of the tree, and that nearby samples in the image domain are also nearby on storage. The reason is that computer memory is read and written in units of blocks of samples, not individual ones. Therefore, it is desirable to put samples that are likely to be needed together near one another on the storage. In this chapter, I will discuss a family of functions that have these desired properties.

4.1 Precision-resolution tree

Given a scalar field defined on a regular grid, the goal is to build a hierarchy that captures both resolution and precision. I will first establish a set of rules that define such a *resolution tree*, T_r. Although T_r does not model precision yet, it is an important starting point for the unified *precision-resolution tree*, denoted as T_r^p. I will show that known hierarchies, such as the hierarchical-Z space-filling curve [223] and multiresolution wavelet hierarchies [240], are specific instances of T_r. T_r may be defined in many ways. Here, I propose a family of trees that can be characterized by a positive integer d, which controls the tree's branching factor. In particular, I require the root of T_r to have $2^d - 1$ children and every other internal node to have at most 2^d children. I will first describe the construction of T_r with $d = 1$ (Subsection 4.1.1) before generalizing to $d > 1$ (Subsection 4.1.2). Hereafter, I refer to a data value stored at a specific grid point as *sample*, and as *grid point* when the value is not relevant.

4.1.1 Formulation of resolution tree for $d = 1$

To construct the resolution tree, I need to construct a map that takes a grid point to a node in the tree; this map consists of two functions. Without loss of generality, I describe these two functions — f and h (and an auxiliary one, g, to aid the derivation of f) — for a 2D grid, and illustrate them in Figure 4.2.

- **The *index function*,** $f(x, y) \rightarrow i$, maps the spatial coordinates (x, y) of a grid point to an integer $i \geq 0$ that represents the breadth-first traversal order of the corresponding node in T_r.

- **The *parent function*,** $h(i) \rightarrow j$, maps the index, i, of a non-root node in T_r to that of its parent, j ($j < i$). With $d = 1$, each non-leaf node except the root has two children, so the parent function is simply $h(i) = \lfloor i/2 \rfloor$.

- **The *level function*,** $g(x, y) \rightarrow l$, which maps the spatial coordinates of a grid point to an integer $l \geq 0$ — the level of the corresponding tree node, with the convention that the root is at level 0.

To understand g, consider traversing the tree in breadth-first order one level at a time (see Figure 4.2). The set of nodes visited through level l constitutes a subgrid $G_l = \{f^{-1}(0), \ldots, f^{-1}(2^l - 1)\}$ (the set of points with indices $0, \ldots, 2^l - 1$). By design, $G_{l-1} \subset G_l$ and $G_{L-1} = G$, with L being the total number of levels, i.e., the complete grid is traversed by the end. Going from G_3 to G_4 in the shown example, 2^3 grid points at level 4 and with odd x coordinate are introduced, whereas going from G_2 to G_3, 2^2 grid points at level 3 and with odd y coordinate are introduced. This alternating pattern continues and can be formalized using the notion of the Z index formed by interleaving the bits of the two coordinates (known as the Morton code [292]). Assuming the interleaving pattern $yxyx \ldots yx$, Z indices that end with the bit 1 correspond to grid points with an odd x-coordinate, or $g(x_{odd}, y) = L - 1$ ($L = 5$ in this example). Similarly Z indices that end with bits `10` belong to level $L - 2$, or $g(x_{even}, y_{odd}) = L - 2$. In general, Z indices that have a trailing bit pattern of `10...0` (one followed by m zeros) belong to level $g(x, y) = L - 1 - m$.

Formally, I partition the n bits of Z into a *prefix-1-suffix sequence*, $Z = \mathtt{P} +\!\!+\ 1 +\!\!+\ \mathtt{S}$, where $+\!\!+$ is bit concatenation, the suffix $\mathtt{S} = \mathtt{0...0}$ starts after the rightmost bit 1 in Z, and m is the number of bits in $\mathtt{S}$. If Z has no ones, $Z = \mathtt{S}$, $\mathtt{P}$ is empty, and $m = n$.

Prop. 1. *A grid point (x,y) with $Z(x, y) = \mathtt{P} +\!\!+\ 1 +\!\!+\ \mathtt{S}$ belongs to level $l = g(x, y) = L - 1 - m$ of T_r. The nested grid sequence $G_0 \subset \cdots \subset G_{L-1}$ has $L = n + 1$ levels.*

Let (x_i, y_i) be the grid coordinates of a node i and (x_j, y_j) those of its parent j. Since $h(i) = \lfloor f(x_i, y_i)/2 \rfloor = f(x_j, y_j)$, in general the binary expansion of $f(x_j, y_j)$ has one more leading zero bit than that of $f(x_i, y_i)$. Further, since $g(x_i, y_i) = g(x_j, y_j) + 1$, the binary expansion of $Z(x_j, y_j)$ in general has one more trailing zero bit than that of $Z(x_i, y_i)$. Therefore, f can be obtained by swapping the prefix ($\mathtt{P}$) and the trailing zeros ($\mathtt{S}$) portions of $Z = \mathtt{P} +\!\!+\ 1 +\!\!+\ \mathtt{S}$ around the middle 1 bit. Effectively, the level-indicating bits ($\mathtt{S}$) are brought to the front, so that lower resolution grid points are traversed first in T_r.

Prop. 2. *The index of a grid point (x,y), with $Z(x, y) = \mathtt{P} +\!\!+\ 1 +\!\!+\ \mathtt{S}$ is $f(x, y) = i = \mathtt{S} +\!\!+\ 1 +\!\!+\ \mathtt{P}$, and that of its parent is $h(i) = \lfloor i/2 \rfloor$.*

Exact calculations of f, g, and h for all the 16 grid points of a 4×4 grid, as well as the resulting T_r, are included in Figure 4.2.

4.1.2 Generalized formulation of resolution tree for $d > 1$

In general, a generic framework for a such resolution hierarchy, T_r, satisfies the following properties.

- There is a bijective map, $\mathcal{M}_d$, from the set of points of a grid G to the set of nodes in T_r. $\mathcal{M}_d$ consists of an index function $f_d(x, y) \to i$ and a parent function $h_d(i) \to j$. The root node of T_r has $2^d - 1$ children, and every other non-leaf node has at most 2^d children.
- Each node in T_r is associated with a *data value* that represents some approximation of the data. This data value may or may not be the same as the corresponding *sample*.
- The root node of T_r (at level 0) represents an approximation of the entire field as a grid, G_0, corresponding to a single value, whereas a fully refined tree represents G exactly. Traversing T_r top-down from the root, the inverse map, $\mathcal{M}_d^{-1}$, can be invoked on the nodes visited at each subsequent level l, yielding a nested sequence $G_0 \subset G_1 \cdots \subset G_{L-2} \subset G_{L-1} = G$ of L grids, each providing an increasingly finer approximation of the field represented by G.

The functions f, g, and h defined in Subsection 4.1.1 are hereafter referred to as f_1, g_1, and h_1, and can be generalized to f_d, g_d, and h_d for any positive integer d. Although d can be used purely to control the branching factor of T_r regardless of the dimensionality (the same way a kd-tree, $d = 1$, branching factor 2 can be built on a 3D grid), letting d equal the dimensionality of the data can be more intuitive and natural since doing so leads to common types of wavelet subband decompositions.

Recall that g_1 was defined by counting the number of trailing zero bits in Z. To generalize, I instead count the number of groups of d trailing zero-bits (left-padding Z with `0`s if necessary) and denote that number as m_d. Then, $g_d(x, y) = L - 1 - m_d = \lceil n/d \rceil - m_d$ with $L = \lceil n/d \rceil + 1$ being the total number of levels in the tree and n the total number of bits in Z. Note that when d equals the dimensionality of the data, n is a multiple of d. To generalize the formulation of f_d, I partition Z into $Z = \mathtt{P} \mathbin{+\!\!+} \mathtt{F} \mathbin{+\!\!+} \mathtt{S}$, where $\mathtt{S}$ is the longest sequence of $d \times m_d$ trailing zero bits in Z, `F` is the next sequence of d bits, and `P` is the remaining prefix of Z. If Z has no ones, $Z =$`S` and both `P` and `F` are empty, and $m_d = \lceil n/d \rceil$.

Prop. 3. *A grid point* (x, y) *with* $Z(x, y) = \mathtt{P} + \!\!\!+ \ \mathtt{F} + \!\!\!+ \ \mathtt{S}$ *belongs to level* $l = g_d(x, y) = L - 1 - m_d$ *of* T_r*. The nested grid sequence* $G_0 \subset \cdots \subset G_{L-1}$ *has* $L = \lceil n/d \rceil + 1$ *levels.*

Prop. 4. *The index of a grid point* (x, y) *with* $Z(x, y) = \mathtt{P} + \!\!\!+ \ \mathtt{F} + \!\!\!+ \ \mathtt{S}$ *is* $f_d(x, y) = i = \mathtt{S} + \!\!\!+ \ \mathtt{F} + \!\!\!+ \ \mathtt{P}$*, and that of its parent is* $h_d(i) = \left\lfloor i/2^d \right\rfloor$.

For example, consider the tree with $d = 2$ for a 4×4 grid in Figure 4.3a. We have $Z(2, 2) = \mathtt{1100}$. Here, `P` is empty, `F = 11`, and `S = 00`. Furthermore, $m_2 = 1$ and therefore $g_2(2, 2) = 4/2 - 1 = 1$. Swapping `P` and `S` around `F`, we obtain $f_2(2, 2) = \mathtt{0011} = 3$. The parent of this node has the index $h_2(3) = \left\lfloor 3/2^2 \right\rfloor = 0$.

If the input grid G is in 3D, with $\mathcal{M}_1$, each subgrid G_i grows twice as large on the next level (i.e., $|G_{i+1}| = 2|G_i|$), but only in one dimension at a time, similarly to a kd-tree. With $\mathcal{M}_2$ instead, each G_i grows in two dimensions at a time similarly to a quadtree and, therefore, $G_{i+1} = 4|G_i|$. With $\mathcal{M}_3$, G_i grows by 8 times like an octree, with expansion happening in all three dimensions with each increasing level. Note that $\mathcal{M}_1$ describes exactly the hierarchical-Z space-filling curve [223] (here, I provide an alternative formulation), whereas $\mathcal{M}_2$ and $\mathcal{M}_3$, respectively, describe the primal subdivision approach introduced by [160], as well as the most standard type of multiresolution 2D and 3D wavelet subband decompositions [270]. Thus, hierarchical-Z indexing can be considered a form of wavelet decomposition but without actual data filtering (also known as the *lazy wavelet transform*). Such concepts are unified in a formal framework in this study.

4.1.3 Precision-resolution tree

By definition, a T_r can be traversed only in the order of resolution, as the corresponding nodes store data values at full precision. I next focus on incorporating precision to T_r to form a *precision-resolution tree*, T_r^p. To achieve progression in precision, I first define the concept of a *bit plane*: assuming all sample values $\{V_k\}$ are P-bit non-negative integers, a *bit plane*, B_i $(0 \le i < P)$ is the set of bits (of the samples or transform coefficients) that share the same bit position, i. With this definition, each of T_r's nodes is split into a sequence nodes in T_r^p and connect the sequence through a chain, such that each node in the sequence now encodes a bit of the original node in T_r. Such a chain connects the bits of a value from the MSB (most significant bit) to the LSB (least significant bit). To finish the construction of T_r^p, let us bring over all the edges of T_r, using the MSB node of each sequence as a proxy to the original node in T_r. Figure 4.3 visually demonstrates this construction.

If the sample values are non-negative floating-point numbers, I express all values in the form $V_k = 2^E Q_k$ with a single exponent E and then consider the quantized integer values Q_k. I adopt the convention that B_0 is the least significant bit plane, and define a *precision level*, B_p, to be the set of all bit planes $\{B_i\}$, such that $i \geq P - p$ with P being the total number of bit planes. Likewise, a *resolution level* L_l is the set of all grid points whose corresponding nodes in T_r^p belong to levels that are at most l. Note that so far we avoided discussing the sign bits, for simplicity. To handle sign bits, we can either treat them as being on their own bit plane, or use *negabinary* base instead of the usual binary base to express the quantized integer values, *i.e.*, $Q_k = \sum -2^i b_i,$.

4.1.4 Approximations using valid cuts

With T_r^p defined, approximations to a scalar field in both precision and resolution can be defined using the classical notion of a *valid cut* [40,54] of T_r^p. Given a T_r^p and a subset C of its nodes that contains the root, a *cut* is defined as the set of edges leaving C. The cut is a *valid cut* if it does not contain two edges on the same path from the root of T_r^p to any leaf. Figure 4.3 gives examples of two invalid cuts and a valid one. If C corresponds to a valid cut, then C defines an approximation of the data in precision and resolution. In my construction of T_r^p, obtaining a C corresponding to a valid cut is equivalent to obtaining an approximation by always retrieving coarse-resolution and higher order bits first. Doing so is desirable because coarse-resolution bits tend to contain more function energy and higher order bits have bigger impact on error. Restricting the set of admissible approximations to valid cuts of the precision-resolution tree also significantly reduce the search space for the optimal approximation under certain constraints, thus making it more possible to devise efficient algorithms to perform such a search. Finally, maintaining a valid cut in memory is simpler compared to maintaining an arbitrary cut, since the former requires only one "marker" on each path from the root to a leaf, whereas the latter requires specifying any subset of the edges. Figure 4.4 shows an example of three different cuts that correspond to three different approximations in the precision-resolution space. The precision-resolution tree is constructed so that any reasonable approximation in the precision-resolution space can be realized by a cut of the tree.

4.1.5 Differences to classical trees

Traditional spatial hierarchies are k-d trees, quadtrees, or octrees [250, 251]. These trees are most popular for the purpose of partitioning space into non-overlapping regions on each level, with each region assigned to a node. A parent node's region is divided into 2 (k-d trees), 4 (quadtrees), or 8 (octrees) regions, each assigned to a child node. In this way, on each level, there exists a mesh where each cell corresponds to a node of the tree, and a parent's cell is the union of its children's. A node's value (*i.e.,* a data sample) is assigned to the corresponding cell: the mesh on each level is "cell-centered". In contrast, the proposed resolution tree (T_r) assigns values at the "vertices" and the corresponding mesh is "vertex-centered" instead. Figure 4.5 illustrates this difference. Because of this, T_r has the same number of nodes as the number of data samples, while the number of extra nodes for kd-trees, quadtrees and octrees are $1, 1/3$ and $1/7$, respectively, (the overhead is $1/(2^d - 1)$ in general with d being the branching factor of the tree). These overheads affect both storage of the tree as well as progressive streaming of tree nodes.

Beside the difference in topology, there is also a difference in the way node values are computed from the original sample values. From the linear algebra perspective, if the original samples and the tree node values are arranged in vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, then oftentimes $\boldsymbol{y}$ is obtained through a linear transformation of $\boldsymbol{x}$, *i.e.,* $\boldsymbol{y} = A\boldsymbol{x}$, using some matrix A. Because kd, quad, and octrees have more nodes than data samples, the corresponding A matrices are non-square and hence necessarily singular (*i.e.,* the basis is overcomplete). In contrast, the transformation matrix for resolution tree is square and often invertible. This is another way of saying that resolution trees have the same number of nodes as the number of data samples. A large class of such a transform is the *discrete wavelet transform*. Even if A is understood to be a general transformation (not necessarily linear), it is also often invertible as well in the case of resolution tree. I will discuss such transformations in more detail in Section 4.2.

Furthermore, because a resolution tree is "vertex-centered", interpolation is easier. To interpolate the value at a point where there are not enough local fine-level nodes, k-d, quad and octrees require at most 3^d lookups of nearby ancestor nodes per level, with d being the dimension. With a resolution tree, since a node shares the same spatial location with its first child, interpolation requires only 2^d lookups per level, provided that the transformation method used is interpolating, *i.e.,* nodes at the same spatial location share the same value. Figure 4.6 illustrates this difference for the case of a binary tree and a resolution tree in 1D. A similar analysis is given in [160], where the authors use the term "dual tree" for the binary/quadtrees and "primal tree" for the slanted versions.

The last major difference between resolution trees and classical space partitioning trees is related to how approximations to the original field can be formed. With resolution trees, approximations are nicely modeled using the well-established concept of a *cut* [41, 54] (Section 4.1). With classical trees, obtaining an approximation instead corresponds to cutting the tree and discarding all internal nodes, whereas with resolution trees, all nodes above the cut define the approximation. Resolution trees also allow forming more flexible approximations. An octree, for example, only allows decimating all dimensions by 2 on each level, and offer no easy way to decimate different dimensions by different factors. Such capability is important for many types of data where the sampling pattern is "anisotropic", *i.e.,* the dimensions are sampled at different rates. In contrast, resolution trees do not enforce this coupling, and allow arbitrary decimation of the dimensions, totally independent of one another. This is because, on the first level, the different *subbands* branch off the root node to form their own subtrees. Each such subtree corresponds to a single subband type that can be traversed and refined independently. I will discuss this in more detail in Subsection 4.1.7.

4.1.6 Indexing template and generalized precision-resolution hierarchy

Formerly, an *indexing template* is a string made from English letters and colons (:). Each letter represent a dimension of the data, and each is repeated a number of times equal to the number of bits needed to index that dimension. For example, the indexing template for a 3D dataset of dimensions $384 \times 384 \times 256$ will contain 9 x's, 9 y's, and 8 z's in some order, separated by colons *e.g.,* $: yx : zyx : zyx : zyx : zyx : zyx : zyx : zyx : zyx$. This is a generalization of the concept of interleaved coordinate such as Morton code, where I have introduced the colons to signify separations between resolution levels. Whereas Morton codes imply round-robin arrangements of the letters (*e.g.*, $xyzxyz$) to promote spatial locality, an indexing template allows more freedom in arranging the letters to optimize for spatial locality and other trade-offs. Note that for non-power-of-two-dimension grids, "virtual" grid points may be inserted so that the dimensions become powers of two, construct T_r, and then discard the virtual nodes.

I use the term *template part* to refer to each colon-separated part of the full template. Let T denote the very first part of the template, one that comes before the first colon; this part can be empty if the first character in the template is a colon. To form a linear index of a grid point, as before, I interleave its coordinates in all dimensions following the indexing template. Then, I identify the longest sequence of parts at the end that are all 0, and denote it as S. The length of such a sequence (in number of parts) is the resolution level of the grid point. Let F denote the part immediately to the left of this sequence, and P denote the sequence of parts between T and F. Note that none of these sequences include colons, nor do they include English letters – they are sequences of bits. To form the linear index using a function f that acts on the interleaved coordinate, I swap the two sides of F while keeping T intact, namely, $f(\mathtt{T} \mathbin{+\!\!+} \mathtt{P} \mathbin{+\!\!+} \mathtt{F} \mathbin{+\!\!+} \mathtt{S}) = \mathtt{T} \mathbin{+\!\!+} \mathtt{S} \mathbin{+\!\!+} \mathtt{F} \mathbin{+\!\!+} \mathtt{P}$, with $\mathbin{+\!\!+}$ being string concatenation.

This is a generalization of the formula discussed in Subsection 4.1.2, with the main difference being that the resolution levels are not specified by the number of groups of zero m bits, but by the number of groups of zero bits of sizes $m_0, m_1, \ldots$, which are the lengths in characters of the template parts from right to left. The reason T is kept intact is that oftentimes, it is not desirable to construct a hierarchy with one single grid point at the root, so the indexing template supports this idea by stopping the creation of increasingly coarser resolution levels as soon as the last colon from the right is reached. This corresponds to first dividing the whole domain into partitions whose dimensions are determined by the prefix T, then constructing a single tree for each partition, resulting in a global forest. Figure 4.7 gives an example of using an indexing template to compute the linear index for grid points.

A general indexing template allows building a resolution tree by combining different maps. The construction may start with $\mathcal{M}_1$, but for the next level switch to $\mathcal{M}_2$, followed by $\mathcal{M}_3$, and so on. One potential use case is when the input grid size is highly non-uniform, e.g., $32 \times 32 \times 512$, in which case, such interleaving patterns as $: z : z : z : z : zyx : zyx : zyx : zyx : zyx$ may be picked which consists of four levels of $\mathcal{M}_1$ (on the $: z : z : z : z$ part) followed by five levels of $\mathcal{M}_3$ (on the $zyx : zyx : zyx : zyx : zyx$ part). The previous example also serves to highlight the fact that in addition to d, the $\mathcal{M}_d$ maps are also parameterized by the indexing template part used to form Z. Furthermore, when data filtering such as the wavelet transform is used, a mismatch between d and the dimensionality of the data can lead to a T_r in which parent-child relationships between nodes do not reflect the correlations between corresponding coefficients. For example, if a 2D wavelet transform is performed such that the resulting subbands are those that are formed with f_1, a tree built from a map that combines f_1 and h_2 may work better than one built from $\mathcal{M}_1 = (f_1, h_1)$ because the former groups wavelets with the same orientation at different scales.

If d is fixed and the $\mathcal{M}_d$ map is used exclusively across all resolution levels, the resulting resolution tree always have a common property: a node and its parent are always in the same subband type (but with resolution levels differ by 1), except possibly for the root and its children – when $d > 1$, the root splits into $2^d - 1$ children, each with a different subband type. The total number of subband types is therefore always $2^d - 1$. With a general indexing template, however, constructing such a hierarchy with the same property is no longer possible. This is because, unlike for the $\mathcal{M}_d$ family of maps, with a general indexing template, the levels often do not have the same number of subbands, so there is no way to form same-subband parent-child relationships. Instead, the resolution tree is built bottom-up using a recursive procedure: for each resolution level from finest to coarsest, partition the domain into *bricks* of dimensions that match the corresponding template part, build a single resolution tree for each brick, then repeat the process for the set of root nodes for the next coarser level. When the local, per-brick trees are merged, the local connections are kept intact. Figure 4.8 illustrates this process.

The merged tree $\tilde{T}_r^p$ can be considered an approximation to the T_r^p that would have resulted from a non-bricking approach. Evidently, these are different trees, which may produce approximations corresponding to subtrees with different number of nodes for two identical queries. In particular, a region-of-interest (ROI) query can produce a smaller cut on a T_r^p built from merging local trees with bricks of the same size as the query region. One may expect $\tilde{T}_r^p$ to produce approximations with either significantly fewer or significantly more nodes compared to T_r^p, depending on the size of the query's ROI. However, having computed the exact number of nodes in several approximations for both trees, across a wide range of brick and ROI sizes, I observe that the differences are almost always negligible, and only matter somewhat for very small query regions (*e.g.*, less than 20^3 in 3D). The intuition is that the number of nodes in the resulting cut, regardless of tree types, closely traces the size of the query region, and the differences quickly become negligible as the ROI grows. This observation suggests that $\tilde{T}_r^p$ and T_r^p are functionally very similar.

In general, the number of resolution levels contained within a brick can be different (smaller) than what the dimensions of the brick permits. For example, a brick can have dimensions $2^4 \times 2^5 \times 2^6$ but only one resolution level is subsumed, using $\mathcal{M}_2$ along x and y, creating four subbands. Figure 4.9 illustrates this situation from the indexing template point of view. Consider bricks on a resolution level l: the bits of the indexing template are divided into three parts, from left to right: P_l, B_l, and S_l. The suffix S_l consists of l template parts, and is always 0. The intra-brick indexing part B_l has length b (excluding the colons), with b denoting the base-2 logarithm of the size of each brick in number of samples. The bits within B_l index the grid points within a level-l brick, while the preceeding bits of the prefix P_l index the bricks themselves on the same level. Compared to B_l, B_{l+1} (for bricks on the next coarser level) is shifted to the left by L_l, the length of the template part on level l. Because of this, each level-$(l+1)$ brick is parent to 2^{L_l} bricks on level l. Note that in general, a brick can also subsume more than one resolution levels, in which case B_l is shifted by more than one template part at a time.

A frequent type of query in visualization and analysis is to retrieve a region of interest (ROI) in space, which corresponds to retrieving the corrresponding nodes in T_r^p. The most straightforward way to support this capability, in the presence of (potentially variable-rate) compression (which precludes implicit on-disk indexing), is to maintain pointers to the locations of all the nodes on disk. Although such a solution is very costly, this cost may be amortized by storing one pointer for every brick of nodes (or samples) instead. On each level, a brick is a set of $B_x \times B_y \times B_z$ contiguous samples in space. Partitioning the domain into bricks can be done either before or after the construction of T_r^p (and any data transformation). In both cases, such partitioning not only helps with ROI queries but also can speed up the construction of T_r^p, since smaller grids are better suited for caching and parallelization. A brick is a unit of data transformation (*i.e.*, samples within a brick are transformed together); it is a grouping of adjacent samples in space (along all dimensions).

Transforming each brick independently facilitates cache-friendly and parallel data transform and reconstruction, as well as efficient random access.

If brick partitioning is done after data transformation, there can be data dependencies among neighboring bricks, which complicates data reconstruction. For example, if wavelet transform is used, the support of each wavelet basis function may span across brick boundaries, requiring neighboring bricks to recover samples within a brick, thus negatively affecting I/O and decoding time. On the other hand, forming bricks before constructing local hierarchies avoids such issues while further facilitating parallelization during T_r^p construction [147], as bricks are now completely independent. The downsides of this brick-first approach are that a forest of (shallow) local hierarchies is created instead of a single global hierarchy, and the transform coefficients at the brick boundary may be artificially large, resulting in blocky artifacts during reconstruction (I present a solution for the latter in Section 5.1).

Compared to the generalized resolution tree, this hierarchy of bricks is often the more useful structure to work with. This is because the nodes of the resolution tree correspond 1-1 to the grid points, forcing unnatural parent-child relationships when the tree is created through merging, or when the indexing template has varying-length template parts, due to different number of subbands across resolution levels. In contrast, each brick subsumes all the subbands, avoiding the need for the resolution levels to have the same number of subbands. To generate the brick-based hierarchy from the generalized resolution tree, every time a parent node is created, I do not promote one of the children node (except at the leaf level), but rather create a new node parent node and then attach $2^d - 1$ other new nodes around it, one for each subband. This process is demonstrated in Figure 4.10. The new parent node together with the $2^d - 1$ new nodes span form a parent brick on the next coarser level. As before, this tree can still be augmented to add the precision nodes.

When accessing low-resolution data, the indexing template is very important in determining which dimensions are refined first and how the dimensions are refined together. For example, consider a dataset consisting of a number of images stacked on top of one another along the z axis. If z appears toward the end of the template, the refinement pattern favors refining in x and y first before refining in z. In other words, each image slice is refined first before the whole stack is refined. This pattern may or may not be amenable to the actual access patterns during regular usage of the data. For example, slicing the image stack vertically (across all images of the stack) can only be done at finer resolution levels (since z appears only at the fine-level parts of the indexing template). If this access pattern is common, the current indexing template should not be considered since it results in unneccesary data movements. Instead, in this situation, one should consider a template where z appears closer to the beginning of the indexing template. Of course, such a pattern is again suboptimal if the image stack is often accessed one image at a time, since z appearing early in the template means that refinement in z is necessary before further refinements in x and y. In general, the "best" template to use depends heavily on what access patterns are common on the data, and how common each pattern is.

4.1.7 Subband decompositions

In signal processing, a *subband* refers to a frequency band obtained by dividing a signal into different frequency ranges using a set of filters. In 1D, this is achieved using a pair of filters, a *low-pass* filter that extracts a low-frequency subband and a *high-pass* filter that extracts a high-frequency subband. This process is known as *subband decomposition* or subband filtering. The filters are applied to the signal at different scales, resulting in a hierarchical set of subbands with varying levels of detail and resolution. Each subband can be analyzed separately and is characterized by its own frequency range and energy content. The decomposition of a signal into subbands allows for more efficient signal processing and compression, as different subbands may have different levels of importance or relevance to a particular application. The subbands can also be used for various signal processing tasks such as denoising, feature extraction, and classification. In 1D, a pair of filter creates two subbands: one containing low-pass filtered samples (L) and the other high-pass filtered ones (H). In higher dimensions, more subbands may be created (*e.g.,* four in 2D: LL, LH, HL, HH, and eight in 3D: LLL, LLH, LHL, LHH, HLL, HLH, HHL, HHH). To create subbands in higher dimensions, filters are applied to each dimension of the higher-dimensional signal. For example, for a 2D image, a wavelet transform can be used to create subbands in both the horizontal and vertical directions. This results in a set of subbands with varying levels of detail and resolution, each corresponding to a different frequency range in both the horizontal and vertical directions.

Subband decomposition can happen without actual filtering of the sample values (this is also called the *lazy wavelet transform*. The maps $\mathcal{M}_i$ ($i = 1, 2, \dots$) previously introduced can separate the grid samples into subsets, each belonging to a resolution level. On each level, the samples are further separated into subbands, each contains a subset of grid samples on the same level, with each subset spanning the entire spatial domain. Using the map $\mathcal{M}_d$, the subband number is the rightmost group of d-bits that are not all zeros in the interleaved index Z. Since there are d bits in the group, there will be $2^d - 1$ subbands on the corresponding level (excluding bit pattern `0...0`, because this belongs to another level by definition). In general, with an indexing template, the subband number is specified by the part F that immediately follows the suffix sequence S of all-zero parts.

Grid points sharing the same suffix P and the same subband part F are on the same level and in the same subband. All grid points in a subband form a sub-grid, whose first starting point, size, and spacing are completely determined by the indexing template. This is best explained with an example. Assuming the indexing template is $: xyz : xyz : xyz : xy$ and consider the grid points whose interleaved coordinates are of the form $*** : 101 : 000 : 00$. These grid points are on level 2 (from the $000 : 00$ suffix), subband 5 (101 in binary). The first grid point in this subband has interleaved coordinate $000 : 101 : 000 : 00$, which can be deinterleaved using the indexing template to obtain the 3D coordinates $x = 4, y = 0, z = 2$. To determine the spacing of grid points in this subband, since the suffix $S = 000 : 000$ corresponds to the $xyz : xy$ part of the template where x and y each appears twice and z appears once, in between consecutive grid points of the subband, there are $2^2 = 4$ grid points in x, $2^2 = 4$ grid points in y, and $2^1 = 2$ grid points in z. Finally, the size of this subband is determined by the prefix $P = ***$ which corresponds to the xyz prefix of the indexing template: the subband has dimensions $2 \times 2 \times 2$.

To reconstruct an approximation of the data with non-uniform downsampling factors along the dimensions, it is straightforward to determine which subbands to fetch data from. Suppose the downsampling factors create a subgrid G_o (for output subgrid), and the subgrids occupied by the subbands are $G_0, G_1, \dots$. By intersecting each G_i with G_o, it can be determined which subbands should be fetched from; a subband is needed when its subgrid overlaps with G_o, otherwise it can be skipped. Furthermore, a subband may not fully needed; this happens when G_i intersects with G_o but has a smaller spacing along at least one dimension. While it is always possible theoretically to obtain a cut on the (precision-)resolution tree that includes only the required samples on each subband, in practice, the fact that nearby samples are often grouped together for compression purposes and that random access of samples is not trivial when using variable-rate compression, make obtaining such a precise cut much more difficult.

There are many subband decomposition patterns. Figure 4.11 gives two examples: k-d tree-style and quadtree-style nonstandard decompositions, using the $\mathcal{M}_1$ and $\mathcal{M}_2$ maps, respectively. Figure 4.12 contrasts nonstandard and standard subband decompositions, and introduces frequency subband decompositions. The more popular nonstandard decomposition performs transforms in all dimensions one each level, and only recurses on the coarsest subband on each level. In contrast, the standard decomposition performs transforms on all levels along one dimension at a time. The standard decomposition creates more subbands and is more expensive to perform; it also creates "rectangular" basis functions, whereas the nonstandard subband decomposition only creates "square" basis functions. Higher-dimension basis functions are created using Kronecker products of one-dimensional basis functions. The frequency decomposition creates the same number of subbands as the number of samples. For a 1D signal of length 2^n samples, 2^n subbands are created, one for each sample. In higher dimensions, 2^{nd} subbands are created, with d being the dimensionality. The proposed indexing template captures both nonstandard and frequency (but not the standard) subband decompositions. From the indexing template point of view, the frequency subband decomposition is created by repeating letters for the dimensions on each level. Standard and nonstandard subband decompositions are typically created by discrete wavelet transforms (DWT), while frequency subband decompositions are created by orthogonal block transforms such as discrete cosine transform (DCT), Walsh–Hadamard transform (WHT), Slant transform (ST), high correlation transform (HCT), discrete Hartley transform (DHT), or Gram polynomials (GP) [169, 301]. Here, the term *frequency* is to be interpreted as one of actual frequency (in case of the DCT), sequency ("discrete frequency", in the case of the WHT, ST, HCT, or DHT), or polynomial order (in the case of the GP). I discuss such transforms in more detail in the next section.

4.2 Data transformation

Recall that each node in a T_r is associated with a data value, which is a transform coefficient computed from the original grid's sample values. Different transforms/filtering methods can be useful in different cases, depending on how the type of analysis later performed on the data. The most common purpose for data transformation is to decorrelate the original data samples to make the data more amenable to compression, but other purposes, such as to retain topological features, exist. In general, there are three types of transform supported for the proposed precision-resolution tree: wavelet transforms, (data-independent) orthogonal block transforms, and mathematical morphology operators. The identity transform is also an option, in which case the resulting hierarchy is subsampling-based.

Subband transforms are computed by convolution of the input signal with a set of filters and decimating the results. Each decimated signal captures a portion of the frequency spectrum, called a subband. The set of filters is called a filter bank. A general multi-channel filter bank can be implemented by iterating a 2-channel filter bank, creating a cascaded system. A filter bank has two parts: an *analysis* part that decomposes the input signal into multiple signals, one for each subband, and a *synthesis* part that combines the output of the analysis part into a single signal. An important property of such a filter bank is *perfect reconstruction* (PR), *i.e.*, the output of the synthesis part is the same as the original signal. Common designs for perfectly reconstructed filter banks are *quadrature mirror filters* (QMF) and *conjugate mirror filters* (CMF), of which the latter is more useful, since it allows for designing finite PR filters with length greater than 2. If filter coefficients are chosen properly, a iterating a filter bank indefinitely using a pair of CMF gives rise to an orthogonal wavelet system [51], where the input signal is projected into a *multiresolution basis* consisting of orthonormal *scaling* and *wavelet* functions at multiple scales. If the scaling and wavelet functions associated with the synthesis part are different from those of the analysis part, we instead have a *biorthogonal* wavelet system [47], where the synthesis scaling functions are orthogonal to the analysis wavelet functions and vice versa. Biorthogonal wavelets arise from biorthogonal filter banks, with carefully chosen filter coefficients; they can also be computed using the *lifting scheme* [53]. A filter bank gives an $O(N)$ algorithm to compute the discrete wavelet transform (DWT), called the *fast wavelet transform* (FWT). The DWT separates a 1D signal into an *average* (even) half and a *detail* (odd) half. The even part becomes the next coarser resolution level while the odd part becomes the difference between the next coarser resolution level and the current resolution level. To reconstruct the signal, the two parts are combined. It is possible to use any cascaded PR filter bank, and thus any DWT, to compute transform coefficients for the proposed hierarchy. The only two requirements are that the filter bank has perfect reconstruction, and that the transform is nonredundant, meaning the number of transform coefficients are the same as the number of input samples. While not all such filter banks result in a DWT, it is often useful to choose the filters so that they correspond to either an orthogonal or a biorthogonal wavelet system, since the wavelet basis functions help understanding the behavior of iterating such a cascade of filtering and subsampling steps.

Block-based orthogonal transforms can also be viewed as subband transforms. Well-known examples inclue the discrete cosine transform (DCT), Walsh–Hadamard transform (WHT), Slant transform (ST), high correlation transform (HCT), discrete Hartley transform (DHT), or Gram polynomials (GP). Details about these transforms can be found in [43,169, 301]. Computing a DCT on non-overlapping blocks is the same as convolving the image with the block DCT basis functions and then subsampling by a factor equal to the block size. The subbands that result from this are shown in Figure 4.12 (far right). In the case of the DCT (which is really the discrete Fourier transform under a certain periodicity assumption), each subband corresponds to a frequency, since each basis function is a (discretized) cosine wave. For other transforms, frequency is replaced by sequency or polynomial order, but all are understood to refer to different scales at which information is captured. Compared to the WHT, ST, and HCT, the DCT has been shown to be superior for the task of lossy data compression [43], due to the fact that its basis vectors are close to those of the (optimal) Karhunen—Loève Transform's (KLT) of first-order Markov processes when the correlation coefficient approaches unity [44]. As such, DCT is used in the JPEG image compression standard [298]. More recently, Lindstrom [169] shows that these orthogonal block transforms can be unified using a single parameterized matrix for 4-sample block, and proposes a new transform of the same family, based on the Gram polynomials (aka discrete Chebyshev polynomials [33,214]), that achieves better decorrelation effiency and coding gain compared to the rest, including the DCT. All of these orthogonal block transforms can be used in the proposed hierarchical model, noting that they need to be performed on blocks whose dimensions are powers of two. Compared to wavelets, orthogonal block transforms tend to exhibit more blocky artifacts in the reconstruction.

Beside linear filters, another example of subband transforms is nonlinear mathematical morphology filters [230], which are built based on two operators: infimum (min) and supremum (max). An infimum filter replaces two samples (a, b) with, for example, $(\min(a, b), b - a)$, while a supremum filter replaces (a, b) with $(\max(a, b), b - a)$, so that the number of output samples is the same as the number of input samples, while the transform is perfectly reversible. From these basic operators, one can define *erosion* (removal of small structures) and *dilation* (enlargement of structures), which in turns lead to "opening" and "closing" transformations. Compared to linear transforms which tend to "blur" the data, such transformations tend to better retain the shape, size, orientation and connectitivy of features in the data. Repeated applications of the min or max operator removes the fine structures, thus corresponds to low-pass filtering. High-pass filter is achieved by complementing the low-pass information. Thus, such transforms also result in a subband decomposition. Note that to inverse transform a max hierarchy, one needs to encode, for each parent node, which sibling is the largest among its children using d additional bits, assuming the $\mathcal{M}_d$ map. When the task is to detect the presence of isocontour components, as shown in Figure 4.13 for example, averaging filters such as wavelets may not be ideal since they can lead to both false positives and negatives, forcing exploration of the full resolution data to guarantee no missing information. In such cases, a max hierarchy can yield coarse approximations with false positives but no false negatives, and therefore entire empty regions can be skipped since no new features can be created by refining those regions.

4.3 Multi-precision encoding

After a hierarchy is constructed and its samples transformed, the transform coefficients can be compressed. In principle, any compression scheme that operates by bit plane can be used. That is, it is required that the outputs of the compression step are bits that can each be associated with a bit plane. Not all compression schemes fit this requirement; for example, Huffman coding produces bits that cannot be assigned to bit planes. The ones that fit in this framework are typically called bit plane coders; examples are EZW [261], SPIHT [249], SPECK [227], EBCOT [281], ZFP [169], or TTHRESH [12]. "Classical" entropy coders work by adaptively quantizing transform coefficients – typically finer-scale coefficients are quantized more than coarse-scale ones. Ideally, it is desirable to not spend any bits to encode coefficients that are quantized to zero. To achieve this, it is typical for the encoder to send to the decoder some kind of a *significance map* to identify which coefficients are zero. For example, JPEG [298] does this using run-length encoding. This method performs poorly at low bit rates, due to the fact that the significance map always needs to be decoded first before any actual value bits.

Instead of encoding and sending a significance map, most bit plane coder works by the principle of successive approximation quantizer: the transform coefficients are tested against a series of progressively smaller thresholds, each is half the magnitude of the previous. If the coefficients are ordered so that their magnitudes are approximately sorted from high to low, these threshold tests will have longer runs of 0s *i.e.,* once a coefficient is smaller than the current threshold, some subsequent coefficients that are smaller will also be. In the wavelet domain, such an order can be obtained by visiting the subbands from coarse to fine, leading to trees whose roots are in the coarsest subbands. This is the main idea behind the "zero tree" family of coders, including EZW, SPIHT, SPECK and SBHP. Often, during the encoding (or decoding) process, the coefficients are divided into two lists: a *dominant* list consisting of coordinates of coefficients not yet found significant, and a *subordinate* list considting of magnitudes of coefficients already found to be significant. Once a coefficient is found significant, it is moved from the former to the latter list, and its subsequent bits are coded verbatim, since they are more or less random. Instead of zero trees, arithmetic coding can also be used to exploit spatial correlation between same-order bits of nearby coefficients; this is the approach taken by EBCOT [281] in JPEG2000 [264,282]. For each bit plane, EBCOT (Embedded Block Coding with Optimized Truncation) works in three passes: a *significance* pass that identifies the significant coefficients, a *magnitude* pass that encodes the significant coefficients and a *cleanup* pass that encodes the remaining insignificant coefficients. The magnitude pass uses CABAC (Context-Adaptive Binary Arithmetic Coding) [185] which adapts the probability of each symbol (either 0 or 1) to the local statistics. Zero tree based and arithmetic coding based techniques loop over the data multiple times per bit plane, which is costly. Instead, ZFP [169] performs only one pass through each bit plane, by assuming that the signigicant and insignificant coefficients are always perfectly separated by a single coefficient (the last significant coefficient in the bit plane), and simply encodes the position of this coefficient. This scheme trades some compression effectiveness for massive performance increase.

All of the above coders are examples of *set partition coding* [228,229], or more generally, *group testing*, in which the significant coefficients are picked out from the set of not yet significant coefficients using a series of tests against a current threshold. The result is a sequence of bits that together form a bit plane associated with the current threshold, before the threshold is halved and subsequent tests produce the next bit plane. The resulting bit stream has the *embedded* property, *i.e.,* any prefix of it can be decoded to produce an approximation of the coefficients; increasingly longer prefixes result in increasingly better quality (and lower error as each bit plane reduces the reconstruction error by half). This property is sometimes also called *quality-scalable* (as opposed to *resolution-scalalbe*) in the image compression literature. Embedded coders avoid the need to first transmitting a significance map, so they perform well even at low bit rates. Furthermore, downstream processing can work even on partially-decoded bit streams, removing the need to decode everything in advance.

For effective compression, it is necessary to group multiple coefficients and compress them together. In the proposed framework, a block is the unit of compression (*i.e.,* transformed bits in a block are compressed together); blocks are formed by grouping adjacent coefficients. The grouping can happen either in the spatial domain or in the tree domain, in either breadth-first or depth-first order. Grouping in spatial domain is best done when no previous transformation is done (*i.e.,* using the identity transform), so that the data samples have not been decorrelated, and the compression step performs decorrelation on its own. This is the approach taken by ZFP [169]. Grouping in the tree domain is best done after the data has been decorrelated, *e.g.,* using a wavelet transform. In the tree domain, breadth-first grouping relies on exploiting redundancy among neighbor coefficients on the same subband. JPEG2000 [264] follows this approach. On the other hand, depth-first grouping [249] exploits local properties that relate ancestors with descendants nodes across subbands at corresponding spatial locations. In particular, the ancestor nodes tend to be larger in magnitude, resulting in bit planes that are conducive to compression since they contain runs of zero bits for the nonsignificant coefficients. Different ways of grouping may be preferred for better compression, but may also preclude certain types of queries due to undesired couplings across bit planes and/or resolution levels, since the blocking method may affect the size of the resulting subtrees corresponding to approximations produced for certain query types. As an example, for query that needs more precision than resolution, depth-first (by resolution) and grid-domain (by space) blocking are not desirable since they tend to conflate samples across resolution levels.

In the proposed framework, the transform coefficients in each brick are partitioned into *blocks* and compressed as blocks. A block can also be as large as the brick itself. Note that this blocking behavior can be modeled directly in T_r^p by letting each node represent a bit plane from not a single but a group of samples. A brick is then a collection of contiguous blocks. A block, when encoded, produces multiple bit planes. Each such bit plane within a block is encoded and decoded as one unit, *i.e.,* from the perspective of reconstructing an approximation from a tree cut, a unit cannot be partially decoded. However, a block can be partially decoded: the bit planes of the block can be decoded at different times and not necessarily all at once, even though at encoding time they were produced together in one sequential process. In practice, this means that the decoder has to store certain decoding states to be able to resume at a later point. It is also possible to use different encoding schemes for different blocks, which would typically require keeping a certain amount of metadata to indicate the scheme for each block. However, it is reasonable to expect that, in practice, the total number of schemes would be small. Thus, the cost of storing this metadata would be quite minimal and well amortized over the size of a brick (*e.g.,* 64^3 coefficients) or a block (*e.g.,* 4^3 coefficients).

In terms of indexing, suppose that the exponents for all transform coefficients range from $2^{e_{min}}$ to $2^{e_{max}}$, the indexing template for the compressed bits of T_r^p can be formed by splicing the bits representing the bit plane (a number between e_{min} and e_{max}) with Z, the indexing template for samples in the spatial domain. The combined index thus gives an address for every bit in the encoded dataset. This index can further be partitioned into blocks or files, which facilitates data lookup in a wide range of underlying storage architectures, from traditional file systems to modern object storages on cloud platforms.

4.4 Linear storage model

The compressed blocks must be sorted in some order to be serialized to disk. Each block has a spatial index I_s (obtained by traversing the resolution part of the tree in some order) and a bit plane index I_b. Let us consider an ordering scheme where the bits of I_s and I_b are interleaved following some pattern. For example, if I_b is put after I_s, let the bit planes for a block be stored contiguously on disk. On the other hand, if I_b is inserted in between the bits of I_s, a bit plane will span multiple blocks on disk before moving to the next bit plane. Depending on how the expected access pattern is, one choice may be more preferred than the other.

In practice, disk I/O usually happens in big chunks of bytes, which I model using the concept of a *chunk*, the smallest unit of I/O. A chunk can be defined to have either a fixed size in bytes or a fixed extent in some space. Storage of chunks into files also has implications for performance. A large file can slow down chunk lookup (as there are too many chunks) and reduces parallelism during file I/O due to potential data races caused by multiple threads writing to the same file, especially in a distributed setting [148]. On the other hand, having too many small files puts pressure on the file system and increases the amount of metadata fetched at read time. In the proposed framework, two parameters control these trade-offs: the number of chunks in a file and the ordering of chunks. An optimized chunk ordering can minimize disk seek latency and take advantage of the operating system's prefetching.

A chunk is out unit of I/O (*i.e.,* bits in a chunk are retrieved together), to model real-world block-based I/O devices such as a disk. Like a block, a chunk is also grouping of bits in the same 2D space where bits are grouped for compression, but the grouping is done at block boundaries (since bits within a block need to be decompressed together). Chunks are themselves organized in files. How chunks are mapped to files have significant performance implications, as it affects chunk lookup time, fetching time (considering both disk seek latency and prefetching), parallel I/O, and file system overheads.

It is also important to consider indexing for random access of levels and bit planes, as well as of blocks, bricks, chunks, and files. If few chunks are included in a file, a simple array of IDs and file offsets would suffice for random access of chunks. When the number of chunks per file is large, however, a B-tree [15] could perform better. Likewise, whether a file can be quickly accessed depends on the number of files per directory, which, if too large, is a bottleneck during file lookup. All such choices are parameters to the proposed data model.

In summary, I have proposed a family of hierarchical data models with various free parameters. Beside the choices for the mapping and bit interleaving pattern when constructing the precision-resolution tree, design choices can be made for all of the above considerations, ultimately describing a family of physical hierarchies. Certain parameter choices lead to known data representations in the literature. For example, IDX [223] uses one brick the size of the entire grid and builds one tree using $\mathcal{M}_1$ with no data filtering. ZFP [169] builds no tree (0 levels), uses grid-based blocking, and encodes each block using the ZFP encoder. JPEG2000 [264] uses the term *tiles* to refer to my bricks, filters data with wavelets, uses depth-first blocking and refers to each block as a *code block*, and encodes each block using its own EBCOT [281] encoder. Finally, the recently proposed system in [188] also defines blocks, bricks, and pages, with semantics that map very well to my blocks, bricks, and chunks. Certain combinations can be intriguing, such as encoding JPEG2000's individual resolution levels using ZFP, or using the ZFP encoder as a lightweight replacement to EBCOT to encode wavelet coefficients.

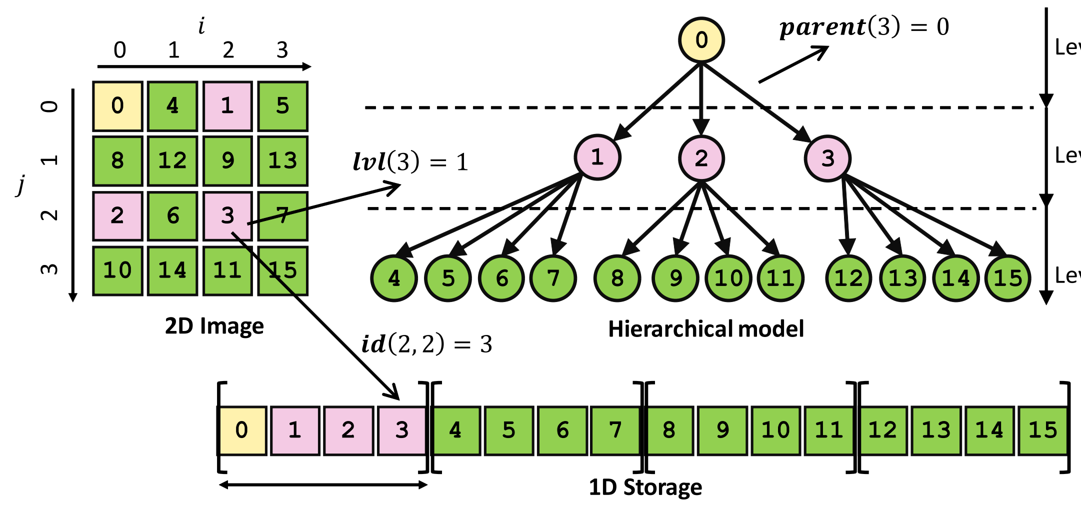
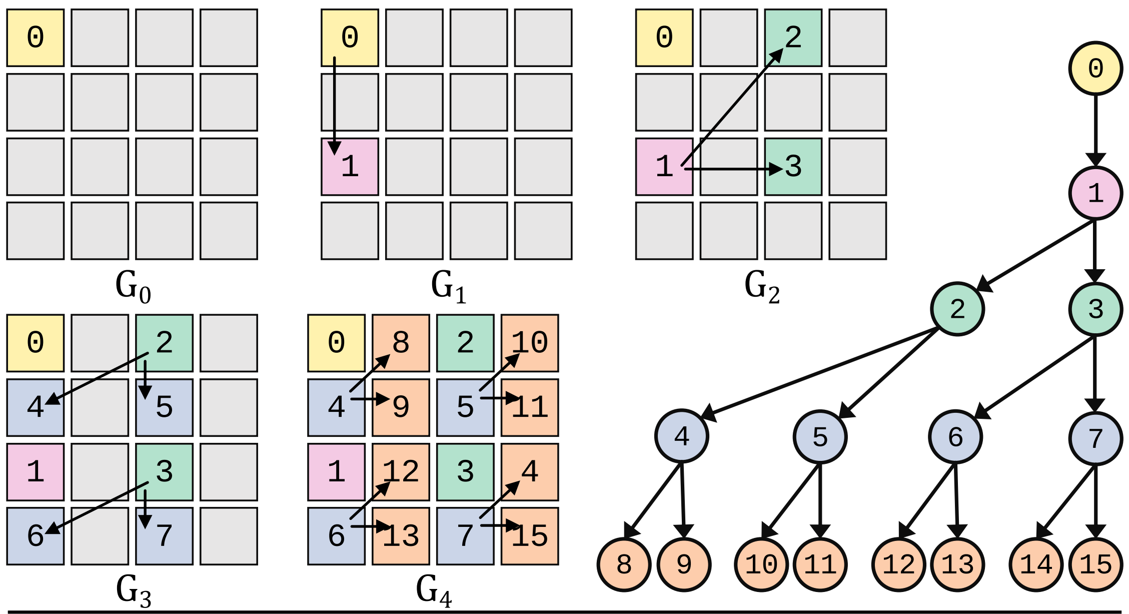
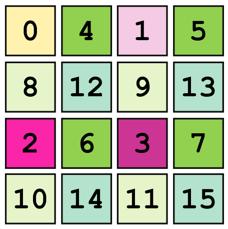


Figure 4.1: Three main views of a discrete scalar field: image, tree, and linear storage. Three functions, *lvl*, *id* and *parent*, related these three views. There are many ways to define such functions. In this example, the discrete field is a 4×4 2D image, the sample at coordinates $(2, 2)$ is at tree level 1, its linear index is 3, and its parent on the tree has index 0.

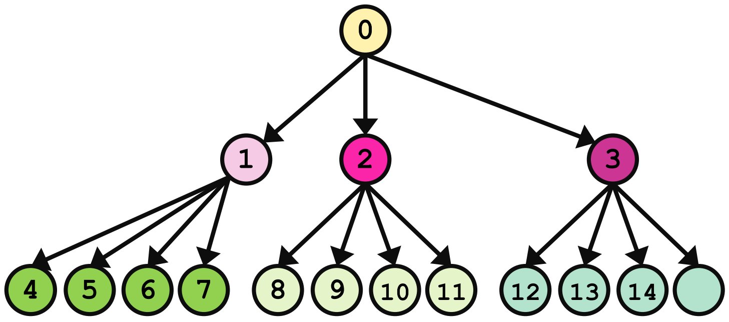


(x,y)	$Z(x,y)$	P ⧺ 1 ⧺ S	S ⧺ 1 ⧺ P	g	f	h
$(0,0)$	0000: 0	⧺ ⧺ 0000	0000 ⧺ ⧺	0	0	-
$(1,0)$	0001: 1	000 ⧺ 1 ⧺	⧺ 1 ⧺ 000	4	8	4
$(2,0)$	0100: 4	0 ⧺ 1 ⧺ 00	00 ⧺ 1 ⧺ 0	2	2	1
$(3,0)$	0101: 5	010 ⧺ 1 ⧺	⧺ 1 ⧺ 010	4	10	5
$(0,1)$	0010: 2	00 ⧺ 1 ⧺ 0	0 ⧺ 1 ⧺ 00	3	4	2
$(1,1)$	0011: 3	001 ⧺ 1 ⧺	⧺ 1 ⧺ 001	4	9	4
$(2,1)$	0110: 6	01 ⧺ 1 ⧺ 0	0 ⧺ 1 ⧺ 01	3	5	2
$(3,1)$	0111: 7	011 ⧺ 1 ⧺	⧺ 1 ⧺ 011	4	11	5
$(0,2)$	1000: 8	⧺ 1 ⧺ 000	000 ⧺ 1 ⧺	1	1	0
$(1,2)$	1001: 9	100 ⧺ 1 ⧺	⧺ 1 ⧺ 100	4	12	6
$(2,2)$	1100: 12	1 ⧺ 1 ⧺ 00	00 ⧺ 1 ⧺ 1	2	3	1
$(3,2)$	1101: 13	110 ⧺ 1 ⧺	⧺ 1 ⧺ 110	4	14	7
$(0,3)$	1010: 10	10 ⧺ 1 ⧺ 0	0 ⧺ 1 ⧺ 10	3	6	3
$(1,3)$	1011: 11	101 ⧺ 1 ⧺	⧺ 1 ⧺ 101	4	13	6
$(2,3)$	1110: 14	11 ⧺ 1 ⧺ 0	0 ⧺ 1 ⧺ 11	3	7	3
$(3,3)$	1111: 15	111 ⧺ 1 ⧺	⧺ 1 ⧺ 111	4	15	7

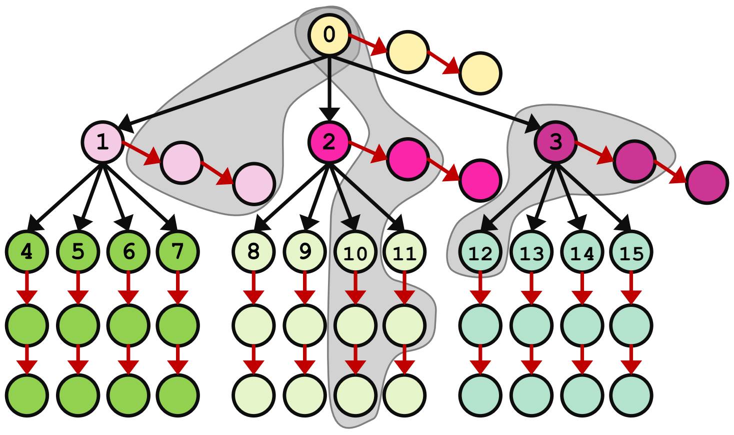
Figure 4.2: Illustration of a resolution tree, T_r, for a 4×4 grid and parameter $d = 1$, which leads to a branching factor of at most 2. The figure tabulates the Z-indices and the proposed bit manipulations thereof, leading to the functions f, g, and h, used for constructing the tree. The sequence of grids G_i shows the nested sub-grids arising from the top-down traversal of the tree. The nodes of the tree and the corresponding grid points are labeled by their indices (the output of f) and colored based on their levels (the output of g).



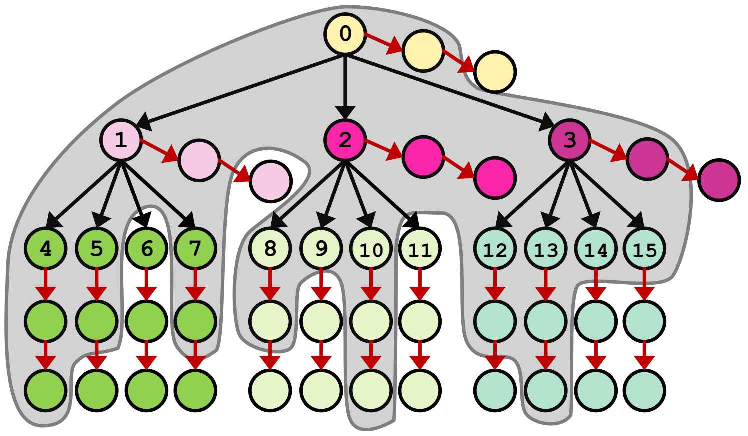
(a) Grid



(b) Resolution tree



(c) Precision-resolution tree with 3 invalid cuts



(d) Precision-resolution tree with a valid cut

Figure 4.3: The input is a 4×4 grid (a) from which a T_r (b) is constructed using $d = 2$. Then, T_r is extended to form T_r^p (c) by adding nodes representing bit planes (the blank nodes) together with necessary edges (in red). The shaded regions correspond to three invalid cuts in (c) and one valid cut in (d). The three cuts in (c) are invalid since each shaded region intersects at least one path from the root to a leaf more than once.

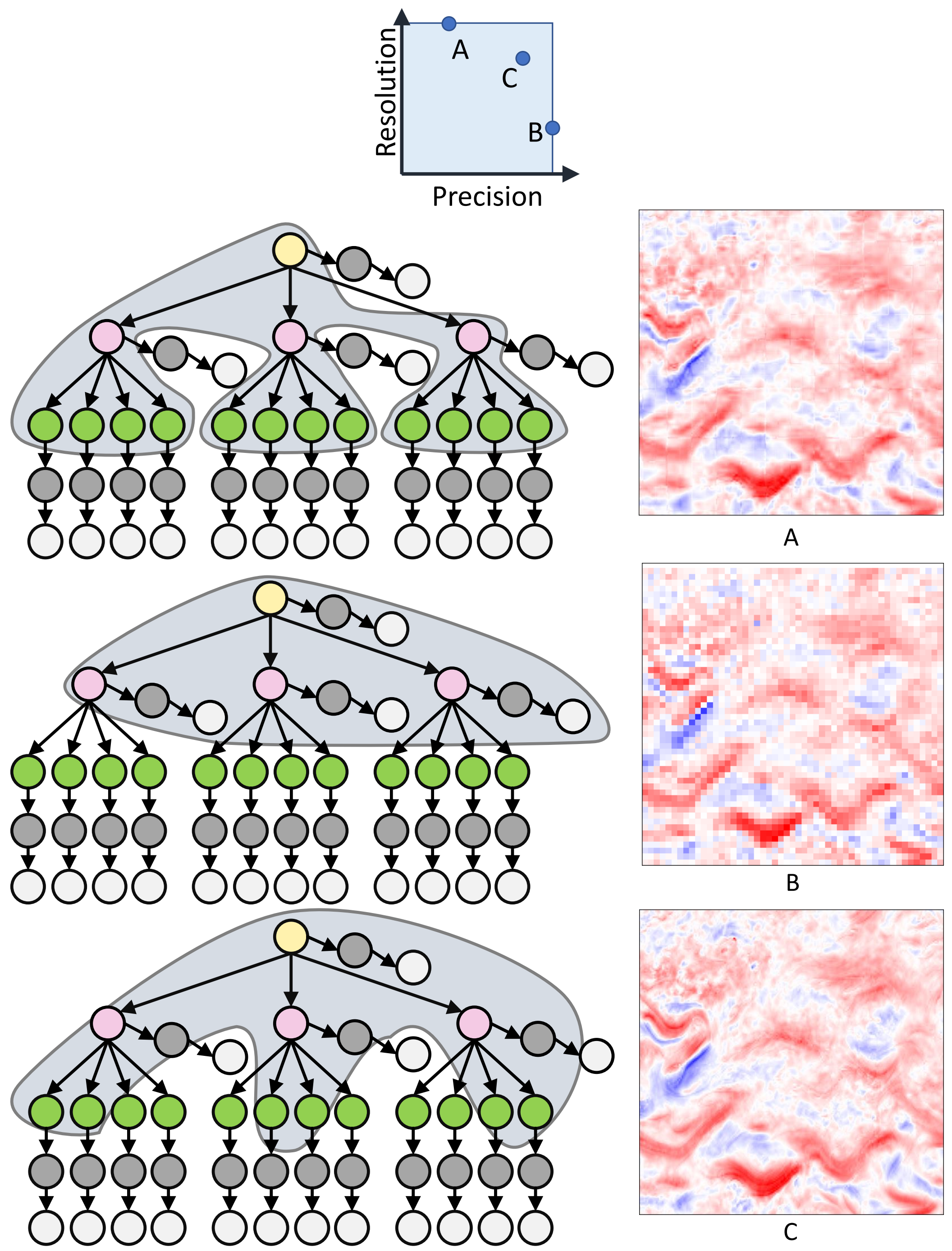


Figure 4.4: Three different cuts of the same precision-resolution tree and the corresponding approximations. Approximation A has high resolution but low precision. Approximation B has high precision but low resolution. Aprroximation C has middle precision and middle resolution. Any reasonable approximation in the precision-resolution space can be realized by a cut of the precision-resolution tree.

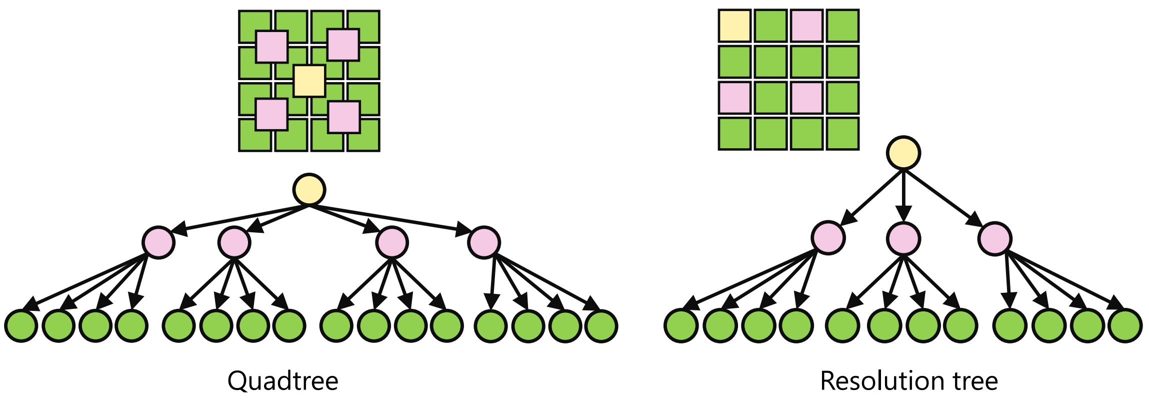


Figure 4.5: Quadtrees put each inner node at the center of its children's bounding box, while resolution trees put it at the location of its first child. In this way, resolution trees do not have extra nodes compared to the number of samples.

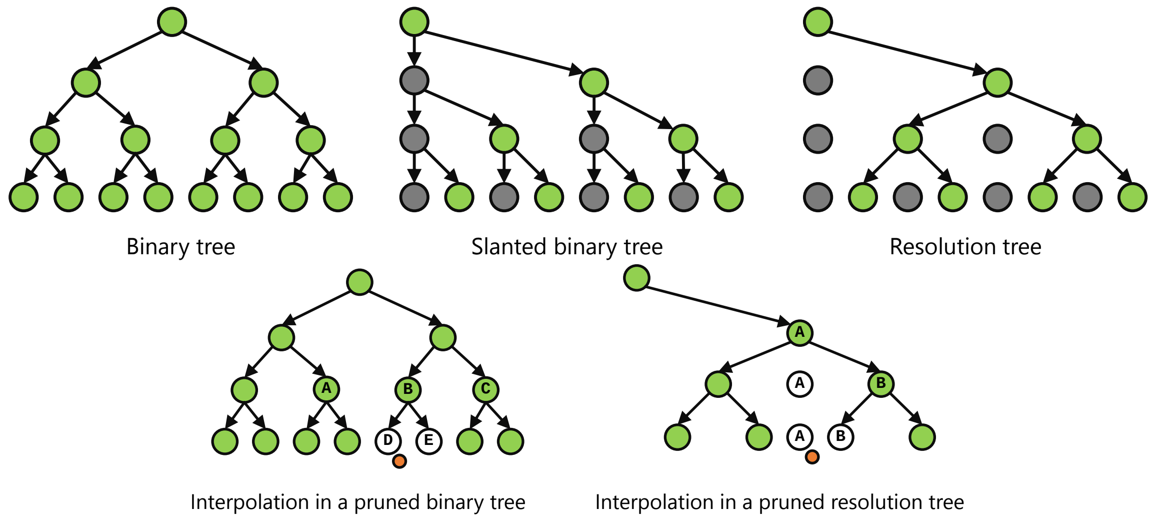

Binary tree
Slanted binary tree
Resolution tree
A
B
C
D
E
A
A
B
A
B
Interpolation in a pruned binary tree
Interpolation in a pruned resolution tree

Figure 4.6: The proposed resolution tree can be obtained from "slanting" a binary tree in 1D, then fixing the edges. Similar slanting operations on quadtrees and octrees work in higher dimensions. In terms of interpolation from a pruned tree, the binary tree requires 3 lookups of previous-level nodes (to interpolate at the orange point, nodes D and E are needed, which in turn need A, B, and C). In contrast, the resolution tree requires only 2 lookups (nodes A and B), provided that the data filtering is interpolating, *i.e.,* a finer-level sample can be directly copied from coarser-level one at the same spatial location.

(a) Indexing template	`ttttttttttddd:dyx:dyx:dyx:dyx:tyx:tyx:tyx:tx:yx:yx:yx:yx:yx:yx`
(b) Interleaved coordinate 1	`*************:***:***:***:***:***:***:***:**:**:**:**:**:10:00`
(c) Linear index 1 (level 1, subband 2)	`*************:00:10:***:***:***:***:***:***:***:**:**:**:**:**`
(d) Interleaved coordinate 2	`*************:***:***:***:***:101:000:000:00:00:00:00:00:00:00`
(e) Linear index 2 (level 9, subband 5)	`*************:000:000:00:00:00:00:00:00:00:101:***:***:***:***`

Figure 4.7: An example demonstrating the use of an indexing template. (a) is the indexing template, for a four-dimensional dataset (the dimensions are denoted using the letters `x,y,d,t`). (b) is an interleaved coordinate for some grid point, following this template (the * character stands for either 0 or 1). (c) is the linear index of the grid point in (b). (d) is another interleaved coordinate, and (e) is the linear index of the grid point in (d). The fixed part (T) is in green, the zero suffix (S) is in orange, the subband part (F) is in black, and the prefix part (P) is in blue.

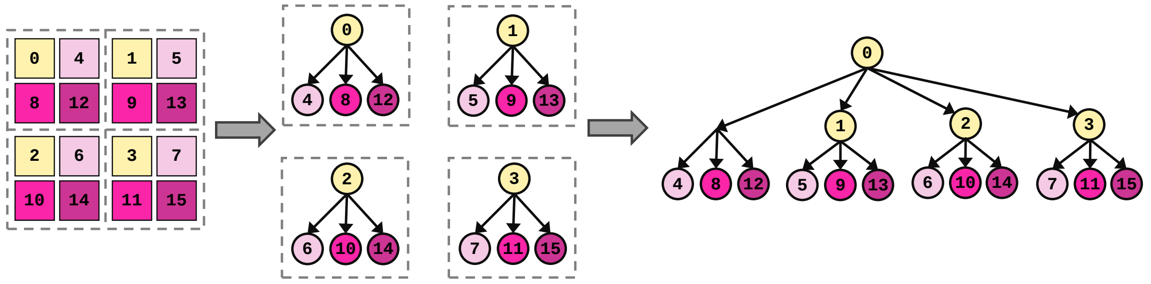


Figure 4.8: A global tree can be constructed by merging local trees built independently from each brick. The merging is done by generating a tree that spans all the local roots ($\{0, 1, 2, 3\}$), while keeping the local edges intact. In general, when using an indexing template where the template parts are different, a resolution tree can only be created through repeated merging. Note that even when the template parts are identical and a resolution tree can be built using maps $\mathcal{M}_d$, this tree is different from the tree obtained by repeated merging.

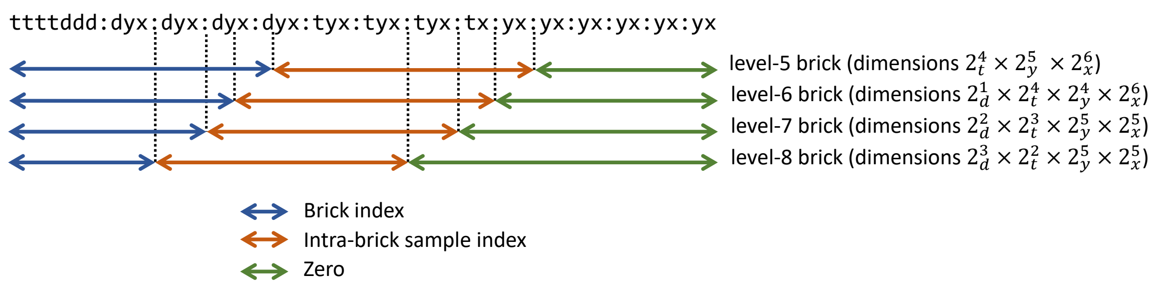

ttttddd:dyx:dyx:dyx:dyx:tyx:tyx:tyx:tx:yx:yx:yx:yx:yx:yx
level-5 brick (dimensions $2^4_t \times 2^5_y \times 2^6_x$)
level-6 brick (dimensions $2^1_d \times 2^4_t \times 2^4_y \times 2^6_x$)
level-7 brick (dimensions $2^2_d \times 2^3_t \times 2^5_y \times 2^5_x$)
level-8 brick (dimensions $2^3_d \times 2^2_t \times 2^5_y \times 2^5_x$)
Brick index
Intra-brick sample index
Zero

Figure 4.9: Bricks from the indexing template point of view. In this example, the indexing template covers four dimensions: time (t), depth (d), horizontal axis (x), and vertical axis (y). On each level l, the indexing template is partitioned into three parts: P_l (brick index), B_l (intra-brick sample index), and S_l (zero). A level-l brick is parent to 2^L_{l-1} level-$l-1$ bricks, with L_{l-1} being the length of the $l-1$-th indexing template part from the right. Note also that while the number of samples for each brick is fixed, a brick's dimensions and dimensionality are not fixed, but are determined by the substring of the indexing template spanned by B_l, as demonstrated here.

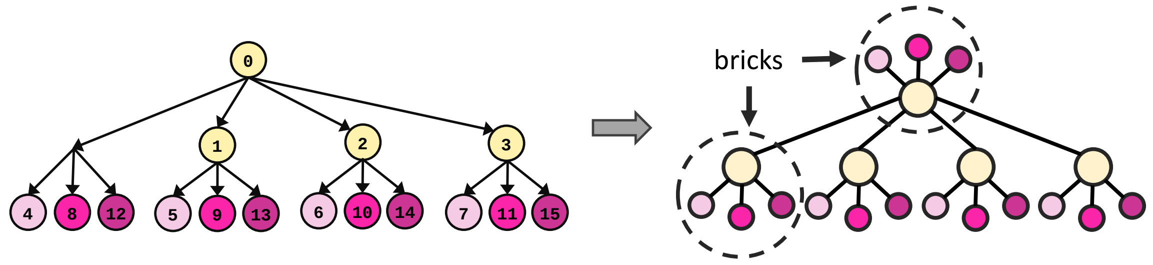
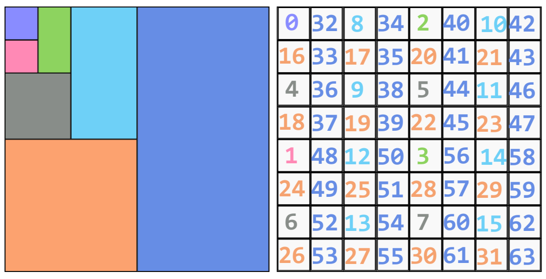
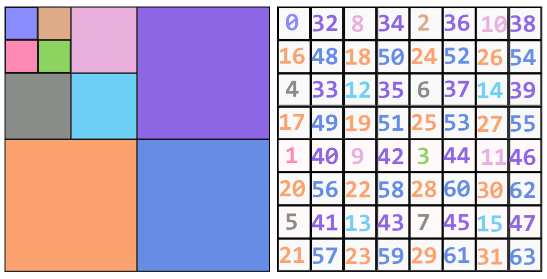


Figure 4.10: Creating a brick-based tree from a generalized resolution tree. To do so requires not promoting existing nodes to the coarser level, but creeating new nodes that together form a brick consisting of all subbands at the coarser level. This results in a tree with a more natural shape compared to the generalized resolution tree.



0	32	8	34	2	40	10	42
16	33	17	35	20	41	21	43
4	36	9	38	5	44	11	46
18	37	19	39	22	45	23	47
1	48	12	50	3	56	14	58
24	49	25	51	28	57	29	59
6	52	13	54	7	60	15	62
26	53	27	55	30	61	31	63

(a) K-d tree-style subband decomposition



0	32	8	34	2	36	10	38
16	48	18	50	24	52	26	54
4	33	12	35	6	37	14	39
17	49	19	51	25	53	27	55
1	40	9	42	3	44	11	46
20	56	22	58	28	60	30	62
5	41	13	43	7	45	15	47
21	57	23	59	29	61	31	63

(b) Quadtree-style subband decomposition

Figure 4.11: Indexing templates create different subband decompositions. Here, I show (a) K-d tree ($\mathcal{M}_1$) and (b) quadtree-style ($\mathcal{M}_2$) subband decompositions, for an 8×8 grid. The grid points are numbered by their linear index, and colored by their subband. The k-d tree-style decomposition is done with the indexing template $: y : x : y : x : y : x$ nd the quadtree-style decomposition is done with the template $: yx : yx : yx$.

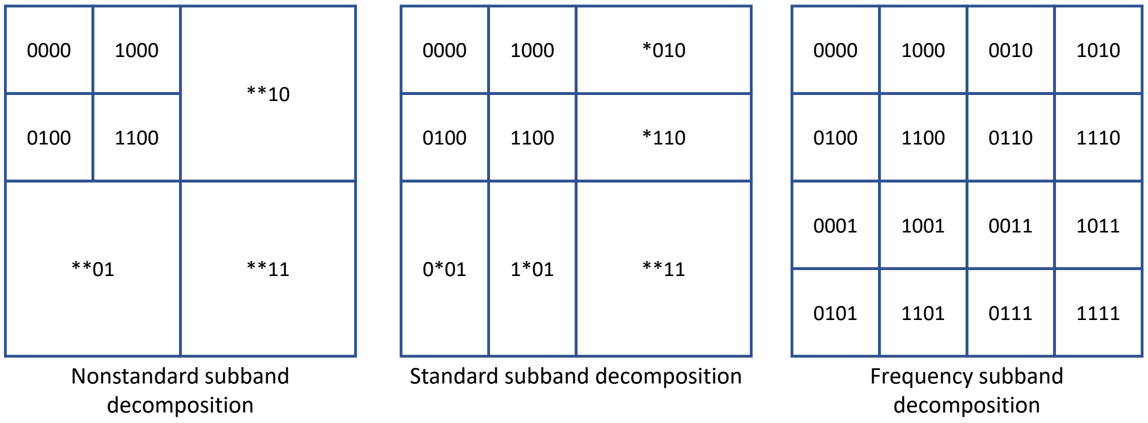


Figure 4.12: Different subband decompositions. Nonstandard and standard subband decompositions [270] are common for wavelet transforms, whereas frequency decomposition can be created using trigonometric (*e.g.*, DCT) and other orthogonal transforms. The subbands are numbered by instances of the indexing template, with $*$ denoting either 0 or 1 (bit).

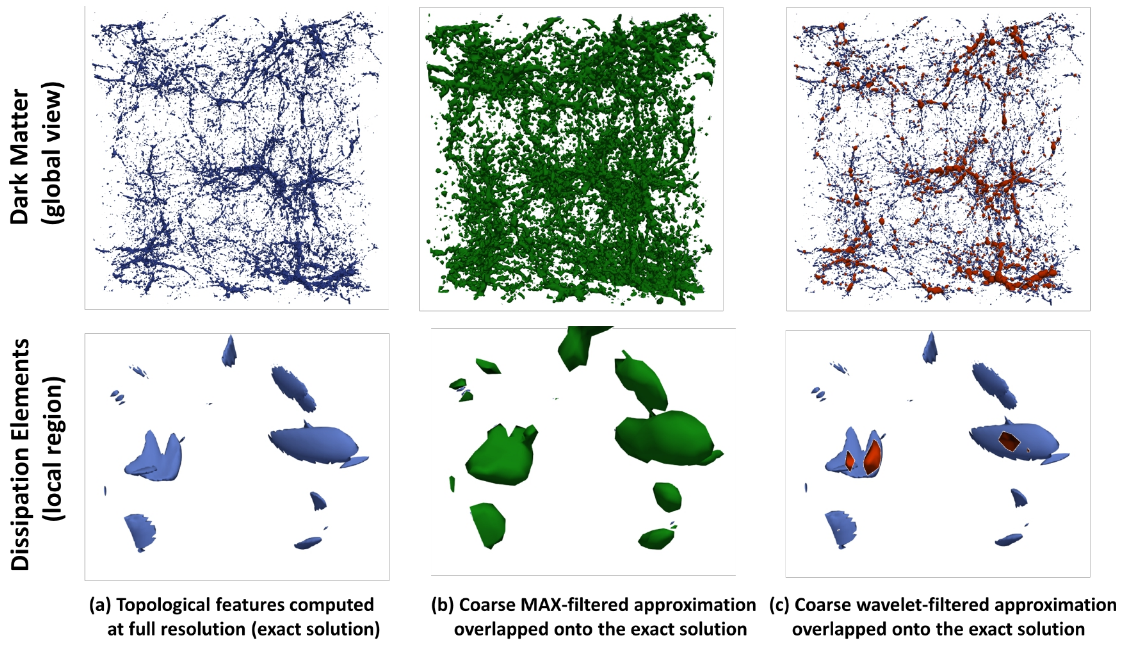

Dark Matter (global view)
Dissipation Elements (local region)
(a) Topological features computed at full resolution (exact solution)
(b) Coarse MAX-filtered approximation overlapped onto the exact solution
(c) Coarse wavelet-filtered approximation overlapped onto the exact solution

Figure 4.13: A demonstration of the need for flexibility in the filtering operator. (b) the max-based approximation in green better preserves the features (blue) than (c) the wavelet-based approximation in red. The max operator tends to grow the components and potentially merge neighboring components, whereas wavelets tend to average away components.

CHAPTER 5

COMPACT ENCODINGS AND LAYOUTS FOR PROGRESSIVE QUERIES

This chapter presents a system that implements a member of the proposed family of data layouts with concrete parameter choices and techniques to handle real-world data. As mentioned earlier, most reduction techniques support only one way to refine data, *i.e.,* as a progression in either precision or resolution. Nevertheless, it is unclear whether such individual schemes can be combined sensibly into a 1D progression navigating the 2D precision-resolution space. In comparison, the proposed approach does not impose any progression order on data chunks themselves, except for when valid cuts are concerned. Rather, I advocate for a database-inspired approach at system level, such that the data layout is designed to best serve chunks from the 3D precision-resolution-sample space in *any* order demanded by the analysis task at hand.

The system works in three steps: data transformation (filtering), data encoding (compression), and data chunking (and I/O). For data filtering, the domain is partitioned into bricks, and each brick is transformed independently. To create multiple resolution levels, transform coefficients in the coarsest subband are merged to form bricks on the next coarser level, and the process repeats until the desired number of resolution level is reached. After the whole transformation process, each brick consists of a number of subbands. Each subband is further partitioned into blocks, and each block is compressed independently, creating multiple data packets, one for each bit plane per block. The data packets are written to in-memory channels, each of which correspond to a unique combination of bit plane and subband (and resolution level). Then, each channel is dividied into chunks, which are the system's unit of I/O. Finally, a chunk is written to disk once it is fully encoded. Figure 5.1 depicts this whole process. I next discuss each stage in more detail.

5.1 Decorrelation transformation

5.1.1 Per-brick wavelet transform

The hierarchy is built from the bottom up in a depth-first manner, with data transformation on each level performed in two steps. In the first step, the map $\mathcal{M}_3$ is performed within each brick for one pass in each dimension. Then, the coefficients in the coarsest subband are copied into corresponding parent bricks of the next (coarser) level. Typically each brick on level l is the parent to 2^3 bricks on level $l+1$ in 3D. When a parent brick on the coarser resolution level is fully formed, the process is repeated. The recursion stops when the desired height for the global tree(s) is reached. Once a brick is transformed, the resulting coefficients are encoded and then the brick is discarded from memory since they are no longer needed. This procedure is illustrated in Figure 5.2.

It is best to move data as soon as possible (to avoid buffering too many bricks in memory), which can be achieved by a depth-first traversal of the set of bricks on all levels. Therefore, bricks are visisted following a Z index created by interleaving the bits of each sample's spatial coordinates. Each Z index consists of a suffix (of length $\log_2(B_x B_y B_z)$ bits) that corresponds to a local coefficient index within the brick, and a prefix (the rest of the bits) to indicate the index of the brick itself. The interleaving pattern for Z is determined by first fixing a pattern for the suffix, and then repeating it as many time as possible for the rest of Z. Intuitively, bricks at level $l-1$ are formed by treating each level-l brick as a single sample. This strategy ensures a 2D brick stays on the same 2D slice in the next coarser level, for example.

For data filtering, the CDF 5/3 multilinear wavelets [47] are used, due to their simplicity and effectiveness in compression. Furthermore, multilinear interpolants are at the foundation of many visualization techniques [11, 41, 180, 208]. The wavelet transform is performed in bricks of size $B_x \times B_y \times B_z$ samples, where $B_{x,y,z} = 2^{k_{x,y,z}}$ for $k_{x,y,z} \geq 0$. The lifting approach [53] is used to compute the wavelet transform: $\hat{f}_{2i+1} = f_{2i+1} - \frac{1}{2}(f_{2i} + f_{2i+2})$ (w-lift) and $\hat{f}_{2i} = f_{2i} + \frac{1}{4}\left(\hat{f}_{2i-1} + \hat{f}_{2i+1}\right)$ (s-lift) where f_i denotes the input sample value at index i and $\hat{f}$ the wavelet (odd-indexed) and scaling (even-indexed) coefficients. Higher dimensional wavelet basis functions are formed by tensor products of the 1D basis functions. The CDF 5/3 basis functions are visualized in Figure 5.3.

5.1.2 Linear-lifting extrapolation

Encoding bricks independently of each other is key to handling out-of-core data sets, because the data can be processed one brick at a time during compression and reconstructed at brick level during decompression. Because the wavelet basis functions span across brick boundaries, discontinuity artifacts often appear in reconstructed data if transform is naively restricted to individual bricks. To largely reduce this effect, I use *lifting-based linear extrapolation* [25] to extrapolate each brick with one extra sample in each dimension. Such an extrapolation ensures zero-valued wavelet coefficients in all dimensions at the boundary during the forward transform, resulting in good compression. Halo exchange to exchange the boundary values among neighboring bricks could have been used; however, such an approach introduces synchronization points and prevents bricks to be encoded and decoded truly independently of one another. Conceptually, the usual lifting steps are performed everywhere in an extended domain of size $(2^L + 1)^d$, but assign values lazily to grid points outside the original domain.

Denoted symbolically as `(*-*-*)`, the w-lift step updates the value at the center using the adjacent ones. With respect to the original and extended domains, there exist four possible scenarios: `(x-x-x)`, `(x-x-o)`, `(x-o-o)`, and `(o-o-o)`, where `x` represents a grid point within the original domain whereas `o` has an uninitialized value due to being outside the original domain (but within the extended domain). In the first case, all three relevant grid points exist within the original domain and the standard w-lift can be applied. In the second case, the two known values are linearly extrapolated to assign a value to the rightmost grid point when needed, resulting in a zero-valued wavelet coefficient. For the third and the fourth case, the wavelet coefficient is set to zero but setting the value for the rightmost sample is deferred to an extrapolation step on some coarser level. Furthermore, s-lift is applied only to values that have been initialized — standard step for the first case, but no effect for the remaining three. With this scheme, uninitialized grid points are given values such that when they are used for lifting, the resulting wavelet coefficients are always zero. Note that the same technique can also extrapolate bricks with dimensions less than $B_x \times B_y \times B_z$ (those at the domain boundary) to $(B_x + 1) \times (B_y + 1) \times (B_z + 1)$, provided that $B_{x,y,z}$ are powers of two. Table 5.1 illustrates this approach using a concrete 1D example. Figure 5.4 demonstrates the effectiveness of linear-lifting extrapolation method for avoiding boundary discontinuities.

This scheme differs from simple linear extrapolation in that it interleaves the potential linear extrapolation steps described above with lifting steps. Simple linear extrapolation does not ensure smoothness along dimensions orthogonal to the one being extrapolated, while also failing to correct for the nonlinear reconstruction introduced by s-lift steps. In contrast, by interleaving lifting steps with extrapolation steps across the hierarchy and across spatial dimensions, my method ensures smoothness in the extended function both across the boundary of the domain as well as across all dimensions (see Figure 5.4). Naive linear extrapolation is also entirely local as it depends only on the last two values near the domain boundary, whereas my method extrapolates at multiple scales and therefore is globally influenced.

In the example from Table 5.1, the extrapolated function is longer than the original function by three elements unlike the traditional wavelet transform, where the two functions have the same length. However, in practice, the inverse lifting steps are never performed, and thus the extrapolated function is never explicitly computed or stored. The potentially extrapolated value at each lifting step needs to be stored to support perfect reconstruction, but this requires storing at most a single extra value along each dimension of the original domain, as the same slot can be reused on the next coarser transform level without compromising reconstruction using inverse lifting steps. Because of this, bricks of size $(2^L)^d$ are only extended to bricks of size $(2^L+1)^d$, regardless of the number of transform levels. In 3D, bricks of size 64^3 are extended to 65^3, for approximately a 5% overhead.

5.2 Compression of exponents and significant bit planes

5.2.1 Block-based compression

I use a breadth-first blocking approach limited to each wavelet subband to group coefficients into blocks of size 4^3 and encode each block with ZFP [169], with modifications by me to suite the purposes of this work. ZFP is a fast, block-based compressor that transforms and encodes every 4^d block independently. The transform is based on the Gram polynomials (also known as discrete Chebyshev polynomials [33, 214]), with a slight change made to the basis to make the transform efficiently implementable using only shifts and adds, causing the transform to be slightly non-orthogonal. After transformation, the resulting coefficients are ordered by sequency [301], ready to be coded. ZFP uses an embedded bit plane coder, in which each bit plane is coded one at a time, starting from the most significant one. A simple group testing algorithm is used where a bit plane is divided into a significant prefix and a non-significant suffix. For any bit plane, the suffix is initially that of the previous bit plane (which is the whole bit plane at the very beginning), but gradually narrowed down by testing for significance one bit at a time, until it consists of all 0 bits. This algorithm is very fast because there is no need for complicated book keeping, unlike the traditional bit plane coders such as SPIHT [249] or SPECK [227].

For this work, a block is restricted to a single subband to avoid coupling the different subbands during decoding (*i.e.,* so that the refinement in resolution is more fine-grained). Note that due to the extrapolation discusses earlier, near the boundary of a subband, a block no longer has dimensions 4^3 but can be reduced to 4^2 or 4^1, and such "partial" blocks are compressed by 2D and 1D ZFP, respectively. Every combination of subband and bit plane writes to a different stream. The value bits across blocks are aligned (shifted) to the same exponent, and equivalent bit planes across blocks are written to the same stream.

5.2.2 Transposition of bit planes

My implementation of ZFP has a few modifications to improve the decoding performance. The default ZFP decoder essentially has two parts: (1) read one bit at a time and decide what to do depending on whether this is a group test or value bit, and (2) "scatter" (or transpose) the value bits back to memory. Based on my profiling, part 2 (bit transposition) is in many cases the bottleneck, due to its involvement of data movements to and from the CPU and not just arithmetic operations. To speed up the bit transposition, I use two approaches: (1) utilizing AVX2 instructions to transpose more bits per instruction, and (2) a recursive transposer which performs more transposition steps per memory access. The AVX2 approach works with 256-bit registers, or eight 32-bit integers at the same time. The idea is to repeatedly use 8-bit chunks of the 64-bit input as a mask to load eight 32-bit integers from a block, perform the addition, then store the result back to memory. This results in 2x improvement in speed for the decoder.

The recursive transposer, on the other hand, partitions a bit matrix (*e.g.,* 64 bit planes of 64 coefficients) into four (2×2) equal-size blocks, and then recursively transpose these four blocks, with rows of submatrices being processed in parallel. Transposed bit planes are output as soon as the corresponding submatrices have been transposed, and the recursive process can end prematurely if the bit planes being decoded have been transposed. The top two submatrices are transposed first, and once that is done their corresponding bit planes can be encoded. The bottom two submatrics needs not be transposed if the corresponding bit planes are not output.

If many (at least 8) bit planes are decoded, the recursive transposer can be 10 times faster than the default ZFP transposer and 4x faster than the AVX2 transposer. However, the AVX2 version is faster if only few bit planes are being decoded; the reason is that the recursive transposer makes many data movements that need to be amortized over a relatively large number of bit planes to transpose. Therefore, in my implementation, both the AVX2 and the recursive transposers are used. The former is used when fewer than 8 bit planes are decoded, the latter is used otherwise.

5.2.3 Compression of floating-point exponents

Because a brick contains many blocks, I also compress the exponent data across blocks (the official ZFP stores this information independently for every block). For each brick, the maximum exponent of all blocks is stored, followed by the dictionary compressed base-128-encoded block exponents expressed as differences from the max exponent. This so-called `VarInt` representation allocates fewer bits for smaller differences, as is the case here due to coherency between neighboring blocks of the same brick. Storing the maximum exponent per brick allows the decoder to skip decoding an entire brick if the maximum exponent is so small that the error tolerance is achieved even with no bit planes decoded.

5.3 Data organization and indexing

To facilitate the upcoming discussion, I define four common types of queries: (1) Q_{prec} queries that request data at coarse resolution but high precision, (2) Q_{res} queries that request data at low precision but high resolution, (3) Q_{mixed} queries that request at some balanced combination of precision and resolution, and (4) Q_{roi} queries that request data at very high resolution and precision, but only for a subset of the whole grid (the region of interest, or ROI). Figure 5.5 visually compares several approximations produced by such queries.

5.3.1 Tiles and chunks

As mentioned earlier, the whole encoded dataset is partitioned into smaller chunks that form the units of disk or network I/O in the proposed framework. For chunking, I do not fix the chunk size in bytes, but let each chunk span the same number of samples in space (I settle on the value of 512^3 through the experiments discussed in Section 5.5). To form chunks, the 3D space of precision, resolution, and spatial domain is partitioned into tiles of size T_B (bit planes) $\times T_S$ (subbands) $\times T_G$ (grid points), and assign compressed bits from the same tile to one chunk (note that this is possible in my particular compression scheme, provided that T_G is a multiple of the block size). Thus, a *tile* can be considered the "extent" of a chunk in the aforementioned 3D space. A small T_G can result in chunks too small for optimal I/O. On the other hand, a large T_G creates spatial couplings that can penalize small Q_{roi} queries; likewise, large T_B and T_S can negatively affect Q_{res} and/or Q_{prec} queries. I let $T_B = T_S = 1$, and choose T_G such that $T_G = 2^k$ and $T_G \geq B_x, B_y, B_z$. For efficient I/O, I do not write data to disk as soon as a brick is compressed, but buffer the compressed bits using one *bit stream* for each combination of subband and bit plane, and flush the bits only for each chunk. Equivalent bit planes across blocks (taking into account the block exponents) are written to the same stream. Figure 5.6 translates the understanding of *cuts* onto the 2D precision-resolution-sample space in terms of chunks of data.

5.3.2 Indexing and metadata

Note that my compression scheme is variable-rate, so true random access is not supported, for a significant gain in compression ratio. Nevertheless, fast access to compressed brick data is supported by storing brick IDs (interleaved brick indices) and the size of every brick in number of bytes, encoded using base-128 compression [50]. Brick IDs are compressed by unary coding the differences between consecutive IDs, which tend to be 1, but can also be different than 1 since a brick can be significant on a bit plane whereas a neighboring brick is not. Likewise, to support fast access of chunks, the (dictionary compressed) chunk IDs, chunk sizes, and the total number of chunks (Figure 5.7, left) are stored in each file. Given a chunk's address, the chunk can be located by a binary search in its corresponding file. In practice, parameters are chosen such that a file can contain at most tens of thousand of chunks, so that such a flat indexing scheme still works well. The exponent data is also stored into chunks in a way that mirrors the bit plane chunks, but in a separate set of files (Figure 5.7, right).

I use a single indexing scheme that works across brick, chunk, file, and directory levels. Starting from the least significant bit of a brick's interleaved index, portions of this index are assigned one-by-one to chunks, files, and directories (Figure 5.8). This indexing scheme defines an implicit tree over the space of directories, files, and chunks, which enables a depth-first lookup algorithm that computes a list of relevant directories, files, and chunks in logarithmic time, skipping irrelevant files entirely. Similar schemes have been used previously, *e.g.,* in the context of sorting objects in a bounding volume hierarchy [155, 217].

5.4 Data lookup and decoding

5.4.1 Chunk and brick lookup

Given a data query that contains an ROI, a resolution level, and an error tolerance, we need to quickly locate the relevant files, chunks, and bricks for decoding. At the API level, I do not make use of *cuts*, due to the complexity of providing such an API and because a cut can always be specified using a number of such queries. Since the topology of T_r^p is implicit (index-based), it is easy to ensure all such queries produce valid cuts, as the only constraint to be satisfied is that whenever a node in T_r^p is retrieved, all of its ancestors (i.e., coarser level nodes and more significant bit planes) — identified by indexing — are also retrieved. In practice, given such a query, the files containing the corresponding bits are located using a two-stage lookup. First, file lookup in the precision-resolution space is handled by defining a function that maps a point in that space to a file ID and a file path, which I define using bit packing (for file ID) and string concatenation (for file path). Given the requested nodes, the algorithm iterates over the relevant levels and bit planes, and applies this function each time to obtain the directories containing the relevant files.

Once inside such a directory, spatial lookup is performed to locate the exact files, chunks or bricks that intersect the requested ROI, relying on the indexing scheme described in Section 5.3. Whenever a relevant file is found, the traversal is continued in the same way to find relevant chunks in the sub-tree under the file. These chunks are then fetched using the offsets stored in the file. Once a chunk is fetched, relevant bricks are located using the metadata stored in the chunk. Although the whole chunk is fetched, only the relevant bricks in the chunk are decompressed to retrieve the requested bits. Note that depending on the query's requested downsampling factors, certain subbands do not need to be decoded and can be skipped completely. If the ROI is small enough, even blocks within a brick can be skipped; however, whereas both the fetching and decoding of a brick can be skipped, for a block, only the decoding can be skipped (*i.e.,* a block is read from disk but not decoded), since the metadata does not include the size of each encoded block.

5.4.2 Brick decoding

At decoding time, the steps performed for encoding are now done in reverse. In particular, bits are fetched from the streams, decoded, and "deposited" into a set of bricks, which start empty. Since not all bricks are present (depending on the query's ROI), I loosely maintain and update the current approximation in memory using a hashtable of bricks of wavelet coefficients, where the key consists of a brick's interleaved index and the brick's resolution level (including the subband). Each brick stores a grid of wavelet coefficients at a certain resolution level and subband. During decoding, bricks are updated with newly decoded coefficient bits, and the field is reconstructed through per-brick inverse wavelet transform, with missing coefficients assuming the value of 0. To keep a minimal state, the hashtable can also be avoided, and the inverse wavelet transform can be perfomred while having no access to previously decoded wavelet coefficients as they are presumably deallocated. Since the transform is linear, however, I can simply dequantize the (integral) increments obtained by decoding the current bit plane, inverse transform the floating-point increments, and update the field with the results.

Given an (absolute) error tolerance, only enough bit planes are encoded to conservatively ensure the reconstruction error is within the tolerance. Beside ZFP's transform and quantization, the wavelet transform introduces additional range expansion that influences error and needs to be compensated for. Range expansion at different levels are estimated using the infinity norms of the Kronecker products of 1D multiresolution wavelet synthesis matrices. As the first 10 norms (for the finest ten levels of resolution) are all less than 64, I conservatively encode/decode 6 additional bit planes on top of the number dictated by the input tolerance.

During block decoding, if the tolerance is τ and the block exponent is e, the smallest bit plane number to decode is $b - 6$, with b being the largest integer such that $2^{b+e} \leq \tau$. In practice, the actual absolute error tends to be smaller than the input tolerance due to data smoothness, so more bit planes than needed are often decoded. Nevertheless, the required error tolerance is always respected, up to machine epsilon with respect to the range. Finally, wavelet coefficients are multipled with the norms of corresponding wavelet basis functions, ensuring equal energy contributions from the same bit plane on all levels.

5.4.3 Progressive decoding

Progressive decoding is the ability to resume decoding as new data arrives without redoing previous work. This capability is useful when a bit stream must be decoded as it is being transferred to show gradually improving results to the user. Supporting this feature often means keeping the state of the decoder in memory, which may not be practical depending on the size of this state. In this case, coefficients in a block always follow the same order and have the same precision, so the state for each block consists of the number of significant coefficients (an integer between 0 and 64), the current bit plane (an integer between 0 and 63), and the block exponent (11 bits for `float64`). Thus, the complete state needed for ZFP to resume decoding can be stored in 24 bits per block of 64 samples in 3D.

5.5 Evaluation

I evaluate the efficacy of the proposed system using the data sets in Table 5.2, which features various types of scientific simulations. The test computer is a laptop with a 4-core CPU (2.8 GHz Intel Core i7-7700HQ), 32 GB of RAM and (unless otherwise specified) a 122 MB/s spinning hard drive. Note that only one CPU core is used in all tests. Throughout this section, the term “tolerance” implies precision levels (which corresponds to the RMSE in the ideal case); a high tolerance corresponds to a low-precision level and vice versa.

5.5.1 Lossless compression and metadata overhead

In Table 5.2, I show the encoding times and near-lossless compression ratios for all the data sets used in this work. The encoding speed reduces linearly from 35 MB/s (for Density at 4 GB) to 6.5 MB/s (for Pressure at 900 GB), likely due to effects of the disk cache and overheads associated with updating bookkeeping data structures that grow linearly in the size of the data. For memory, I note that the largest data set, Pressure (900 GB), is encoded using only 2.5 GB of RAM. For near-lossless encoding (where the tolerance is set to about the machine epsilon relative to data range), varying degrees of reductions are achieved, but all are reduced to less than the original size. For all tested data sets, the overhead of metadata is on the order of 1/1000 relative to the compressed data. The proposed system also supports also lossless compression of `float32` fields, but we note that in practice the last few bit planes are effectively noise and hence are expensive to compress while adding little to no value.

5.5.2 System parameters

To find optimal values for the free parameters we execute concrete instances of the various query types on the different data sets. I have found that grouping chunks of different bit planes and intra-brick subbands in the same file reduces query time across the board (compared to separating them), likely due to disk prefetching and reduced seek time. Chunks belonging to different spatial file regions (Figure 5.8) are still appropriately stored in separate files.

5.5.2.1 Tile size

To determine a tile size (i.e., T_G, the spatial extent of a chunk) that supports fast I/O across queries, I fix the brick size to 64^3 samples, vary the tile size from 2^3 to 16^3 bricks, execute different types of queries, and record the I/O as well as decompression time; the latter involves chunk/brick lookup, low-level decompression, inverse transforms and any in-memory data movement/transformation. The results can be seen in Figure 5.9 (left). Larger tiles (hence larger chunks) tend to result in significantly reduced I/O time, especially at high-resolution levels; in my experiments the tile size of 16^3 bricks reports the lowest I/O time, except for the smaller ROIs of sizes 250^3 and 50^3. I choose the tile size of 8^3 bricks for the system as this size works almost as well and is better for small-ROI queries. With this choice, $T_G = 512^3$ since the brick size is 64^3. Note that since a chunk is only an I/O unit, the chunk size (controlled by T_G) as expected has almost no effects on the decompression time.

5.5.2.2 Brick size

I next vary the brick sizes from 16^3 to 128^3 samples, while keeping the tile size constant in number of samples (computed using the choices of 64^3-samples per brick and 8^3-bricks per tile). The results are shown in Figure 5.9 (right), in which a few trends can be observed. First, I/O time is lower with larger brick sizes, likely due to less metadata for bookkeeping of bricks in each chunk, resulting in smaller chunks on average. Decompression time increases for bricks that are either too small or too large. Small bricks are faster to decompress themselves but also require reading and decoding a large amount of bookkeeping metadata. Using very small bricks helps only the tiny 50^3-ROI query. Based on these observations I choose the brick size to be 64^3 (i.e., $B_x = B_y = B_z = 64$), which works well across the queries for both I/O and decompression. Subsequent experiments assume the brick size of 64^3 samples and the tile size of 8^3 bricks.

5.5.3 Decoding time

Next I measure the decoding time (I/O and decompression) in three different environments: with a 100 MB/s spinning hard drive (HDD), a 1GB/s SSD, and a 1 MB/s network (Net) (Figure 5.10). I use four queries that read increasingly more data to evaluate I/O and decompression speed at different size scales. For HDD and SSD, at all scales the decompression time dominates the I/O time by an order of magnitude. The SSD only magnifies this difference. Decompression only becomes faster than I/O when using a network. Across data sizes, the I/O time also increases at a lower rate compared to decompression; however; even the I/O rates still do not saturate the hardware's capacity. Note that while these numbers are useful to compare the current state of I/O and decompression in my system; there are many opportunities for optimization left unexplored such as multi-core decompression, overlapping I/O and decompression as well as better I/O request batching and simply faster raw decoding.

5.5.4 Compression comparisons

I compare the proposed method against the state-of-the-art techniques, namely SZ [278], TTHRESH [12], JPEG2000 [264] (using OpenJPEG [3]), ZFP, and VAPOR [162]. Compared to SZ and TTHRESH, my method's decompression time and memory usage are orders of magnitude lower (Figure 5.11). My data quality is competitive against both at $\approx 300\times$ compression ratio and is only slighty worse than TTHRESH at very high ratios (Figure 5.12). Note that my results are decoded from a single data layout, whereas TTHRESH and SZ have to re-compress each time. Similarly, at comparable data sizes, JPEG2000 has slightly better data quality (39.0 dB versus 43.9 dB) at the expense of 2000× higher memory usage and 15× slower decompression time. Using a series of Q_{mixed} queries with increasingly lower tolerances, in Figure 5.13, I show that my method achieves ZFP's decoding speed while enabling very high compression ratios. Furthermore, for mid- and low-quality levels (PSNR < 50 dB), my system can decode at lower resolutions, thus achieving significantly lower decoding time. Compared to VAPOR, my method achieves substantially better quality at 300× compression ratio, while retrieving the data using one-fourth the memory and one-third the time, as well as avoiding blocking artifacts at very low bit rates (Figure 5.14).

5.5.5 Reconstruction quality

Next, I study data quality for 16 approximations at the vertices of a 24-point grid in the precision-resolution space. The PSNR at each point is plotted, and points that lie on the same resolution are connected (Figure 5.15). The rate-distortion curves suggest that the data quality at low-resolution levels quickly reaches a plateau (which is expected since the low number of data points, no matter how accurate, puts a hard limit on data quality measured in PSNR). The best rate-distortion curve can be thought of as an imaginary "envelope" that is the upper bound of all four individual curves.

Finally, to demonstrate the scalability of the proposed system, I compress a 900 GB turbulent channel flow field [159] ($10240 \times 7680 \times 1536$, float64) and decode three approximations (Figure 1.4) of progressively increasing quality decoded along a curve in the precision-resolution space. The figure reports the decode time and memory usage. The results show that my system can achieve a 120,000× compression (900 GB to 7 MB) with minimal quality loss in volume rendering. The 7 MB approximation is also decoded in 1.1 seconds using only 13 MB of RAM.

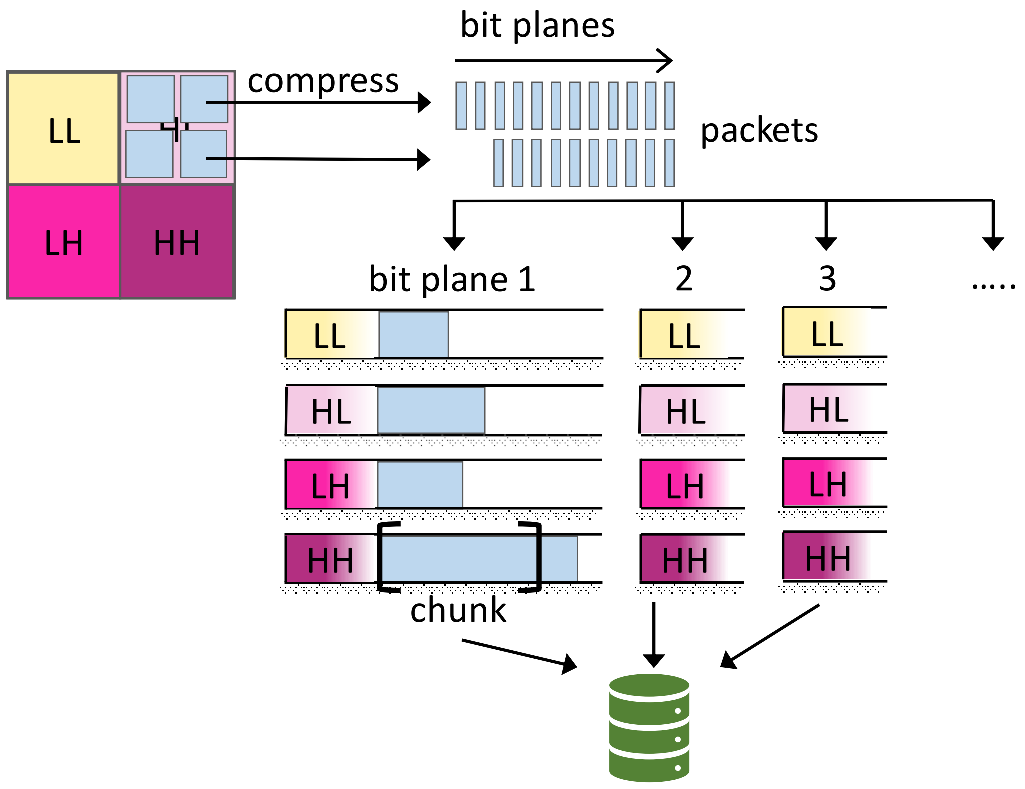


Figure 5.1: Overview of the proposed system. Bricks are transformed independently, creating multiple subbands. Subbands are compressed independently in blocks, creating multiple packets, one for each bit plane of the block. Packets are written into channels, each corresponds to a (bit plane, subband) combination. Each channel is split into contiguous chunks, and the chunks are written to disk.

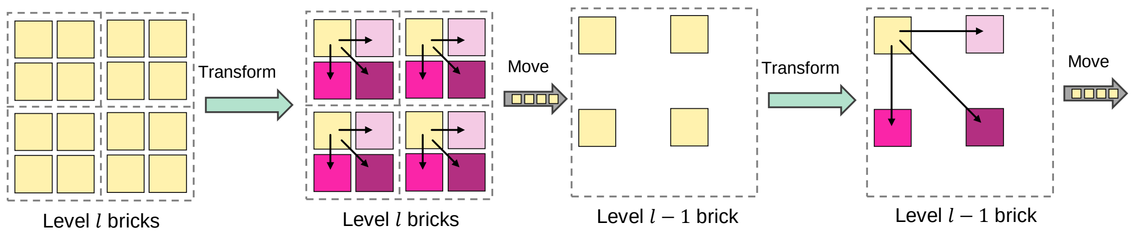
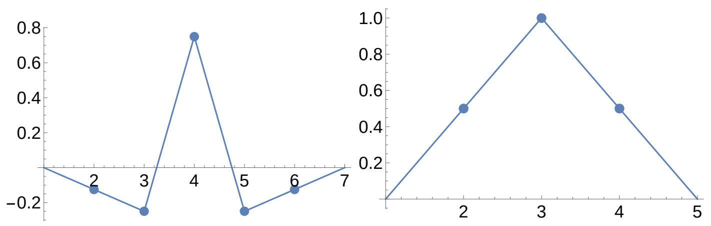
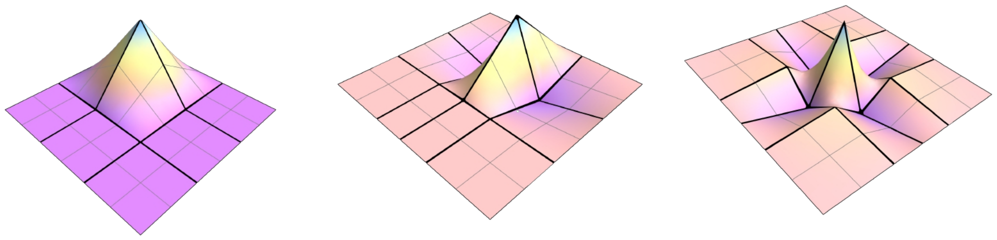


Figure 5.2: At each level, the samples in each brick are transformed to form local trees, and then every 2^d root nodes (in yellow) of such trees are copied into a parent brick, before the process is repeated for the next coarser resolution level.

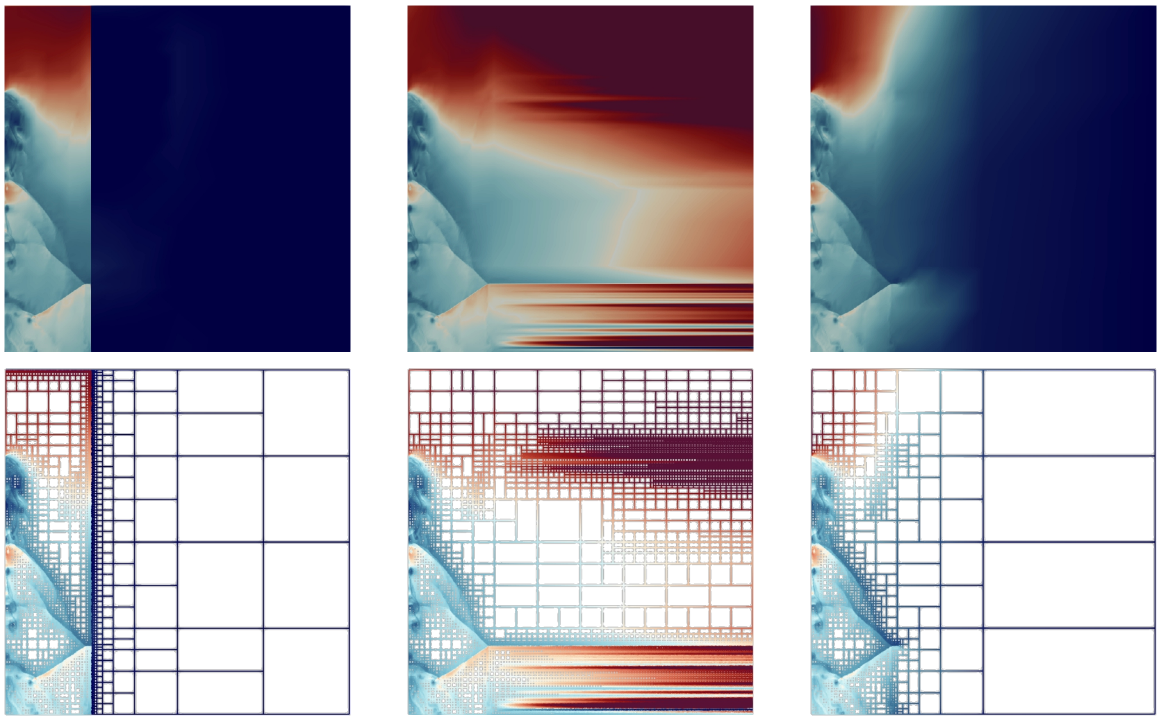


(a) 1D scaling (S) and wavelet (W) basis functions



(b) 2D tensor-product wavelets (SS, SW and WW)

Figure 5.3: The CDF 5/3 linear B-spline wavelet basis functions in 1D and 2D.

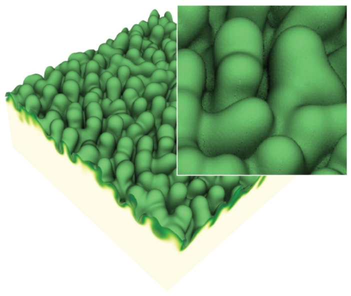


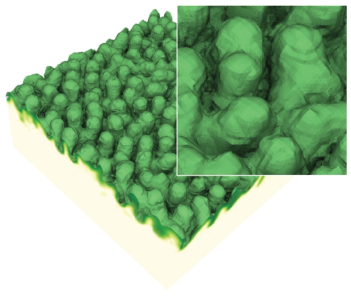
(a) Zero padding
(7997, 9918, 4664)

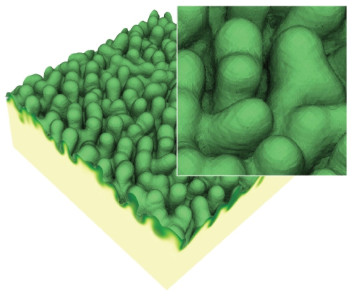
(b) Linear extrap.
(6263, 34742, 19007)

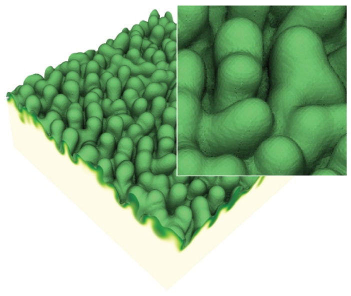
(c) Linear-lifting
(6263, 6342, 2965)

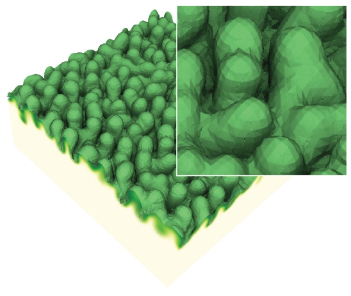
Figure 5.4: Function extensions (top) and the corresponding meshes (bottom) for a shockwave defined on $[256 \times 1024]$ domain and extrapolated to $[1025 \times 1025]$ using different methods. The associated metrics are number of (cells, leaf nodes, internal nodes). Zero padding introduces artificial discontinuities at the boundary of the input domain (notice the vertical blue streak of finest-level cells). Linear extrapolation maintains smoothness near the boundary, but can create discontinuities farther out from the original domain. My linear-lifting approach avoids artificially large wavelet coefficients at the boundary and in the extrapolated region.


(a) $384^2 \times 256, \epsilon = 0$, 288 MB


(b) $192^2 \times 128, \epsilon = 1/4$, 195 KB


(c) $192^2 \times 128, \epsilon = 1/32$, 357 KB


(d) $192^2 \times 128, \epsilon = 1/128$, 563 KB


(e) $96^2 \times 64, \epsilon = 1/128$, 199 KB

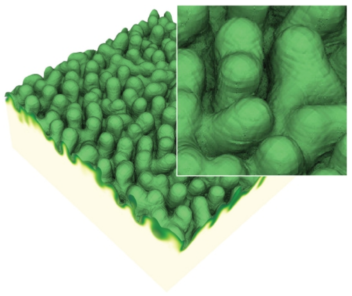

(f) $384^2 \times 256, \epsilon = 1/16$, 1024 KB

Figure 5.5: Isocontours extracted at different resolution (R) and precision levels (expressed as ϵ for absolute error) using the proposed system. It is interesting to see that better surface quality can be obtained with less data at a lower resolution but higher precision level ((d) vs. (f)). The opposite can also be seen, where higher resolution but lower precision (c) results in better surface quality (than (e)). This example demonstrates both the versatility of my system (all approximations are obtained from the same data layout on disk), and the need for fine control in precision and resolution.

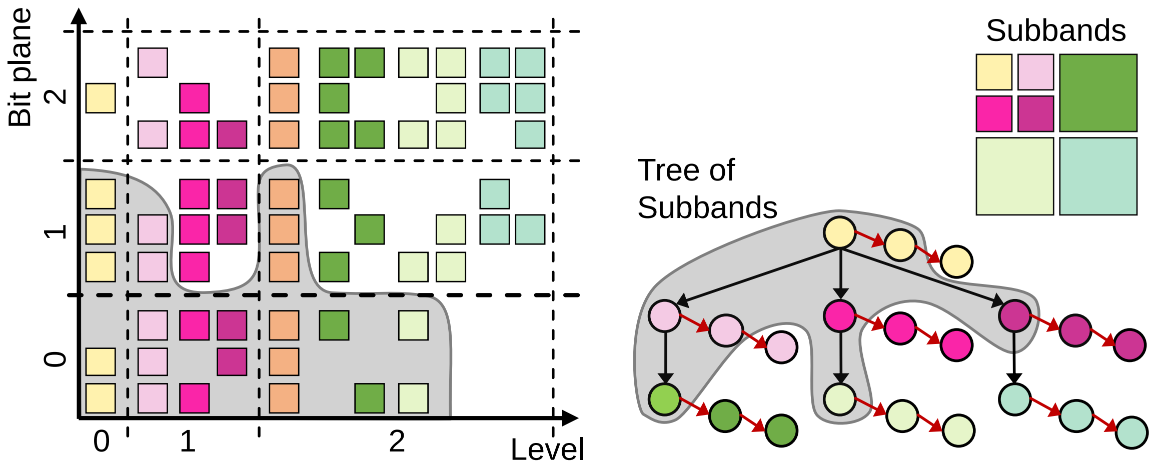


Figure 5.6: Visualization of chunks in the precision-resolution space. The hierarchy shown here has three levels, seven subbands, and three bit planes. Each combination of subband and bit plane contains a number of chunks, shown as squares and colored by subband (light brown squares are extrapolated chunks). Each node of the tree represents an entire subband instead of individual coefficients/blocks. The shaded regions represent a cut in the tree and the corresponding subset of chunks.

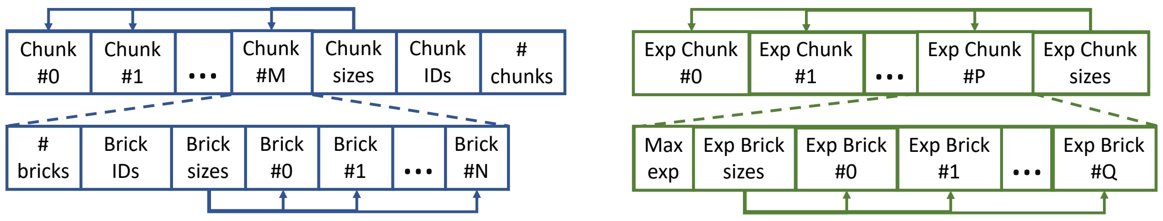


Figure 5.7: File and chunk organization for compressed bit plane data (left), and for exponent data (right). Top is how chunks are stored in a file, and bottom is how a chunk is organized internally.

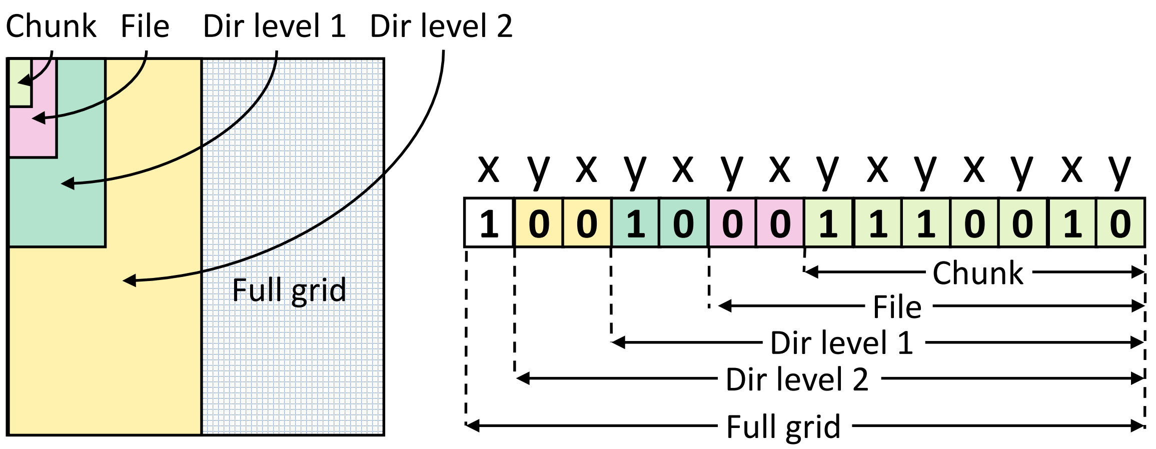


Figure 5.8: Portions of the interleaved brick index are assigned to chunks, files, and directories. In this toy example, *spatially*, every 2^7 bricks form a chunk, every 2^2 chunks form a file, every 2^2 files form one directory, etc.

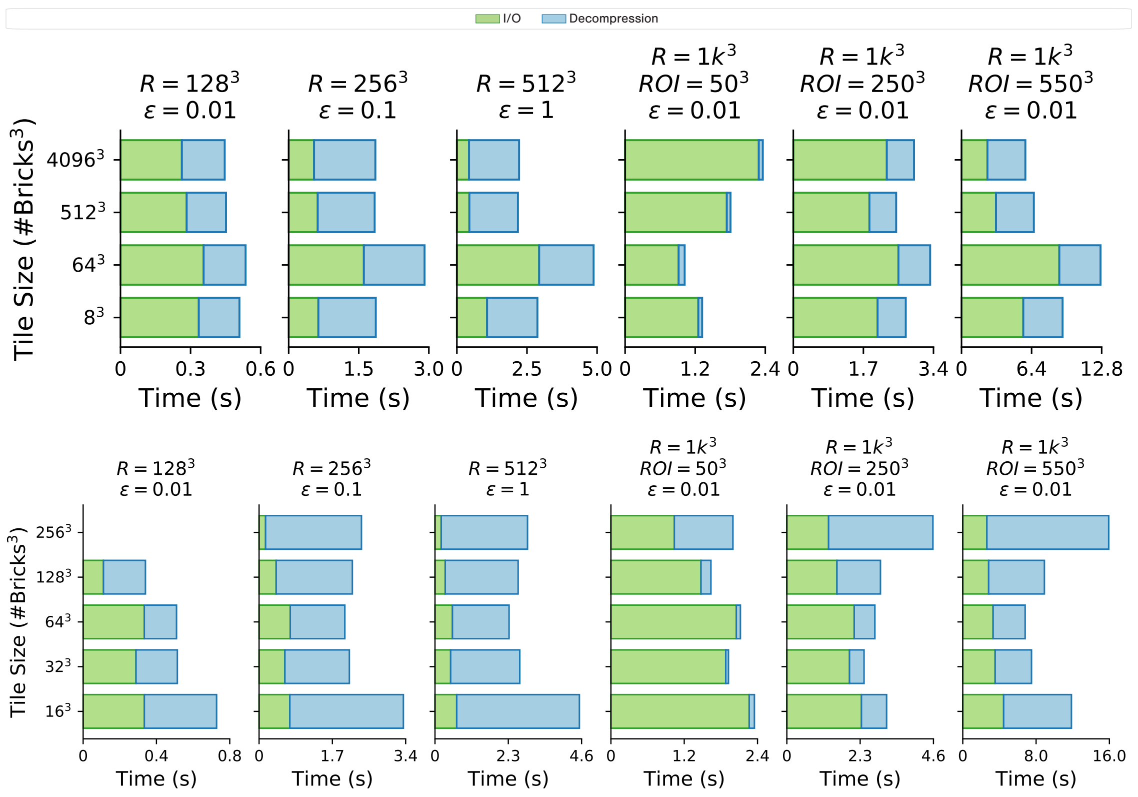


Figure 5.9: I/O and decompression times for different tile sizes (left) and brick sizes (right) across a wide range of query types. The resolution and precision are controlled by the R and ϵ (absolute error) parameters, whereas *ROI* controls the size of the region of interest (which, when omitted, means the same as R). Tile size of 8^3 bricks and brick size of 64^3 samples appear to be sweet spots that work well across queries.

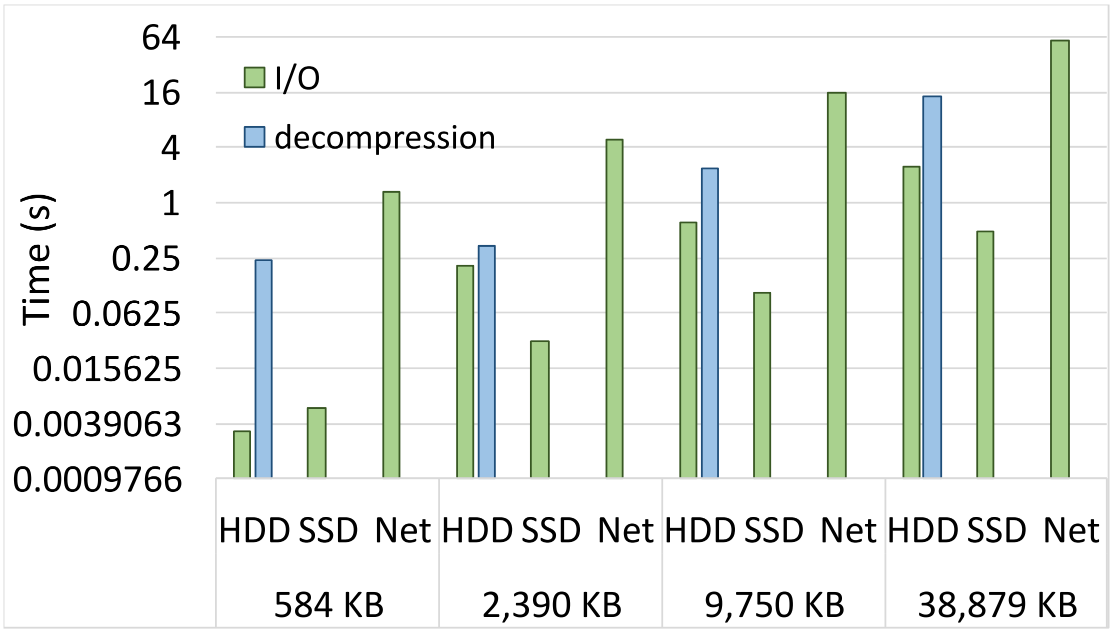


Figure 5.10: I/O and decompression time on a 100 MB/s spinning hard drive (HDD), a 1 GB/s SSD, and a 1 MB/s network (Net), across four scales of decompression rates (the full data is 9 GB). The vertical axis is in logarithmic scale. The decompression times for SSD and Net are not repeated to avoid cluttering.

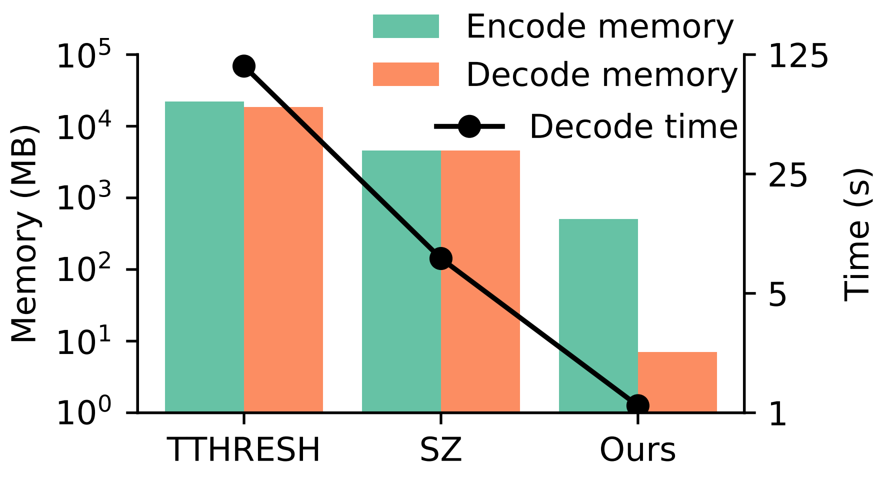
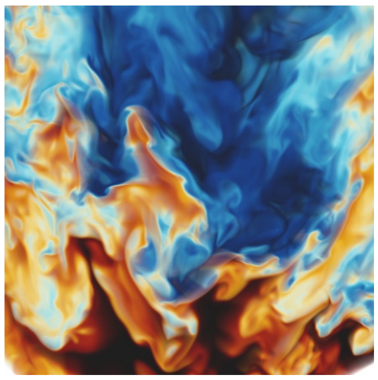
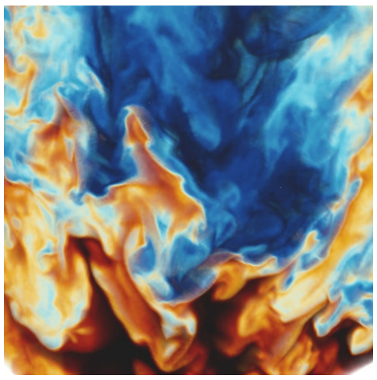
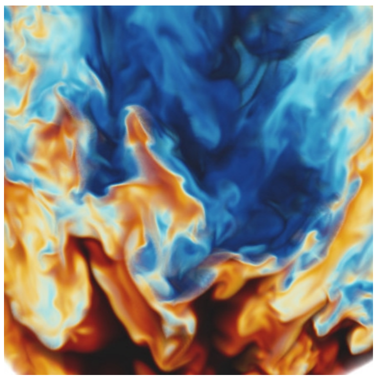
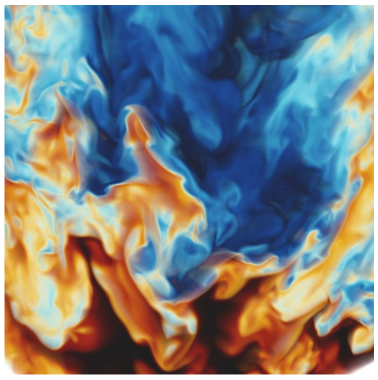


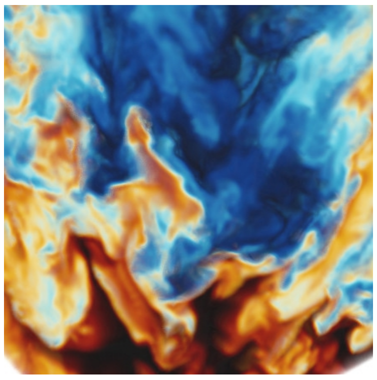
Figure 5.11: Memory usage and decoding time for the three decompressors (SZ, TTHRESH and ours), both in logarithmic scale.


(a) Reference – 2 GB


(b) SZ 6.5 MB, 0.976 SSIM


(c) Ours 6.4 MB, 0.965 SSIM


(d) TTHRESH 6.4 MB, 0.982 SSIM


(e) Ours 800KB, 0.910 SSIM

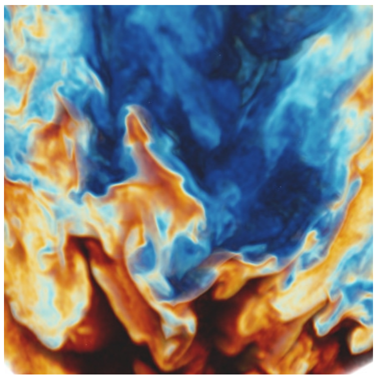

(f) TTHRESH 857 KB, 0.968 SSIM

Figure 5.12: Comparison of data quality between (b) SZ, (c) ours and (d) TTHRESH at 300× compression ratio. (e, f) Same comparison but at 2600× ratio, (SZ did not produce a result here). (g) Plots of decode time and memory usage for the three methods. My method uses orders of magnitudes less time and memory for decoding compared to SZ and TTHRESH.

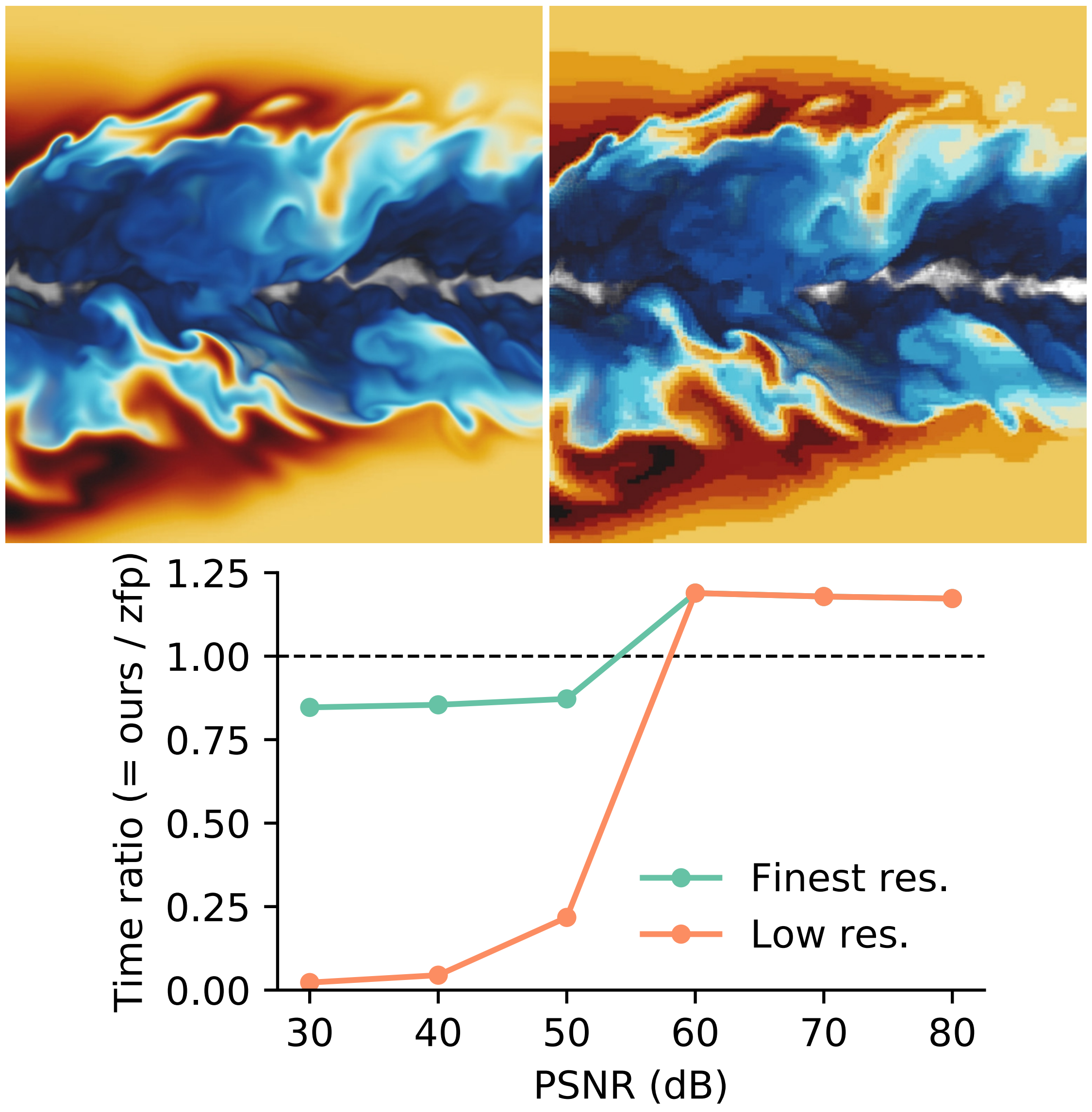


Figure 5.13: Comparison between my method and ZFP. My method (left, 2MB) supports very high compression ratios > 1000× where ZFP (middle, 12 MB) struggles, while maintaining the same decoding performance (bottom plot).

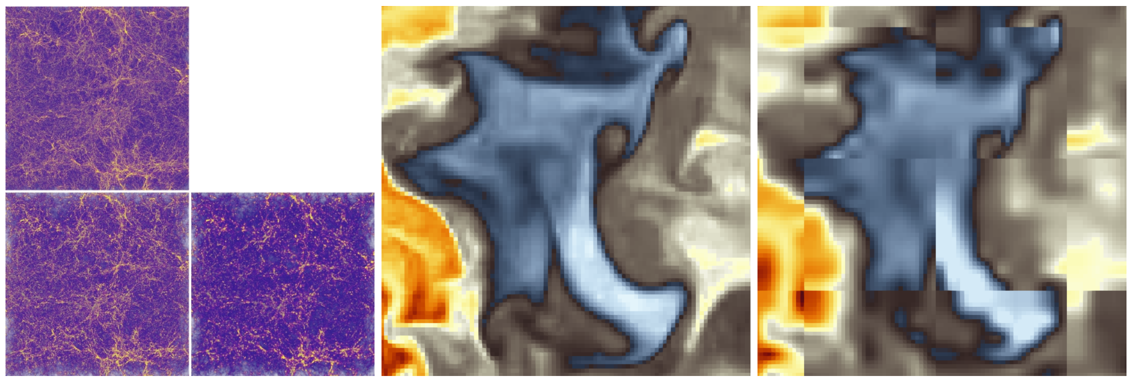


Figure 5.14: Comparison of data quality between the proposed system (bottom left, 109 MB, SSIM 0.78) and VAPOR (bottom right, 96 MB, SSIM 0.69). Ours (middle, 10 MB) is free from blocking artifacts visible with VAPOR (right, 13 MB).

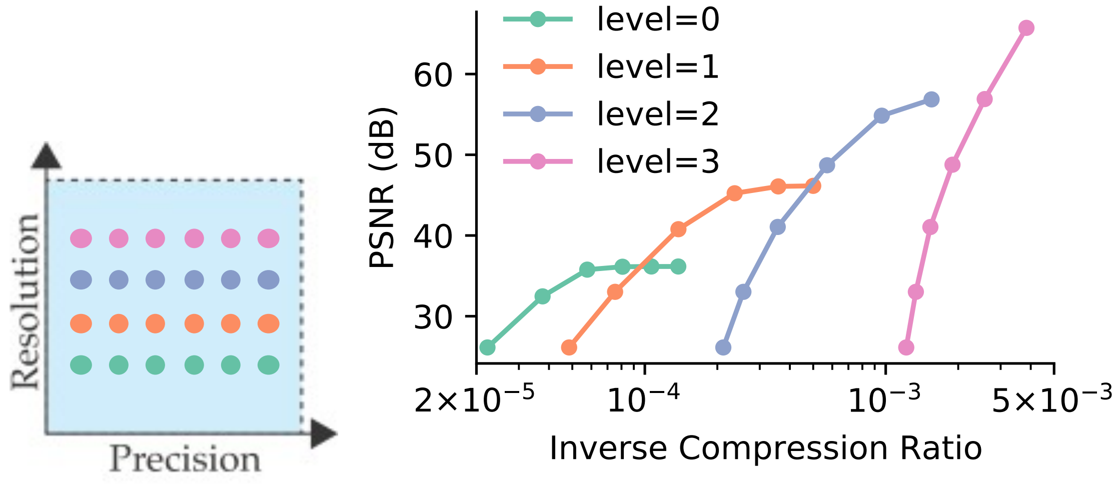


Figure 5.15: Rate-distortion curves going through fixed-resolution points in precision-resolution space.

Table 5.1: The proposed linear-lifting approach extrapolates a 6-point function using two levels of transform with integer arithmetic. A (forward) lifting phase begins with linear extrapolation (pink), followed by w-lift (brown) and s-lift (blue). Inverse lifting extends the input function to 9 points, but only 7 coefficients are stored for full reconstruction. Note that the extrapolated function is different from one obtained via simple linear extrapolation in the last two elements.

Input function	56	8	48	44	32	8			
Level 1: Extrapolate	56	8	48	44	32	8	-16		
Level 1: Forward w-lift	56	-44	48	4	32	0	-16		
Level 1: Forward s-lift	45	-44	38	4	33	0	-16		
Level 2: Extrapolate	45		38		33		-16		-65
Level 2: Forward w-lift	45		-1		33		0		-65
Level 2: Forward s-lift	45		-1		33		0		-65
Coefficients stored in memory	45	-44	-1	4	33		-16		-65
Level 2: Insert w-coefficients	45		-1		33		0		-65
Level 2: Inverse s-lift	45		-1		33		0		-65
Level 2: Inverse w-lift	45		38		33		-16		-65
Level 1: Insert w-coefficients	45	-44	38	4	33	0	-16	0	-65
Level 1: Inverse s-lift	56	-44	48	4	32	0	-16	0	-65
Level 1: Inverse w-lift	56	8	48	44	32	8	-16	-41	-65
Extrapolated function	56	8	48	44	32	8	-16	-41	-65

Table 5.2: Tabulation of the data sets used for evaluation, with compression ratios and compression speeds when absolute tolerances are 5×10^{-8} for `float32` and 10^{-16} for `float64`.

Data set	Resolution × Precision (X × Y × Z) × Bits	Range [min, max]	Compression Ratio	Compression Speed (MB/s)	Metadata overhead
Pressure [159]	(10240 × 7680 × 1536) × 64	[−0.23, 1.26]	1.36×	6.5	0.06
Dissipation [110]	(4096 × 4096 × 4096) × 32	[0.00, 82.67]	1.41×	11.2	0.08
Dark matter [10]	(2048 × 2048 × 2048) × 32	[0.00, 486.31]	1.16×	14.2	0.07
Temperature [319]	(2025 × 1600 × 400) × 64	[4.4819.24]	1.95×	19.3	0.09
Mixed fraction [24]	(920 × 1400 × 720) × 32	[0.00, 1.00]	11.36×	26.0	0.16
Density [48]	(1024 × 1024 × 1024) × 64	[1.00, 3.00]	2.11×	35.9	0.09

CHAPTER 6

OPTIMAL PROGRESSIVE STREAMS FOR DATA ANALYSES

With the data encoding and organization system in place, I present a progressive data streaming framework that takes advantage of such a system to progressively improve data quality in either resolution or precision. Within this framework, I propose a greedy approach to compute "optimal streams" that give estimations for lower bounds of error for various analysis tasks. I also model traditional data reduction schemes as progressive streams in the proposed framework, therefore making it possible to fairly compare these schemes with the proposed data streams to study the gains and trade-offs that result from combining reducing data precision and reducing data resolution.

6.1 Packet streaming framework

In order to systematically study the resolution-versus-precision trade-offs among different data reduction schemes, it is important to perform fair and consistent comparisons. In this section, I develop such a consistent methodology by proposing to model different data reduction schemes as *streams* of uniformly sized data *packets*, where the original data contains all the packets, and any reduction step removes a set of packets (comparable amounts of data). These data streams are transmitted using a client-server model. At any point, the client is assumed to have received a subset of packets (in some predetermined sequential order or in some order requested by the client), which can be used to reconstruct an approximation to the original data. Therefore, to compare different streams, the original data is reconstructed using the *same number* of packets from each stream and perform desired tasks on each of the (approximate) reconstructions. A stream is considered better suited for a given task if it produces results that are closer to the reference results computed from the original data. Figure 6.1 gives a schematic view of the proposed data streaming model.

Although both the server and the client in my model can be on the same physical machine, only the server has full knowledge of the data. Thus, when the client receives a packet, it might not know where that packet should be deposited. A common solution is to have both the client and the server agree beforehand on a static ordering of packets, independent of the data. I use the term *data-independent streams* to refer to streams using such solutions. In contrast, for *data-dependent streams*, an additional mechanism is needed to inform the client about the subbands and bit planes of incoming packets. In this work, I consider both types of streams, as well as specialized *task-dependent* streams optimized for given tasks (see Table 6.1). The list of datasets studied here can be found in Table 6.2.

6.1.1 Decomposition of data into packets

Although one way to compare different data reduction strategies is to restrict the techniques to the same data size and compare data quality, it is difficult to enforce consistency. For example, the amount of change (in data) in one step of multiresolution simplification may be different from removing one bit in the quantization of every sample. To make all data reduction schemes comparable, in each scheme, a data set is redefined as a stream of equally sized *packets*. These packets are the smallest units of data transfer in this framework. A packet consists of a relatively small number $\left(\approx 2^3\right)$ of bits and is associated with a resolution level and a precision level (i.e., bit plane). In this framework, different data-reduction schemes become different orderings of packets, called *streams*. Restricting two (or more) data streams to the same number of packets allows us to perform fair and consistent comparisons.

- **Resolution levels.** Although there exist several ways to define the notion of resolution/scale/frequency, I choose the multilevel basis functions of the wavelet transform because they have compact support, and they avoid interpolation problems associated with other representations. Wavelet transform enables spatial adaptivity (i.e., finer resolution in regions that contain sharp features, at the expense of coarser resolution elsewhere). In particular, I choose the CDF5/3 multilinear wavelets [47] for their balance between simplicity and effectiveness at decorrelating the input signal in practice [264].

  A multidimensional wavelet transform can be performed in multiple passes, which partitions the original domain into *subbands*, each of which can be thought of as a resolution level associated with one or more spatial direction. One transform pass (in 3D) creates eight subbands, of which the first is a low-pass, downsampled version of the original data, and the remaining add fine details in each subset of the dimensions (see Figure 6.1 for a visualization of subbands in 2D). A subsequent transform pass recurses only on the first subband (of the previous level), creating the next (resolution) level of subbands. Let us use l $(0 \le l < L)$ to index the subbands, with $l = 0$ referring to the coarsest subband and L denoting the number of subbands. In 3D, the eight subbands created after one transform pass are indexed in the following order, from coarse to fine: LLL, LLH, LHL, LHH, HLL, HLH, HHL, HHH (L stands for "low" and H stands for "high", referring to the low- and high-pass filter pair that perform the wavelet transform). The LHL subband, for example, contains coefficients that are low-pass transformed along X and Z, and high-pass transformed along Y. In my experiments, the number of subbands, L, is fixed at $1 + 3 * 7 = 22$, corresponding to three transform passes in 3D.

- **Precision levels.** For creating packets corresponding to different precision levels, floating-point wavelet coefficients are quantized to B-bit signed integers. For most of the experiments in this work, $B = 16$. This quantization eliminates the floating-point exponent bits, such that every bit (except the sign bit) can be associated with a bit plane b $(0 \le b < B)$. I use the convention that the higher indexed bit planes are less significant. I convert quantized coefficients to the negabinary representation, where integers are represented in base -2, i.e., $\sum_{b=0}^{B} c_b(-2)^b$ with $c_b \in \{0, 1\}$. Negabinary encoding is preferred over two's complement encoding, because data reconstruction starts by zero-initializing all bits, and negabinary encoding has no single dedicated sign bit and ensures that small coefficients have many leading zero-bits. This transformation increases the number of bit planes by one, i.e., $0 \le b \le B$.

- **Blocks and packets.** Precisely, a *packet* consists of bits from the same bit plane, from a *block* of negabinary wavelet coefficients. A block is a $[g \times g \times g]$ grid of adjacent coefficients from the same subband. I let g be a constant ($g = 2$ here), so that finer resolution subbands contain more packets, which presents a trade-off between packets that provide wider (but coarser) coverage and packets that provide finer (but more local) details. Every packet (of size one byte in this work) comes from a bit plane b and a subband l. g is chosen to be larger than one for performance reasons, as in practice, most systems read bits in batches.

6.2 Data-dependent and data-independent streams

I define two streams: *by level* and *by bit plane*, which model two common reduction schemes in the literature. The *by level* stream, $\mathcal{S}_{\text{lvl}}$, orders the packets strictly from coarser to finer subbands. Within the same subband, packets follow the row-major order of blocks and then bit plane order (from 0 to B) within each coefficient. All bits for each coefficient are streamed together. The other common ordering, *by bit plane*, or $\mathcal{S}_{\text{bit}}$, proceeds strictly from higher ordered to lower ordered bit planes. Within the same bit plane, packets follow the subband order (from 0 to $L-1$) and then row-major order in each subband. $\mathcal{S}_{\text{lvl}}$ and $\mathcal{S}_{\text{bit}}$ are designed to mimic the way data is accessed in traditional methods that work either in resolution ($\mathcal{S}_{\text{lvl}}$) or in precision ($\mathcal{S}_{\text{bit}}$).

Additionally, I define a third stream that combines these two dimensions and refer to it as *by wavelet norm*, or $\mathcal{S}_{\text{wav}}$. This stream orders packets in descending order of weights $w_{\text{wav}}(p) = 2^{B-b(p)} \times ||\psi_{l(p)}||$, where p denotes a packet and $\psi_{l(p)}$ represents wavelet basis on subband $l(p)$. The $||$ notation refers to the L_2 norm of the wavelet basis function. The first term captures the contribution of a bit on bit plane $b(p)$, and the second term captures the contribution of a wavelet coefficient on subband $l(p)$. In the wavelet representation, a function f is written as a linear sum of wavelet basis functions, i.e., $f = \sum c_i \psi_i$, where c_i are the coefficients. Since the wavelet transforms are based on lifting, this norm is usually not one, but it increases with level. Basis functions in the same subband share the same norm, hence $w_{\text{wav}}(p)$ is simply the contribution (in L_2 norm) of a bit on bit plane $b(p)$ and subband $l(p)$, to the whole function f. This ordering based on norms of wavelet basis functions was proposed previously by Weiss et al. [305]. For details of computing the norms of basis functions, see Appendix A.

Another common way to reduce data in the wavelet domain is to leave out the coefficients of the smallest magnitudes. Note that coefficient magnitudes are only weakly related to error, as the error also depends on the wavelet basis function norm [305]. This scheme is modeled with a stream called *by magnitude*, or $\mathcal{S}_{\text{mag}}$. Here, the weight function is $w_{\text{mag}}(p) = \sum_{c \in \text{block}(p)} ||c||$ (the sum is over all coefficients in the block that contains packet p). If two packets have the same weight, they are ordered by subband index and then by bit plane.

Unlike $\mathcal{S}_{\text{lvl}}$, $\mathcal{S}_{\text{bit}}$, and $\mathcal{S}_{\text{wav}}$, the $\mathcal{S}_{\text{mag}}$ stream is data dependent because the coefficient magnitudes are not known without the data. In principle, data-dependent streams are better than data-independent streams because they can prioritize important packets based on the actual data. However, data-dependent streams are ill-suited for practical purposes, because the cost of sending position information likely outweighs any potential benefit. Nevertheless, I study them for various reasons. First, the *by magnitude* scheme is well known in the literature [162]. Second, the "best" data-dependent streams (which do not include position information for packets) can serve as a baseline to evaluate the performance of their data-independent counterparts. Finally, in addition to being data dependent, streams can also be task dependent (Subsection 6.2.1), which may provide insights into how data should be queried to perform certain analysis tasks.

6.2.1 Task-optimized streams

Each analysis task may require a fundamentally different stream for optimal results. Studying such "optimal" streams is important because they not only serve as a baseline but also can provide insights into other, more practical streams. Given the original data set f and its reconstructed approximation f' using a subset of packets, let q represent some quantity of interest, e.g., histogram, isosurface, etc., computed on f or f'. For a given q, a well-defined error metric $e(q(f'), q(f))$, which returns a single scalar, is needed. Given f, q, and e, my goal is to generate an *optimal* (and data-dependent) stream, $\mathcal{S}_{\text{opt}}$, for q with respect to e. One possible definition for $\mathcal{S}_{\text{opt}}$ is a stream such that the area under the plotted curve of e, with respect to the number of bits, is minimized for all packets to be streamed. However, this definition is limited in practice because a stream should be able to terminate at any point and still produce as small an error as possible.

Instead, I employ a greedy approach to define the optimal stream. I notice through experiments, however, that a straightforward greedy algorithm can pick unimportant packets too early. For example, starting with an all-zero reconstruction $f' = 0$ and an empty stream, new packets can be repeatedly appended to the stream, which when included in the current f', would minimize e at every step. At some point, the algorithm might pick a packet that introduces the lowest error yet contributes very little to improve the quality of f' (because more important packets would increase the error), leading to a nonoptimal stream. To avoid this problem, I make a modification to this greedy algorithm and build the stream backwards. Starting with a "lossless" f' (i.e., $f' = f$), and at each step the packet that has the least impact on the error e is removed from f'. This modification largely avoids the previous problem where less important packets were added to $\mathcal{S}_{\text{opt}}$ too early, because by starting with the full (instead of empty) set of packets, the error measurement better captures the importance of packets.

Unfortunately, such a greedy algorithm is still expensive in practice, as its complexity is at least $O(n^2)$ (n is the number of packets), due to the 2-level nested loop. For a $nx \times ny \times nz$ volume, a block size of $bx \times by \times bz$, and $B + 1$ bit planes, n is $\frac{nx}{bx} \times \frac{ny}{by} \times \frac{nz}{bz} \times (B + 1)$. Thus, even a small volume, e.g., $nx, ny, nz = 64$ and $bx, by, bz = 2$, can result in a prohibitively high run time, as $n^2 = (32768 \times 17)^2$. Therefore, I adopt a simplified version of this algorithm, where only one pass through n packets is needed. In iteration i ($0 \leq i < n$), a new packet p_i is set to zero, then the incurred error w_i using the error metric e is computed and recorded, and then enabled p_i again at the end of iteration i. After n iterations, each packet has an associated weight w_i. The stream $\mathcal{S}_{\text{opt}}$ is simply the sorted list of packets in decreasing order of the weights. This simplified algorithm (6.1) has significantly lower running time, while (by observation) retaining the same quality for $\mathcal{S}_{\text{opt}}$.

Algorithm 6.1 Computing a task-optimized stream

```
1: Inputs:
       An original function $f$
       An unordered set of $n$ packets $P = \{p_i\}$, produced from $f$
       A quantity of interest $q$, and an error function $e$
2: Initialize:
       A set of $n$ weights $\{w_i\}$
3: for each packet $p_i$ do
4:     $p_i := 0$
5:     $P \rightarrow$ wavelet coefficients $C = \{c_j\}$
6:         (inverse quantization and inverse negabinary transform)
```

```
7:     $\{c_j\} \rightarrow f'$ (inverse wavelet transform)
8:     $w_i := e(q(f'), q(f))$
9:     Restore $p_i$
10: Sort the $p_i$'s in descending order of $w_i$.
11: **Output:**
       The $q$-optimized stream, which is the sorted $P$
```

For a more optimized implementation, the inverse wavelet transform on line 7 can be replaced by "splatting" coefficient p_i onto the domain, due to the transform being linear and the fact that p_i is the only coefficient changed in the current iteration. Since tt is almost never advantageous to stream bits belonging to one coefficient out-of-order, I also enforce that in the final stream, packets belonging to the same group follow the bit plane order (from 0 to B).

6.2.2 Stream signatures

Unlike data-independent streams, data-dependent streams do not impose a static ordering of packets. To concisely represent and characterize the dynamic ordering of data-dependent streams, I introduce the notion of a *stream signature*. Any stream can be represented with respect to the two-dimensional space of resolution (subbands) and precision (bit planes), i.e., $\mathbb{L}_{L,B} = \{(l, b) \mid 0 \leq l < L,\ 0 \leq b \leq B\}$. Given a stream, its signature A is defined as an $L \times (1 + B)$ matrix, where each (l, b) element is associated with $P_{l,b}$, the set of packets belonging to subband l and bit plane b. In particular, $A(l, b)$, i.e., the (l, b) element of A, is an integer in the range $[0, (1 + B) \times L)$, and indicates, on average, the position at which packets in $P_{l,b}$ appear in the given stream. For example, the signature $A = \begin{bmatrix} 0 & 1 & 4 \\ 2 & 3 & 5 \end{bmatrix}$ indicates that the stream begins with packets that lie on the first bit plane of the first subband, as $A(0, 0) = 0$. Those are followed by packets on the second bit plane of the first subband ($A(0, 1) = 1$), and then the first bit plane of the second subband ($A(1, 0) = 2$), and finally, the third bit plane of the second subband ($A(1, 2) = 5$). Thus, a stream's signature shows how the stream traverses the space $\mathbb{L}_{L,B}$ and highlights the different resolution-versus-precision trade-offs among streams, especially among $\mathcal{S}_{\text{opt}}$ streams optimized for different tasks. In Figure 6.2 I visualize the signatures of $\mathcal{S}_{\text{bit}}$, $\mathcal{S}_{\text{lvl}}$, and $\mathcal{S}_{\text{wav}}$, defined in Subsection 6.1.1.

To compute a stream signature, I partition the whole domain (not individual subbands) into several *regions*, compute one signature per region, and average these local signatures. Partitioning is used since it is only when packets are relatively well localized that their relative ordering in the $\mathbb{L}_{L,B}$ space becomes meaningful. For example, a packet at one corner of the domain may be streamed before one at an opposite corner, but this fact contains no useful information. I define a region to be the spatial volume that is covered by a packet in the coarsest subband. Algorithm 6.2 lists the steps of my approach.

Algorithm 6.2 Computing a stream signature

```
1: **Inputs:**
       A stream $P = \{p_i\}$
2: **Initialize:**
       Per-region signature matrix $A_r := 0$
       Global signature matrix $A := 0$
3: **for** each packet $p_i$ in $P$ **do**
4:     Let $r, b, l$ be the region, bit plane, and subband that $p_i$ belongs
5:     $A_r(l,b) := A_r(l,b) + i$
6: **for** each region $r$ **do**
7:     Sort the elements of $A_r$
8:     Assign each element of $A_r$ its index after sorting
9:     $A := A + A_r$
10: Sort the elements of $A$
11: Assign each element of $A$ its index after sorting
12: **Output:**
       The signature matrix $A$
```

Finally, a signature can be used to construct a stream denoted generically as $\mathcal{S}_{\text{sig}}$. This construction is done by iterating through each element $A(l,b)$ in ascending order and adding to the end of $\mathcal{S}_{\text{sig}}$ all the packets in $P_{l,b}$. An $\mathcal{S}_{\text{sig}}$ captures the behavior (in the $\mathbb{L}_{L,B}$ space) of the stream it derives from, but it is stripped from any spatial adaptivity. Hence, when $\mathcal{S}_{\text{sig}}$ is derived from an $\mathcal{S}_{\text{opt}}$, it can serve as a bridge when comparing the resolution-versus-precision trade-offs between data-independent and data-dependent streams.

6.3 Evaluation on different analysis tasks

Thus far, I have presented several types of streams: data-independent ($\mathcal{S}_{\text{lvl}}$, $\mathcal{S}_{\text{bit}}$, $\mathcal{S}_{\text{wav}}$), data-dependent and task-independent ($\mathcal{S}_{\text{mag}}$), and task-dependent ($\mathcal{S}_{\text{opt}}$, $\mathcal{S}_{\text{sig}}$). In this section, I consider a variety of common analysis and visualization tasks to evaluate the performance of these streams. For each task, I define an error metric, e, for the evaluation and comparison of streams. Using 6.1, I compute streams specifically optimized for each task, $\mathcal{S}_{\text{[task]-opt}}$, and use its signature to compute the corresponding $\mathcal{S}_{\text{[task]-sig}}$. For a variety of data sets, I compare these streams by evaluating the error as a function of bits per samples (or *bps*), defined as the total number of bits received divided by the total number of samples. To mimic the effects of entropy compression commonly used in practice, I remove from each stream all packets that consist only of leading-zero bits. The wavelet basis allows us to always reconstruct data at full resolution, which greatly simplifies computation of errors, as there exists no standard method to compute error between grids of different dimensions.

6.3.1 Function reconstruction

One of the most fundamental analysis tasks is that of reconstructing the original function itself. A commonly used error metric in this case is the root-mean-square error (RMSE). Figure 6.3 shows a comparison of the different streams for a variety of data sets. It can be noted that, in general, $\mathcal{S}_{\text{rmse-opt}}$ (the stream optimized to minimize the RMSE) performs better than $\mathcal{S}_{\text{rmse-sig}}$ due to spatial adaptivity, whereas $\mathcal{S}_{\text{rmse-sig}}$ slightly outperforms $\mathcal{S}_{\text{wav}}$, followed by $\mathcal{S}_{\text{bit}}$, $\mathcal{S}_{\text{mag}}$, and $\mathcal{S}_{\text{lvl}}$. In particular, $\mathcal{S}_{\text{bit}}$ outperforms $\mathcal{S}_{\text{lvl}}$ (for *kingsnake* and *boiler*, it does so after approximately 1 bps), which can be attributed to the removal of leading-zero packets. Empirically, wavelet coefficients on finer scale subbands are much smaller in magnitude [249]. Such coefficients contain a majority of the leading-zero bits, whose removal benefits $\mathcal{S}_{\text{bit}}$ the most. *diffusivity* and *plasma* contain a significant amount of empty space, which translates to more leading-zero bits after the wavelet transform that $\mathcal{S}_{\text{bit}}$ can take advantage of, and thus, it outperforms $\mathcal{S}_{\text{lvl}}$ immediately from the beginning.

$\mathcal{S}_{\text{mag}}$ underperforms for the same reason that $\mathcal{S}_{\text{lvl}}$ does, but to a lesser extent, since $\mathcal{S}_{\text{mag}}$ is adapted to the data. $\mathcal{S}_{\text{wav}}$ outperforms both $\mathcal{S}_{\text{lvl}}$ and $\mathcal{S}_{\text{bit}}$, because it follows the optimal (data-independent) bit ordering in $\mathbb{L}_{L,B}$ in the L_2 norm, which is also the norm that the

RMSE is based upon. Unsurprisingly, $\mathcal{S}_{\text{rmse-opt}}$ outperforms all the others, as it is the most data-adaptive (i.e., it can optimize packet ordering in the spatial domain in addition to the $\mathbb{L}_{L,B}$ domain). $\mathcal{S}_{\text{rmse-sig}}$ is the second best stream, as it follows the bit ordering of $\mathcal{S}_{\text{rmse-opt}}$ in $\mathbb{L}_{L,B}$ but lacks any spatial adaptivity. In general, $\mathcal{S}_{\text{wav}}$ and $\mathcal{S}_{\text{sig}}$ have similar performances, but $\mathcal{S}_{\text{sig}}$ performs better when the data is less smooth or noisy, as is the case for *boiler* and *kingsnake*. For such data, fine-level packets tend to have very few leading one bits among a majority of leading zero bits, which $\mathcal{S}_{\text{wav}}$ does not take into account (data-independent streams in effect assume every packet contains all one bits).

Let us explore the errors visually by rendering the *plasma* volume at 0.2 bps, for all streams except $\mathcal{S}_{\text{rmse-opt}}$ (Figure 6.4). Although $\mathcal{S}_{\text{lvl}}$ has the precision to obtain an accurate background, it lacks resolution to resolve the fine details. $\mathcal{S}_{\text{bit}}$, instead, lacks the precision to reconstruct the (mostly smooth) background, but it has enough resolution to capture the fine details well. $\mathcal{S}_{\text{wav}}$ balances both precision and resolution, producing a more accurate picture as a whole. In this case, the $\mathcal{S}_{\text{sig}}$ stream produces the most accurate rendering overall.

6.3.2 Derivative computation

Computation of derivative-based quantities is important in data analysis. Examples include vorticity (curl) computation from velocity fields to identify vortical structures, gradient computation for accurate Morse segmentation and shading, and ridge extraction (e.g., for Lagrangian coherent structures). In this work, derivatives are always computed using finite differences, which is common in practice. In this section, I use 32 bits for quantization to ensure enough precision for finite differences. Finite differences are always computed on the finest resolution grid to avoid computing distances between quantities defined on grids of different resolutions.

6.3.3 Gradient computation

Given a function f defined on a grid, its gradient at a grid point $\boldsymbol{x} = (x, y, z)$ is $\nabla f(\boldsymbol{x}) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)$. For accuracy, I use a five-point stencil to compute the gradient, i.e., $\frac{\partial f}{\partial x} \approx \frac{1}{12} f(x-2, y, z) - \frac{2}{3} f(x-1, y, z) + \frac{2}{3} f(x+1, y, z) - \frac{1}{12} f(x+2, y, z)$, but note that the relative performances of the streams stay the same, using the more common two- and three-point formulas. The error between a gradient field ∇f, and its low-bit-rate approximation $\nabla f'$, is defined as $e(\nabla f', \nabla f) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left\| \nabla f'(\boldsymbol{x}_i) - \nabla f(\boldsymbol{x}_i) \right\|^2}$. Using 6.1, I compute a *gradient-optimized* stream, $\mathcal{S}_{\text{grad-opt}}$, that minimizes the difference between the reconstructed and the original gradient fields.

Figure 6.5 shows the gradient error incurred by different streams for four data sets. In general, I observe the ordering of performance (from best to worst) as: $\mathcal{S}_{\text{grad-opt}}$, $\mathcal{S}_{\text{grad-sig}}$, $\mathcal{S}_{\text{bit}}$, $\mathcal{S}_{\text{wav}}$, $\mathcal{S}_{\text{mag}}$, $\mathcal{S}_{\text{lvl}}$. This ordering can also be seen in Figure 6.7, where the x-component of the gradient field for *tuburlence* is rendered at 0.3 bps. Unlike in the RMSE case, $\mathcal{S}_{\text{bit}}$ performs nearly the same as $\mathcal{S}_{\text{wav}}$. To investigate this difference, I extract a 1D line from the *plasma* data set and reconstruct the function using $\mathcal{S}_{\text{bit}}$ and $\mathcal{S}_{\text{wav}}$ at 0.6 bps (Figure 6.6). $\mathcal{S}_{\text{wav}}$'s reconstruction is more accurate on average compared to $\mathcal{S}_{\text{bit}}$, which captures well the function's shape (due to the presence of fine-scale bits), but not the function values (due to the lack of precision in the coarse-scale coefficients). Functions reconstructed with $\mathcal{S}_{\text{bit}}$ tend to be "shifted" in the range domain, as seen in Figure 6.6. However, the gradient operator has the tendency to cancel the shifting effect, bringing the performance of $\mathcal{S}_{\text{bit}}$ closer to that of $\mathcal{S}_{\text{wav}}$.

$\mathcal{S}_{\text{grad-opt}}$ again outperforms the rest of the streams. $\mathcal{S}_{\text{lvl}}$ and $\mathcal{S}_{\text{mag}}$ perform poorly for gradient computation, lacking the resolution to capture sharp features. $\mathcal{S}_{\text{grad-sig}}$ mostly closely follows $\mathcal{S}_{\text{bit}}$ in performance but outperforms it for *boiler*. Again, compared to the other fields, *boiler* is less smooth, resulting in less spatial coherency in the magnitudes of the fine-scale coefficients, which $\mathcal{S}_{\text{grad-opt}}$ and $\mathcal{S}_{\text{grad-sig}}$ can take advantage of, whereas $\mathcal{S}_{\text{wav}}$ or $\mathcal{S}_{\text{bit}}$ do not take into account actual bit values. Overall, the results suggest that besides minimizing RMSE, $\mathcal{S}_{\text{wav}}$ also works well for gradient computation, although for the latter task, $\mathcal{S}_{\text{bit}}$ is also good alternative.

6.3.4 Laplacian computation

The Laplace operator is a second-order differential operator defined as the divergence of the gradient field. The Laplacian of a 3D field is defined as $\Delta f = \frac{\partial^2}{\partial x^2} f + \frac{\partial^2}{\partial y^2} f + \frac{\partial^2}{\partial z^2} f$. Using a five-point finite difference, we can approximate $\frac{\partial^2 f}{\partial x^2} \approx -\frac{1}{12} f(x-2,y,z) + \frac{4}{3} f(x-1,y,z) - \frac{5}{2} f(x,y,z) + \frac{4}{3} f(x+1,y,z) - \frac{1}{12} f(x+2,y,z)$. I use the root-mean-square error to compare two Laplacian fields, i.e., $e(\Delta f', \Delta f) = \text{RMSE}(\Delta f', \Delta f)$. I use 6.1 to compute a *Laplacian-optimized* stream, $\mathcal{S}_{\text{lap-opt}}$, which minimizes e, and an $\mathcal{S}_{\text{lap-sig}}$ stream from its signature. Figure 6.8 plots the errors for all relevant streams. The plots here largely follow the ones in Figure 6.5, in terms of relative performance among the streams, but with more discernible gaps between $\mathcal{S}_{\text{bit}}$ and $\mathcal{S}_{\text{lap-sig}}$, as well as between $\mathcal{S}_{\text{lap-sig}}$ and $\mathcal{S}_{\text{bit}}$. The results suggest that similar to the gradient case, computation of the Laplacian favors resolution over precision, but to a higher degree.

6.3.5 Histogram computation

A histogram succinctly summarizes the distribution of sample values, and thus it is useful as a cursory "look" into the data and in guiding further analysis. For example, it can be used to guide the selection of colors and opacities in a transfer function. To decide on an error metric to compare histograms, I experimented with several popular metrics such as Kolmogorov-Smirnov [266], Kullback-Leibler [144], and Earth Mover's Distance [247], among others [26, 111]. I chose histogram intersection [276] as the metric of choice, because it is fast to compute and is reasonably insensitive to changes in precision, as well as the number of bins. The intersection distance between two histograms H_1 and H_2 is defined as $e(H_1, H_2) = 1 - \sum_i \min\left(H_1(i), H_2(i)\right)$ (the sum is over all bins i). Every histogram is normalized by dividing the value in each bin by the total number of samples. It is decided that the error metric should take into account both the histogram shapes and the range of values, and I clamped the range of values in reconstructed functions to that of the original function, so that corresponding histogram bins, i.e., $H_1(i)$ and $H_2(i)$, share the same range.

As before, for each data set, I use 6.1 to compute an $\mathcal{S}_{\text{hist-opt}}$ stream, optimized for histogram error, and then construct an $\mathcal{S}_{\text{hist-sig}}$ from its signature. I plot the error curves for all relevant streams using the intersection error metric (compare Figure 6.9). I use 64 for the number of bins but note that there exist no meaningful differences across a wide range of number of bins (from 64 to 512) in my experiments. In all cases, the group consisting of $\mathcal{S}_{\text{bit}}$, $\mathcal{S}_{\text{lvl}}$, and $\mathcal{S}_{\text{mag}}$ underperforms the other group by a large margin. $\mathcal{S}_{\text{mag}}$ performs poorly, because it ignores regions of smooth variations, which nevertheless count toward the distribution. $\mathcal{S}_{\text{lvl}}$ generally outperforms $\mathcal{S}_{\text{bit}}$ at low bit rates, although there are several crossover points between the two curves. As can be seen in Figure 6.10, $\mathcal{S}_{\text{lvl}}$ outperforms $\mathcal{S}_{\text{bit}}$ when leading zero packets are present, because increasing resolution does not help as much as increasing precision. This is because the histogram is oblivious to spatial locations of samples (which require resolution to resolve) but is sensitive to sample values (which require precision). However, when leading zero packets are removed, $\mathcal{S}_{\text{bit}}$ benefits significantly more than $\mathcal{S}_{\text{lvl}}$ does (for the same reason explained in Subsection 6.3.1), resulting in the observed crossovers.

In the latter group, the performances of $\mathcal{S}_{\text{wav}}$ and $\mathcal{S}_{\text{hist-sig}}$ (and even $\mathcal{S}_{\text{hist-opt}}$) differ by a negligible amount. This observation is confirmed in Figure 6.11, where I plot various histograms, reconstructed at 0.3 bps, for the *boiler* data set. The histograms produced by $\mathcal{S}_{\text{wav}}$ and $\mathcal{S}_{\text{hist-sig}}$ have approximately the same shape and are the closest to the reference histogram. The next best histogram is produced by $\mathcal{S}_{\text{lvl}}$, followed by the one produced by $\mathcal{S}_{\text{bit}}$, and finally $\mathcal{S}_{\text{mag}}$. These results suggest that histogram computation benefits from a bit ordering that combines both resolution and precision, with a strong bias toward precision.

6.3.6 Isosurface extraction

Studying isosurfaces of a given function is an essential task in many visualization and analysis pipelines, as they can highlight features of interest. For measuring error between isosurfaces, I have found that the commonly used Hausdorff distance does not work well in this case, because two very different reconstructed surfaces may share the same Hausdorff distance to the reference. More sophisticated metrics exist, focusing on different characteristics such as geometric [74] and topological [73] properties, but they assume the surface has certain properties. Since isosurfaces partition the domain into "inside" and "outside" regions, I opt for a simpler error metric that assumes nothing about the shape of the isosurfaces, but simply counts misclassified voxels. This metric differs from comparing histograms with two bins in that we care about the spatial position and not just voxel counts.

However, if the error caused by discarding a packet is of subvoxel resolution, such a metric fails to capture the importance of that packet, causing $\mathcal{S}_{\text{iso-opt}}$ to be ineffective. Therefore, I add the relative difference in surface areas ($|A_1 - A_2|/|A_1|$) to the error term. This additional term is often between $[0, 1]$ and is meant to capture the subvoxel error when the number of misclassified voxels is zero.

With the error metric defined, we can now compute a data-dependent stream optimized for this metric ($\mathcal{S}_{\text{iso-opt}}$) and a stream based on its signature ($\mathcal{S}_{\text{iso-sig}}$) using 6.1 and 6.2. Figure 6.12 compares the performances of these two streams, along with $\mathcal{S}_{\text{bit}}$, $\mathcal{S}_{\text{lvl}}$, $\mathcal{S}_{\text{wav}}$, and $\mathcal{S}_{\text{mag}}$. As can be observed, $\mathcal{S}_{\text{lvl}}$ performs poorly, indicating that isosurface extraction, requires higher levels of resolution compared to histogram computation, as we need to resolve a surface in the spatial domain and not just the range domain of the function. $\mathcal{S}_{\text{wav}}$ and $\mathcal{S}_{\text{iso-sig}}$ typically outperform $\mathcal{S}_{\text{bit}}$, especially at low bit rates. Figure 6.13 renders the isosurfaces reconstructed at 0.6 bps for all streams. In terms of the quality of the reconstructed surfaces, $\mathcal{S}_{\text{iso-sig}} \approx \mathcal{S}_{\text{wav}} > \mathcal{S}_{\text{bit}} > \mathcal{S}_{\text{mag}} > \mathcal{S}_{\text{lvl}}$, which agrees with the plots in Figure 6.12. For isosurface extraction, $\mathcal{S}_{\text{wav}}$ appears to be the only stream — among the data-independent ones — that consistently works well in all cases.

Comparing streams across tasks, it can be observed that histogram-based streams prefer more precision, while derivative-based streams prefer resolution. This is demonstrated in Figure 6.14. The same figure also shows that the primary difference between an optimized stream and a signature stream for the same task is that the optimized stream is able to spatially adapt, which can often result in significantly better bit allocation, and thus better data quality at the same bit rate, compared to the signature stream.

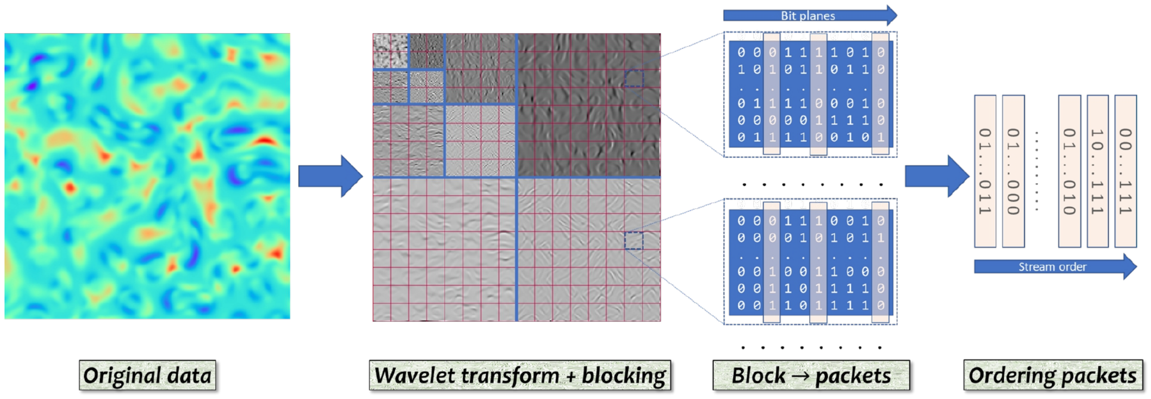
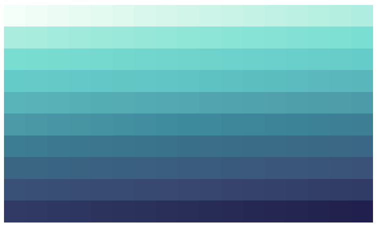
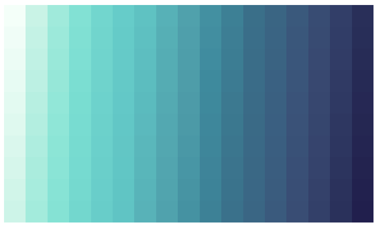


Figure 6.1: My proposed data streaming model. The input is a regular grid of floating-point samples; the output is a stream of *packets*. A packet consists of bits from the same bit plane, from a block of negabinary wavelet coefficients. Different data reduction schemes generate different streams. The wavelet subbands are separated by blue lines in the second image, with the coarsest subband at the top left corner. Quantization and negabinary conversion happen immediately after wavelet transform.


(a) *by level* ($\mathcal{S}_{\text{lvl}}$)


(b) *by bit plane* ($\mathcal{S}_{\text{bit}}$)

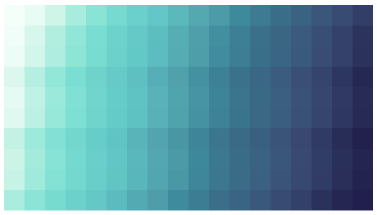
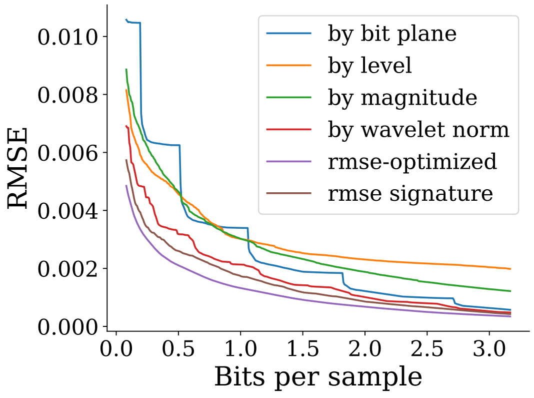
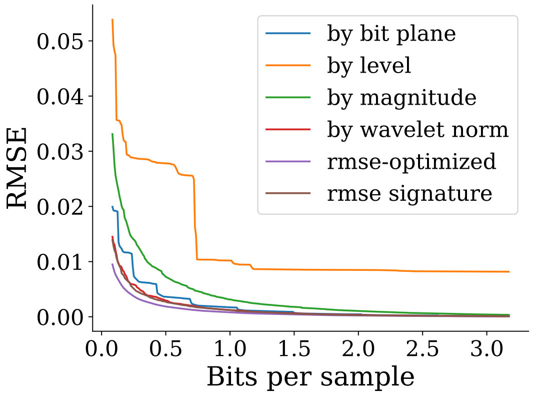

(c) *by wavelet norm* ($\mathcal{S}_{\text{wav}}$)

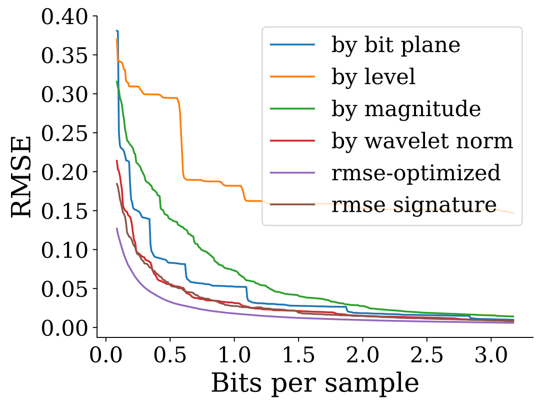
Figure 6.2: Visualization of signatures for $\mathcal{S}_{\text{lvl}}$, $\mathcal{S}_{\text{bit}}$, and $\mathcal{S}_{\text{wav}}$ in 2D. Each signature is a 10×17 image, corresponding to 10 subbands (in 2D) and 17 bit planes. Each (l, b) "cell" contains a unique value from 0 to 169, indicating its "priority" in the stream, and is mapped to a white–blue color scale. $\mathcal{S}_{\text{lvl}}$ streams bits by resolution (from top to bottom), $\mathcal{S}_{\text{bit}}$ streams by precision (from left to right), while $\mathcal{S}_{\text{wav}}$ mixes precision and resolution.



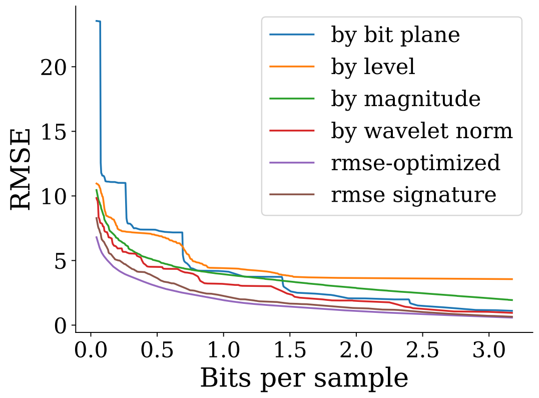
(a) *boiler* $[2.351e{-9}, 0.138]$



(b) *diffusivity* $[-0.848, 0.711]$

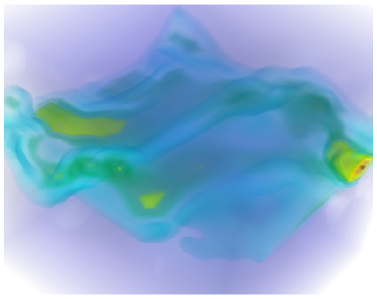


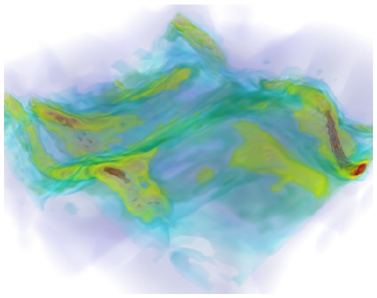
(c) *plasma* $[0.024, 14.67]$

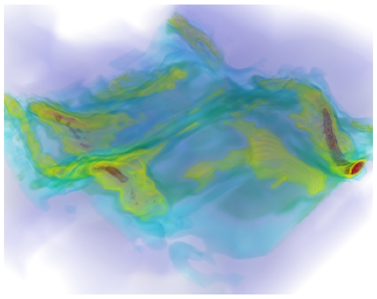


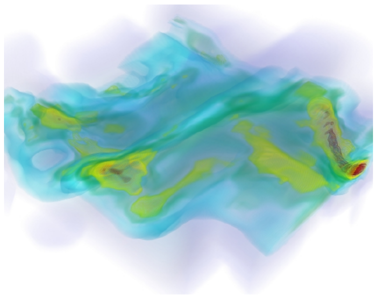
(d) *kingsnake* $[63, 185]$

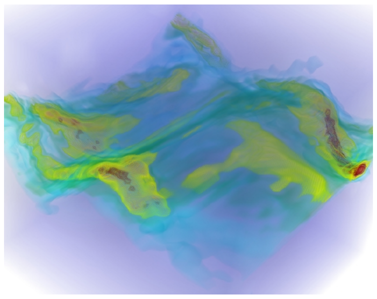
Figure 6.3: Root-mean-square error (RMSE) of reconstructed functions for different streams and data sets; lower RMSE is better. The streams are truncated at both ends to highlight the differences, without omitting important information. The numbers in brackets are the ranges of original data samples. The general ordering of error, from lowest to highest, is $\mathcal{S}_{\text{opt}} < \mathcal{S}_{\text{sig}} < \mathcal{S}_{\text{wav}} < \mathcal{S}_{\text{bit}} < \mathcal{S}_{\text{mag}} < \mathcal{S}_{\text{lvl}}$.


(a) *by level* ($\mathcal{S}_{\text{lvl}}$)


(b) *by bit plane* ($\mathcal{S}_{bit}$)


(c) *by wavelet norm* ($\mathcal{S}_{wav}$)


(d) *by magnitude* ($\mathcal{S}_{mag}$)


(e) *by signature* ($\mathcal{S}_{sig}$)

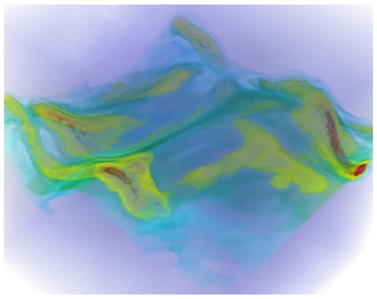
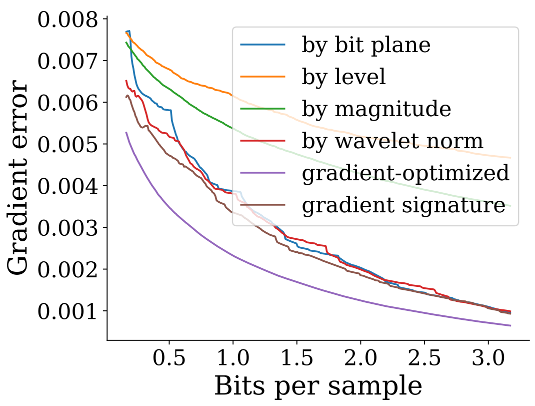
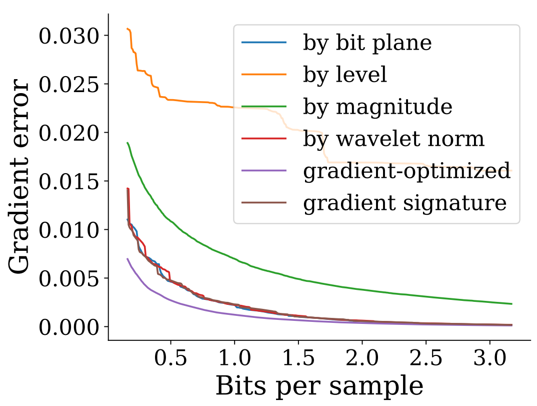

(f) *reference*

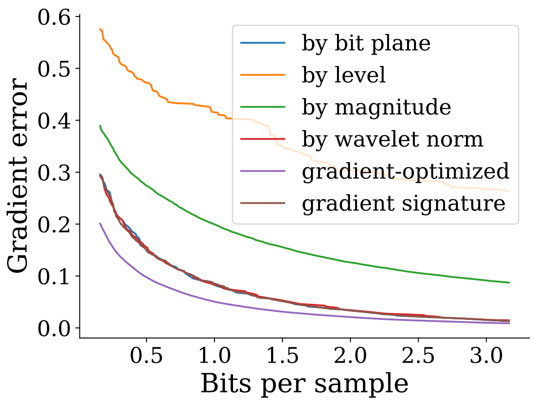
Figure 6.4: Volume renderings of a 64^3 region of *plasma* data set at 0.2 bps. $\mathcal{S}_{\text{lvl}}$ captures the background (purple-blue) well, whereas $\mathcal{S}_{\text{bit}}$ captures the fine details better. $\mathcal{S}_{\text{wav}}$ combines the strength of both. $\mathcal{S}_{\text{sig}}$, however, produces the most accurate rendering in overall.



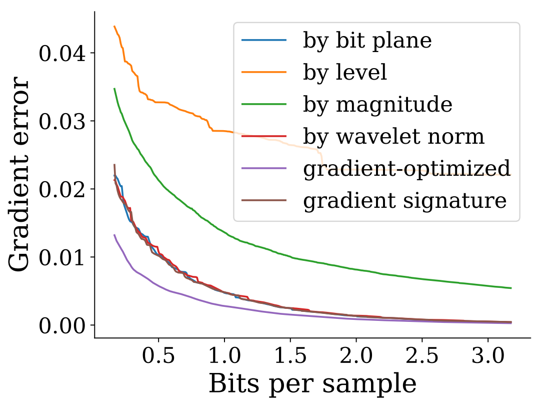
(a) *boiler*, $[2.020e{-9}, 0.148]$



(b) *diffusivity* $[1.063e{-10}, 0.497]$

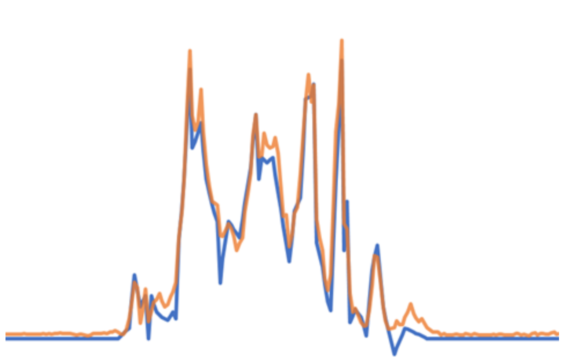


(c) *turbulence* $[0.465e{-3}, 12.19]$



(d) *pressure* $[2.542e{-6}, 0.536]$

Figure 6.5: Gradient error of reconstructed functions. Lower gradient error is better. Leading zero packets are removed, and the plots are truncated in the same way as in Figure 6.3. The numbers in brackets are the ranges of original gradient magnitudes. The trend in error, in all cases, is $\mathcal{S}_{\text{grad-opt}} < \mathcal{S}_{\text{grad-sig}} \approx \mathcal{S}_{\text{bit}} \approx \mathcal{S}_{\text{wav}} < \mathcal{S}_{\text{mag}} < \mathcal{S}_{\text{lvl}}$.


(a) *by bit plane* ($\mathcal{S}_{\text{bit}}$)

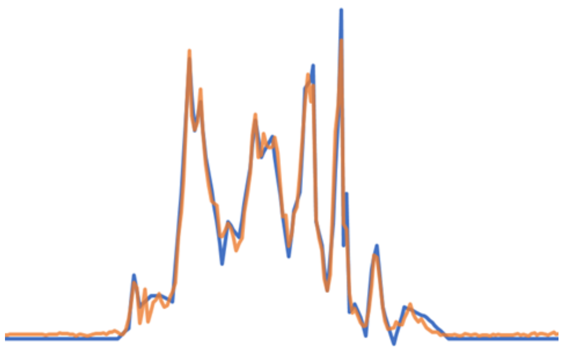
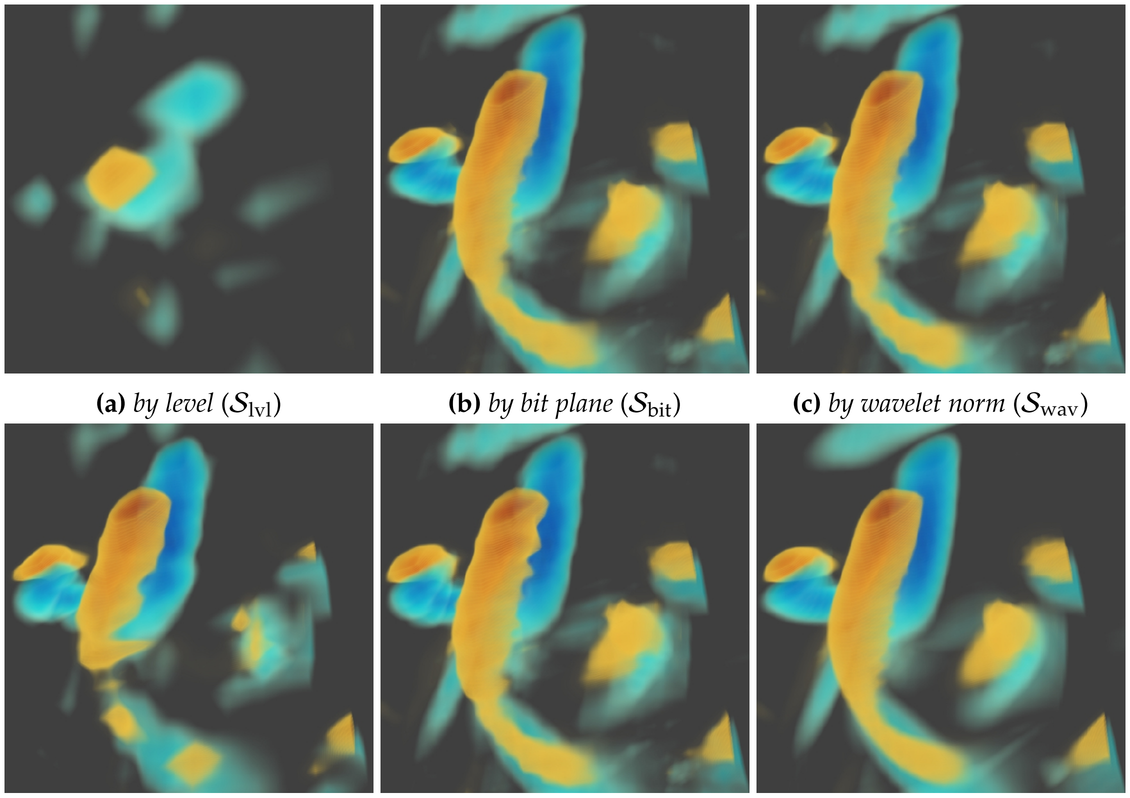
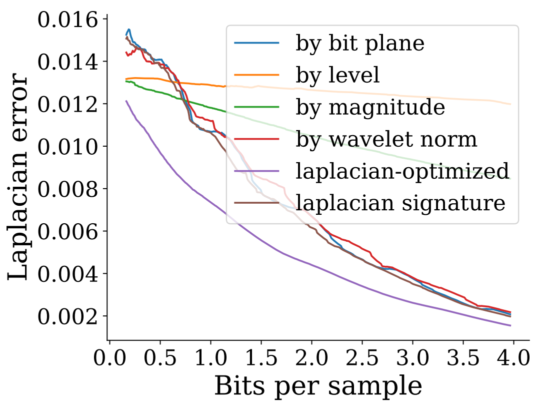

(b) *by wavelet norm* ($\mathcal{S}_{\text{wav}}$)

Figure 6.6: A 1D line extracted from *plasma*, and reconstructed using $\mathcal{S}_{\text{bit}}$ and $\mathcal{S}_{\text{wav}}$ at 0.6 bps. The original data is in orange and the reconstructions are in blue. $\mathcal{S}_{\text{bit}}$ is worse at capturing the function values (seen as a slight vertical shift) but it is comparable to $\mathcal{S}_{\text{wav}}$ in capturing the shape of the function, which is important for gradient computation.

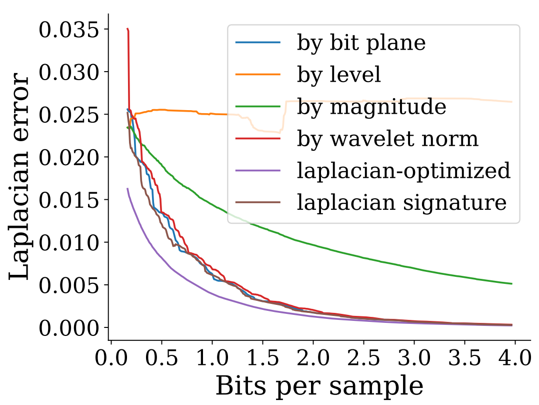

(a) *by level* ($\mathcal{S}_{\text{lvl}}$) **(b)** *by bit plane* ($\mathcal{S}_{\text{bit}}$) **(c)** *by wavelet norm* ($\mathcal{S}_{\text{wav}}$)

(d) *by magnitude* ($\mathcal{S}_{\text{mag}}$) **(e)** *by signature* ($\mathcal{S}_{\text{grad-sig}}$) **(f)** *reference*

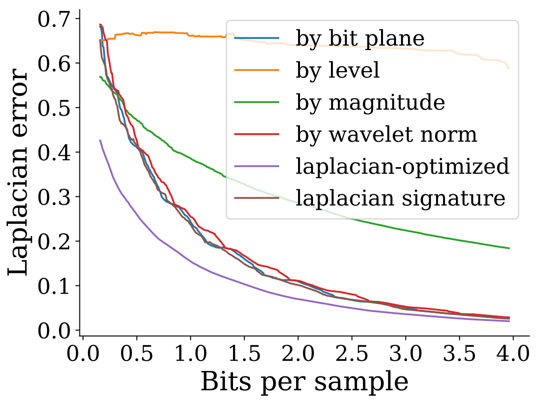
Figure 6.7: The x-component of the (64^3) gradient field of *turbulence*, reconstructed at 0.3 bps. $\mathcal{S}_{\text{bit}}$, $\mathcal{S}_{\text{wav}}$, and $\mathcal{S}_{\text{grad-sig}}$ produce visually comparable gradient fields.



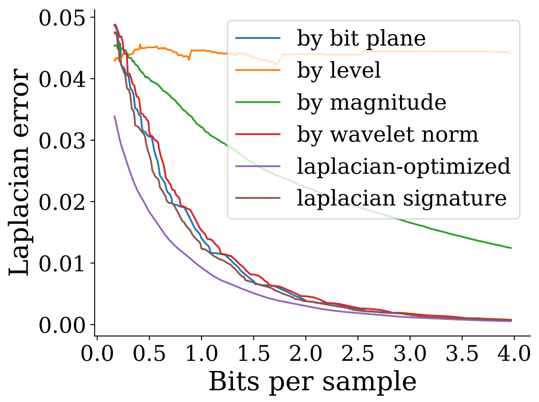
(a) *boiler* $[-0.393, 0.221]$



(b) *diffusivity* $[-0.404, 0.269]$

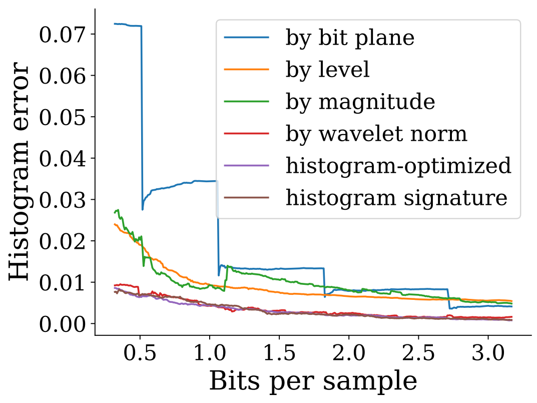


(c) *turbulence* $[-17.44, 11.99]$

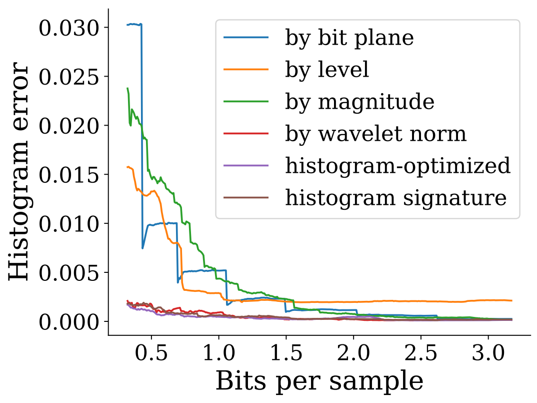


(d) *pressure* $[-0.467, 0.432]$

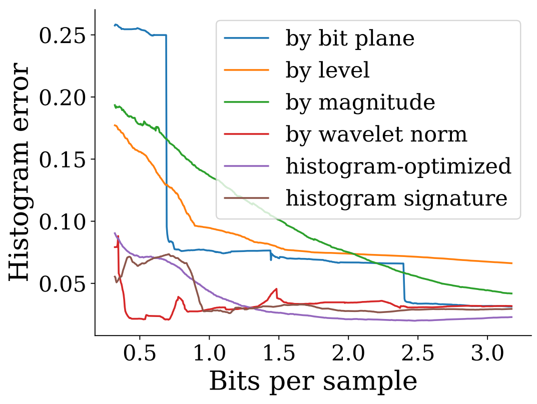
Figure 6.8: Laplacian error comparison among streams. The plots are truncated to better highlight differences without discarding important information. The numbers in brackets are the ranges of the original Laplacian fields. In all cases, in terms of error, $\mathcal{S}_{\text{lap-opt}} < \mathcal{S}_{\text{lap-sig}} < \mathcal{S}_{\text{bit}} < \mathcal{S}_{\text{wav}} < \mathcal{S}_{\text{mag}} < \mathcal{S}_{\text{lvl}}$.



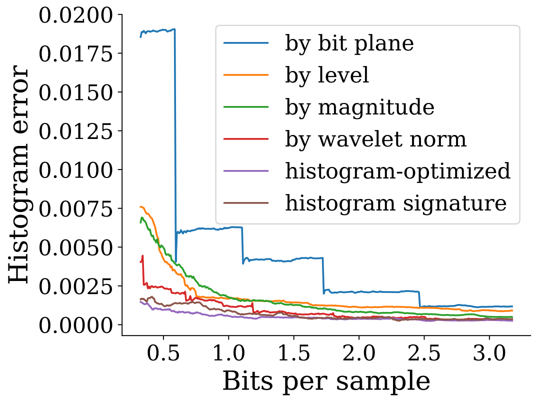
(a) *boiler*



(b) *diffusivity*

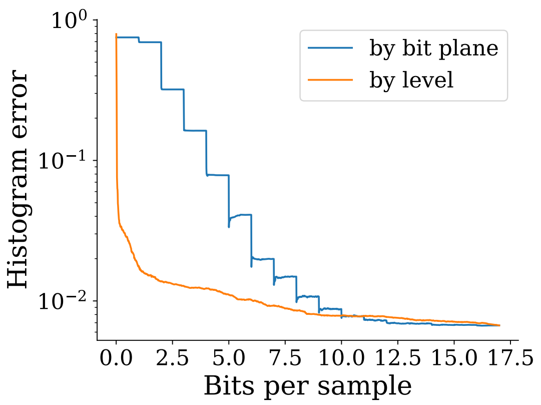


(c) *kingsnake*

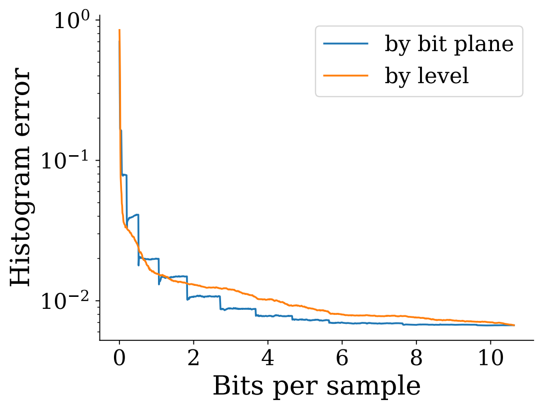


(d) *foam*

Figure 6.9: Comparison of histogram errors among streams. Plots are truncated to highlight differences without hiding important trends. In general, in terms of error, $\mathcal{S}_{\text{hist-opt}} \approx \mathcal{S}_{\text{hist-sig}} \approx \mathcal{S}_{\text{wav}} < \mathcal{S}_{\text{lvl}}, \mathcal{S}_{\text{bit}}, \mathcal{S}_{\text{mag}}$. The erratic behavior at the beginning for *kingsnake* is likely due to the data being too noisy. The especially poor performances of $\mathcal{S}_{\text{bit}}$ for *boiler* and *foam* are due to the "shifting" effect explained in Subsection 6.3.3. Crossover points between $\mathcal{S}_{\text{bit}}$ and $\mathcal{S}_{\text{lvl}}$ are explained in Figure 6.10.

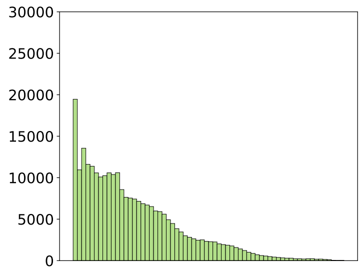


(a) with leading zero packets

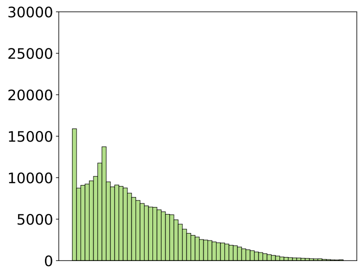


(b) without leading zero packets

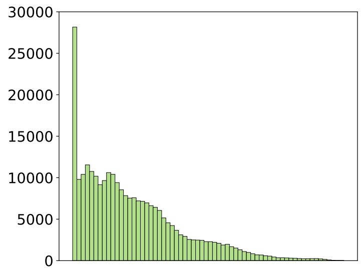
Figure 6.10: Histogram error curves produced by $\mathcal{S}_{\text{bit}}$ and $\mathcal{S}_{\text{lvl}}$, for *boiler*, with and without leading zero bits. The vertical axis is in log scale. The error for $\mathcal{S}_{\text{bit}}$ reduces in a stair-step fashion, where each step corresponds to a new bit plane streamed. $\mathcal{S}_{\text{bit}}$ benefits significantly more from the removal of leading zero bits (the blue curve shifts more to the left).



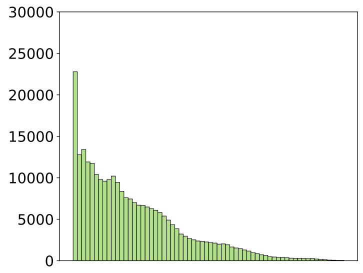
(a) *by level* ($\mathcal{S}_{\text{lvl}}$)



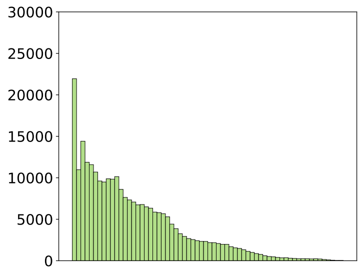
(b) *by bit plane* ($\mathcal{S}_{\text{bit}}$)



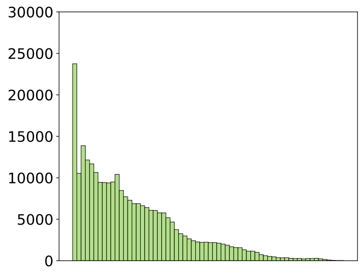
(c) *by magnitude* ($\mathcal{S}_{\text{mag}}$)



(d) *by wavelet norm* ($\mathcal{S}_{\text{wav}}$)

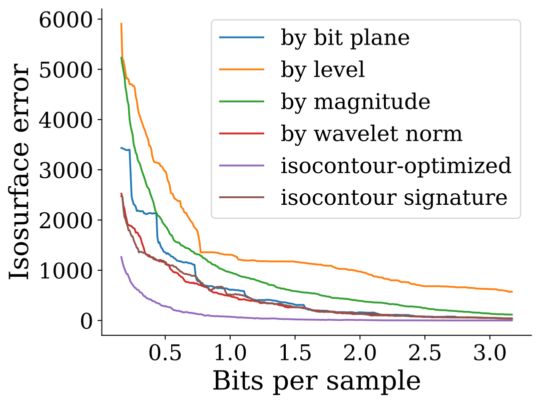


(e) *by signature* ($\mathcal{S}_{\text{hist-sig}}$)

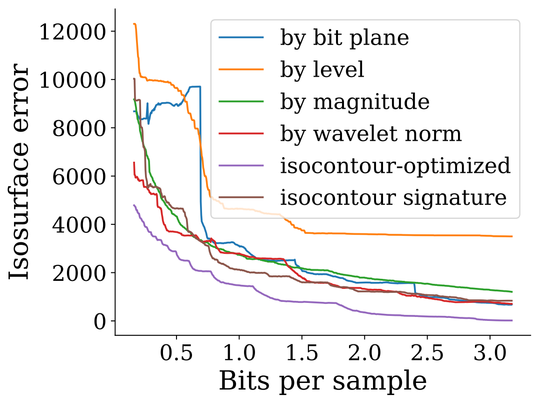


(f) *reference*

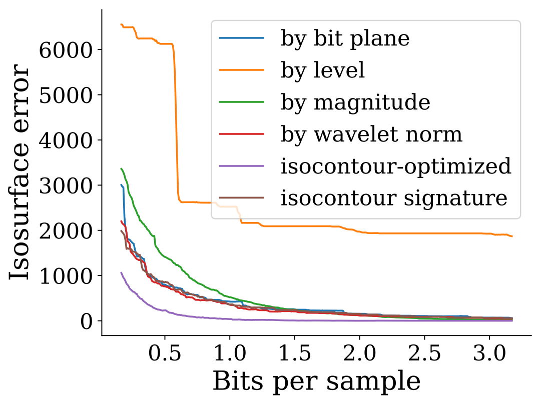
Figure 6.11: Histograms of the *boiler* data set, reconstructed at 0.3 bps. $\mathcal{S}_{\text{lvl}}$, $\mathcal{S}_{\text{wav}}$, and $\mathcal{S}_{\text{hist-sig}}$ produce histograms that share a shape similar to the reference histogram, with most of the peaks and valleys preserved. In contrast, $\mathcal{S}_{\text{bit}}$ produces a spurious peak not found in the reference. Finally, $\mathcal{S}_{\text{mag}}$'s histogram has a widely skewed distribution where too many values fall into the first bin.



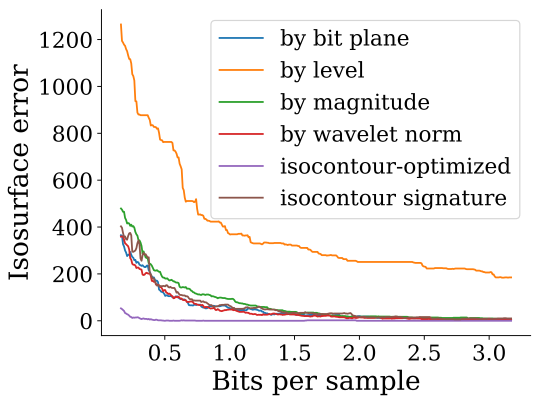
(a) *pressure, #cells = 36149*



(b) *kingsnake, #cells = 45783*

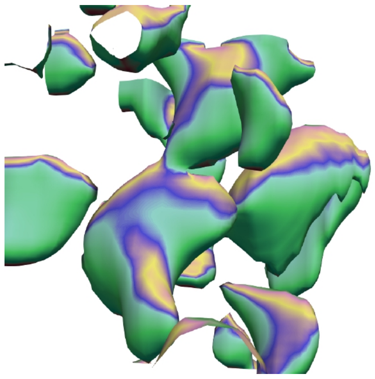


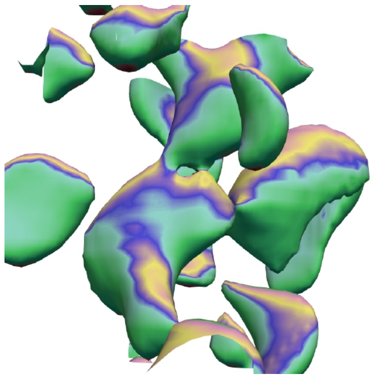
(c) *plasma, #cells = 24856*

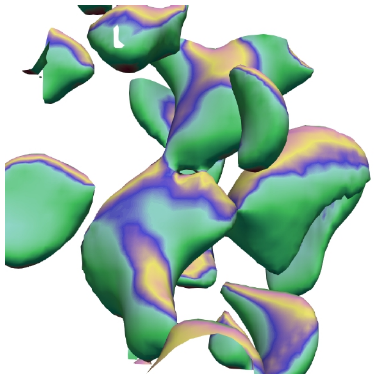


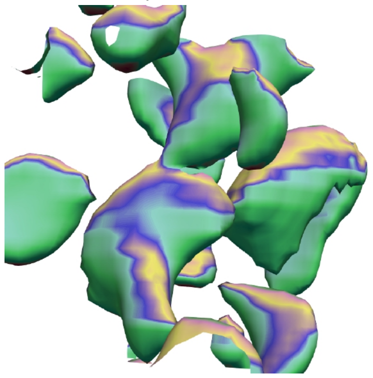
(d) *turbulence, #cells = 33742*

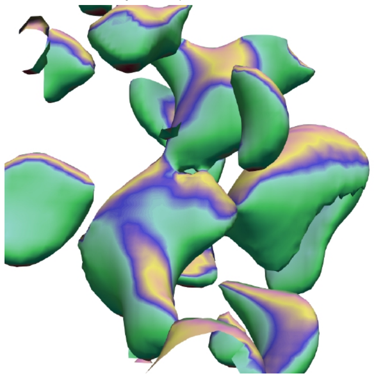
Figure 6.12: Comparison of isosurface errors among streams. Plots are truncated to highlight differences without hiding important trends. The number of cells that each original surface occupies is reported. The trend in error is $\mathcal{S}_{\text{iso-opt}} < \mathcal{S}_{\text{iso-sig}} \approx \mathcal{S}_{\text{wav}} < \mathcal{S}_{\text{bit}} < \mathcal{S}_{\text{mag}} << \mathcal{S}_{\text{lvl}}$.


(a) *by level* ($\mathcal{S}_{\text{lvl}}$)


(b) *by bit plane* ($\mathcal{S}_{\text{bit}}$)


(c) *by wavelet norm* ($\mathcal{S}_{\text{wav}}$)


(d) *by magnitude* ($\mathcal{S}_{\text{mag}}$)


(e) *by signature* ($\mathcal{S}_{\text{iso-sig}}$)

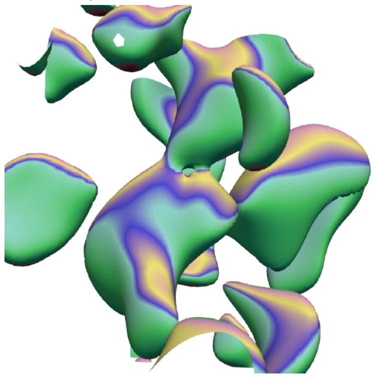
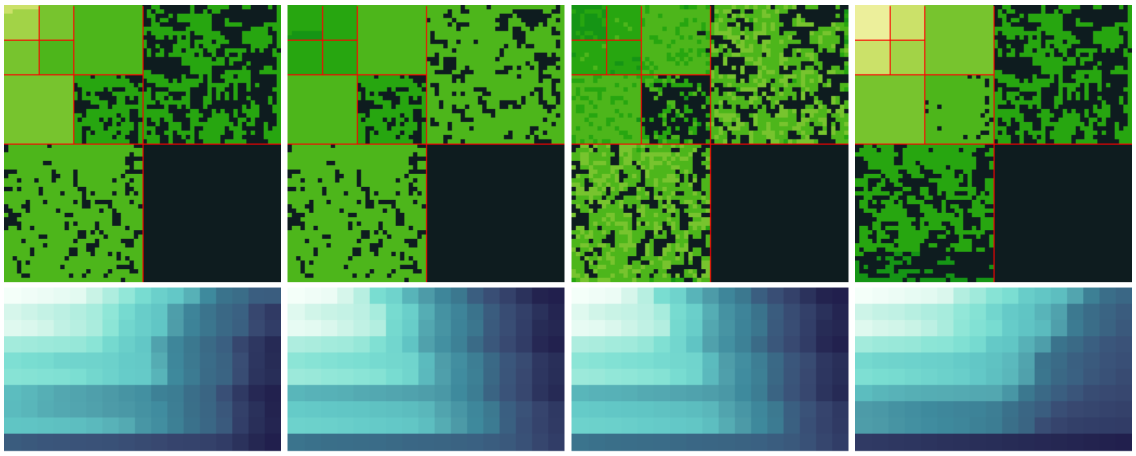

(f) *reference*

Figure 6.13: Rendering of isosurfaces at isovalue of 0.2, at 0.6 bps, for the *pressure* data set. The surfaces are colored by the x-component of the normal vector at each point. $\mathcal{S}_{\text{wav}}$ and $\mathcal{S}_{\text{iso-sig}}$ produce surfaces that are closest to the reference, followed by $\mathcal{S}_{\text{bit}}$, $\mathcal{S}_{\text{mag}}$, and $\mathcal{S}_{\text{lvl}}$.



(a) $\mathcal{S}_{\text{rmse-sig}}$ **(b)** $\mathcal{S}_{\text{lap-sig}}$ **(c)** $\mathcal{S}_{\text{lap-opt}}$ **(d)** $\mathcal{S}_{\text{hist-sig}}$

Figure 6.14: Bit distribution across subbands at 1.6 bps for signature-based streams (top), and corresponding stream signatures (bottom). The data is a 2D slice from *diffusivity*. Subbands are separated by red lines. The color of each pixel indicates the bit plane at which the corresponding coefficient currently is. Brighter greens correspond to more precision. Both the top and the bottom rows show that $\mathcal{S}_{\text{hist-sig}}$ allocates more bits to the lower subbands, while $\mathcal{S}_{\text{lap-sig}}$ prefers to stream bits from higher subbands. $\mathcal{S}_{\text{rmse-sig}}$ is somewhere in the middle. Both $\mathcal{S}_{\text{lap-sig}}$ and $\mathcal{S}_{\text{lap-opt}}$ share the same signature by definition but only $\mathcal{S}_{\text{lap-opt}}$ provides spatial adaptivity, seen as nonuniform colors in each subband.

Table 6.1: I define various types of data streams, including data-independent, data-dependent, task-independent, and task-dependent streams. $\mathcal{S}_{\text{[task]-sig}}$ can be data dependent or task dependent, depending on the stream from which it is derived.

Symbol	Name	Data Dependent	Task Dependent
$\mathcal{S}_{\text{lvl}}$	*by level*	✗	✗
$\mathcal{S}_{\text{bit}}$	*by bit plane*	✗	✗
$\mathcal{S}_{\text{wav}}$	*by wavelet norm*	✗	✗
$\mathcal{S}_{\text{mag}}$	*by magnitude*	✓	✗
$\mathcal{S}_{\text{[task]-opt}}$	*task-optimized*	✓	✓
$\mathcal{S}_{\text{[task]-sig}}$	*by signature*	✓/✗	✓/✗

Table 6.2: All data sets used in experiments. The resolution of data sets is 64^3 and they are subsets of the original volumes (no downsampling performed).

Name	Type	Data type
boiler [268]	combustion simulation	float64
plasma [104]	magnetic reconnection simulation	float32
diffusivity [48]	hydrodynamics simulation	float64
pressure [48]	hydrodynamics simulation	float64
turbulence [62]	fluid dynamics simulation	float32
kingsnake [97]	scan of a snake egg	uint8
flame [126]	combustion simulation	float32
csafe	fluid dynamics simulation	uint8
enzo v [215]	cosmology simulation	float32
brain	microscope image of a marmoset brain	uint8
foam [189]	CT scan of an aluminum foam	uint16
vismale	CT scan of a human	uint8
karfs [235]	combustion simulation	float32
aneurysm	scan of brain aneurysm	uint8
velocity z [48]	hydrodynamics simulation	float64

CHAPTER 7

EFFICIENT TREE-BASED PROGRESSIVE COMPRESSION OF PARTICLES

With rapid advances in computational capabilities, simulations and equipment can generate datasets with trillions of particles [30,225,253], posing serious challenges to studying such datasets for scientific insights. Compression is a promising solution to the problem of ever-expanding particle data. However, no widely accepted compressors for particle data currently exist, and attempts to adapt grid-based compressors for particles [131,279] have seen limited success. Outside of HPC, techniques designed to compress point clouds representing scans of objects [153,194,257] focus largely on improving compression ratios at the expense of scalability in performance, making them unsuitable for large datasets. On the other hand, multiresolution rendering systems [85,245,248,253,258] can handle large data but do not aim for effective compression.

Toward bridging the gap between high compression ratios and low-memory-footprint compression, I introduce novel methods for hierarchy construction, traversal, and encoding that improve on the state-of-the-art tree-based compression methods. I introduce novel tree-based particle compression methods that enable high-quality progressive reconstructions without requiring excessive computational or memory costs. I focus on compressing particle *positions*, since they are needed in almost all applications and, in many applications, are the only attributes needed. Particle positions in scientific applications are difficult to compress losslessly, since they are often specified to such precision that many lower order bits are essentially random. Nevertheless, valuable trade-offs can be made in the space of lossy and progressive (de)compression, in which a decompressor produces approximations that can be progressively refined by decoding more bits that are streamed from the disk or over the network. Progressive decompression allows the user to immediately work with data approximations that improve over time without having to wait for the full data to load or decompress, which can greatly enhance the user experience and accelerate the rate of at which insights are obtained. A progressive decoder can also adapt to the computational resources and time available since decompression can stop as soon as a certain time or data size threshold is reached.

7.1 Tree construction

Here I discuss the method of Devillers and Gandoin [57] (DG), which serves as a base upon which my technical contributions are built. The DG k-d-tree-based coder (implemented in Google's Draco [1]) has competitive compression ratios while being very fast and general, partly due to the coding scheme being nonstatistical (*i.e.,* it does not rely on knowing the distribution of the particles). This method constructs a k-d tree where each node stores the number of particles, n, encapsulated by a bounding box, $\mathbf{B}$. A given node $(\mathbf{B}, n)$ is split into two children $(\mathbf{B}_1, n_1)$ and $(\mathbf{B}_2, n_2)$, with $\mathbf{B}_1$ and $\mathbf{B}_2$ formed by splitting B exactly in the middle along one of the dimensions, and n_1 and n_2 being the numbers of particles bound by $\mathbf{B}_1$ and $\mathbf{B}_2$.

By construction, only n_1 needs to be encoded at each node, since n_2, $\mathbf{B}_1$, and $\mathbf{B}_2$ can be inferred. Furthermore, n_1 can be encoded using approximately $\log_2(n+1)$ bits (since $0 \leq n_1 \leq n$). As n decreases toward the leaf level, the number of bits needed for encoding each node gets smaller, resulting in compression. The tree can be implicitly built, traversed, and encoded at the same time, by having the encoder partition an array of particles inplace, following a certain traversal order, which the decoder also follows. Here, the term *k-d tree* always refers to a tree constructed with this method.

Figure 7.1 gives an example for the DG coder. In their paper, the authors give a theoretical analysis on the number of bits required to separate the particles. Assuming the tree is balanced and every split divides the number of particles in half, on tree depth k, the total number of bits needed is approximately $2^k \log_2(\frac{N}{2^k}+1)$, with N being the total number of particles. The total number of bits needed to separate the particles is therefore $\sum_{k=0}^{\log_2 N-1} 2^k \log_2(\frac{N}{2^k}+1) \leq 2.4N$. Using this result, the paper also gives a lower bound on the number of bits saved using the k-d tree coder compared to verbatim encoding of the particle positions, which is $N\log_2 N$. Since $O(N\log_2 N)$ is also the number of bits needed to encode the relative ordering of the particles (of which there are $N!$), the k-d tree compresses by discarding the original order of particles, on top of compression achieved by quickly separating particles from empty space.

7.1.1 Generic tree-based decoder

I give a generic template for a decoder that can be stopped at any point to produce an approximation to the original particles. The inputs include the total number of particles n_0, an initial bounding box $\mathbf{B}_0$, and a bit stream BS storing the encoded bits. A data structure C, supporting a PUSH and a POP operations (*e.g.*, a stack or queue), keeps track of the traversal front. In each iteration, a front node $(\mathbf{B}, n)$ is popped from C, followed by an optional callback, which, depending on whether $(\mathbf{B}, n)$ is a leaf, can append it to an output list of particles or to a tree. For inner nodes, the number of particles in the left child (*i.e.*, n_1) is decoded from BS with the knowledge of n. $(\mathbf{B}, n)$ is then SPLIT into two children, which are then pushed back into C, and the whole process repeats until either C is empty or when DONE(BS) is true (*e.g.*, when enough bits have been read from BS).

Algorithm 7.1 Generic tree-based decoder. Inputs: n_0 particles in bounding box $\mathbf{B}_0$, bitstream BS, node container C (*e.g.*, stack, queue).

```
 1: function DecodeTree($n_0$, $\mathbf{B}_0$, BS, C)
 2:     C.Push($\mathbf{B}_0, n_0$)                         ▹ push node $(\mathbf{B}_0, n_0)$ to C
 3:     D ← X                                        ▹ initial dimension of splitting
 4:     while not Done(BS) and not C.IsEmpty do
 5:         $(\mathbf{B}, n)$ ← C.Pop
 6:         if $n = 0$ then
 7:             continue
 8:         if IsLeaf($\mathbf{B}, n$) then
 9:             LeafFunc($\mathbf{B}, n$)                ▹ e.g., output a particle
10:             continue
11:         else
12:             InnerFunc($\mathbf{B}, n$)               ▹ e.g., create a tree node
13:         $n_1$ ← Decode($n$, BS)                      ▹ particles in the left child
14:         $n_2$ ← $n - n_1$                            ▹ particles in the right child
15:         $\mathbf{B}_1, \mathbf{B}_2$ ← Split($\mathbf{B}$, D)              ▹ split $\mathbf{B}$ along D
```

```
16:     D ← Next(D)                          ▷ e.g., if D = X, Next(D) = Y
17:     C.Push($\mathbf{B}_1, n_1$)
18:     C.Push($\mathbf{B}_2, n_2$)
```

7.1.1.1 Tree construction

Most tree-based compression techniques work by encoding (and decoding) nodes that implicitly give quantized particle positions. A general template for a tree-based decoder is given in 7.1 in the Appendix. Encoding the number of particles may at first seem wasteful: it has been noted [232, 234] that at coarse levels, occupancy-based octrees are better than the k-d tree used by DG [57], since encoding the number of particles in child nodes often requires several bits compared to at most one bit for occupancy. However, occupancy encoding requires both children of a node to be coded instead of just the left child. Toward the leaf level, past the particle separation stage, when a single bit is needed to encode n_1, this extra cost is significant.

Furthermore, as seen in Subsection 7.2.1, encoding the number of particles also allows us to perform adaptive tree traversal to minimize reconstruction error, which is estimated using the number of particles and the spatial extent of a node. Finally, by knowing the number of particles, I can employ a grid-based approach and switch to encoding the number of empty grid cells when the grid is too dense (*i.e.,* it has more particles than empty cells), which significantly saves coding cost (as discussed in Section 7.1.1.1).

7.1.1.2 Odd-even splits and odd-even trees

When decoding is run to completion, all tree nodes are visited, in an order that depends on the traversal strategy. In practice, however, it is often desirable to traverse and decode large trees only partially to save I/O bandwidth, memory, and decoding time, reconstructing particles whose positions only approximate the original particles' positions. In this context, the shape of the tree and the traversal order can profoundly affect the accuracy of the approximation.

- **Trade-offs between depth-first and breadth-first traversals.** On a traditional k-d tree constructed with the typical *k-d split*, a node's bounding box **B** is split along a certain dimension (one of x, y, z in 3D) to give children boxes $\mathbf{B}_1$ and $\mathbf{B}_2$. BT visits $\mathbf{B}_2$

after $\mathbf{B}_1$ on each tree level, whereas DT only visits $\mathbf{B}_2$ after $\mathbf{B}_1$ has been fully visited. Therefore, when stopped midway, BT often gives coarse representations of the whole space whereas DT reconstructs a spatial region perfectly but completely misses the rest. In most cases, the former behavior is preferred.

DT however is significantly less resource intensive, since it requires only a small stack whose size is at most the height of the tree ($O(\log_2 n)$), whereas BT requires a queue large enough to keep all nodes at the current depth, which can grow as large as the total number of particles ($O(n)$). A k-d split thus offers two contrasting choices: high-cost and coarse reconstructions for both children (with BT), or low-cost and perfect reconstruction for one child but none of the other (with DT). Here, cost mostly means memory footprint, but a high memory footprint often also translates to lower cache utilization and accordingly lower speed.

- **Odd-even splits.** To alleviate the main drawback of DT while retaining its main benefit, I introduce the notion of an *odd-even split*, which spatially "interleaves" the children boxes $\mathbf{B}_1$ and $\mathbf{B}_2$ by having each contain many disjoint slices instead of being a whole contiguous region. This scheme is inspired by the hierarchical indexing scheme [223] and the lazy wavelet transform [277], multiresolution techniques invented for data sampled on regular grids.

  I first impose (but do not build) a regular grid on top of the particles such that each cell contains at most k particle. One way to build such a grid is to recursively subdivide the particles' bounding box into equal halves along the longest dimension, stopping when the target k is met. For the odd-even splitting scheme to work best, k should ideally be 1. However, when particle coordinates are given in floating point, $k = 1$ may produce a grid that is too large if any two particles have almost exactly the same coordinates. In this work I use $k = 1$ in all experiments, but in general k is a parameter that can be set by the user. In addition, to avoid potential rounding errors when multiplying and dividing floating point numbers, I work with quantized particle positions in deciding which grid cell a particle belongs to, but note that the original particles' positions can still be encoded losslessly if needed.

  After the full grid is defined, the root of the tree is associated with the full grid, and every other node is associated with a different subgrid **G** and the particles contained in **G**. If **G** is of dimensions $G_x \times G_y \times G_z$, I index its cells from $(0, 0, 0)$ to $(G_x - 1, G_y - 1, G_z - 1)$, along three fixed axes. An odd-even split decomposes a node $(\mathbf{G}, n)$ into $(\mathbf{G}_e, n_e)$ and $(\mathbf{G}_o, n_o)$, such that $(\mathbf{G}_e, n_e)$ contains the even-indexed cells in **G** (along the dimension of splitting) and the n_e particles occupying those cells, while $(\mathbf{G}_o, n_o)$ contains the rest of the (odd-indexed) cells and particles.

- **Odd-even trees.** A tree constructed exclusively from odd-even splits is called an *odd-even tree*, illustrated in Figure 7.2 in contrast to a k-d tree. The idea of the odd-even split is that either the odd or the even child node represents a coarse approximation of the particle set associated with the parent node, so that a DT can never miss an entire region as with k-d splits. Odd-even trees carry this idea to an extreme where every node is split in the odd-even scheme, and therefore DT on an odd-even tree provides the best coarse-to-fine refinement of the full data with respect to the number of particles reconstructed, but not in terms of coding cost (or compression ratio) which will be discussed later.

- **Odd-even subsampling.** Picking either the odd or the even subgrid to traverse can be viewed as a subsampling method. It may at first seem that random subsampling (*e.g.,* as done in [288]) achieves the same effects as odd-even subsampling while being simpler. However, unlike random sampling, odd-even sampling produce subgrids ($\mathbf{G}_o$ and $\mathbf{G}_e$) that are half the size of the parent grid **G**, which is important for locating the particles using fewer bits. Unfortunately, an odd-even tree is still not conducive to compression. This is due to the fact that for most datasets, particles do not scatter randomly in space but form clusters and structures that can be well separated from empty cells. With odd-even splits, the empty cells are "distributed" into the odd and even subtrees, effectively increasing the number of tree nodes to be coded. Instead, k-d splits could be used to quickly cull away entire empty subtrees (as can be seen in Figure 7.3).

- **Coding costs.** For a more quantitative analysis, I calculate the number of bits required to locate, using a k-d tree, n particles in a grid **G** with G cells, of which n contain particles and $G - n$ are empty. Denote the answer as $T(n, G)$. In the best case, a k-d split will put most particles in one child and empty space in the other, leading to $T(n, G) = (\log_2 n) + T(n, G/2)$, *i.e.,* n stays the same but G is reduced by half. After $i = \log_2(G/n)$ such iterations, $G/2^i$ and n are approximately equal, *i.e.,* $G/2^i < 2n$. The k-d tree now requires $\approx 2.4n$ bits to separate the n particles, and an additional n bits to finally locate the particles (assuming at the leaf level, each particle needs to be separated from one empty cell). In contrast, an odd-even split implies a different recurrence relation: $T'(n, G) = (\log_2 n) + 2T'(n/2, G/2)$ (*i.e.,* both n and G are halved but two substrees are created instead of one). After the particles are separated from one another (after $\approx 2.4n$ bits), each particle needs to be further located among G/n cells, for a cost of $n \log_2(G/n)$ additional bits. Therefore, the difference between $T(n, G)$ and $T'(n, G)$ is that between $n + \log_2 n \log_2(G/n)$ (for the k-d tree) and $n \log_2(G/n)$ (for the odd-even tree). The two are similar if G is close to n (the grid is dense in particles), but in most cases, G is significantly larger than n, making the odd-even tree worse. In experiments, I have seen odd-even trees that are almost twice as large as a k-d tree for the same input. I later discuss a solution in Section 7.1.1.3.

- **Encoding dense particle grids.** Besides facilitating the odd-even splits, an underlying grid allows us to effectively encode sparse as well as dense particle sets (relative to the size of the grid). Whenever the number of particles, n, is greater than half the number of cells in **G**, the algorithm can switch from encoding the number of particles in the left child (n_1) to encoding the number of empty cells in the left child, *i.e.,* $G/2 - n_1$, and thus more quickly bound the values to be encoded further down the tree. In the extreme case where every cell contains a particle, my method simply stops after encoding the number of particles at the root node, since the number of empty cells is now 0, whereas other methods, such as DG [57] or MPEG [257] which encodes node's occupancy, will keep refining this dense grid until the individual cells.

- **Tree traversal as particle indexing.** To decode particles' positions by traversing a tree is to reconstruct the bits of their quantized integer coordinates, or, equivalently, to index (order) the particles using their coordinate bits. It is well known that a k-d tree sorts the particles using their Morton codes, which interleave particle coordinate bits in x, y, z, with an interleaving pattern that depends on the order of the dimensions along which nodes are split. In other words, a k-d tree reconstructs the interleaved coordinate bits from left to right (MSB to LSB) if traversed using DT, whereas an odd-even tree reconstructs them from right to left (LSB to MSB) (see Figure 7.4), since the LSB determines whether a particle is “odd” or “even”.

7.1.1.3 Hybrid trees

As seen in 4.1, odd-even trees create too many nodes because every odd-even split distributes both the particles and the empty cells into two children, instead of (mostly) particles in one and empty cells in the other. To reduce this adverse impact on compression while retaining most of their benefits, it is necessary to reduce the use of odd-even splits. Here, I borrow a technique from the wavelet literature, where multiresolution decomposition is done by recursively transforming only the low-pass filtered subband in every iteration. Similarly, I restrict the use of odd-even splits to only left child nodes, with k-d splits used everywhere else. Furthermore, once a k-d split is used, subsequent descendant splits will all be k-d splits. I also use the convention that the left child node is $(\mathbf{G}_e, n_e)$, *i.e.,* it contains the even-indexed cells. The dimension of splitting is the largest dimension of the parent's grid $\mathbf{G}$, as is also the case for all the other trees discussed in this chapter. Constructed this way, the impact of the hybrid trees on compression is minimal; in the worst case, I have noticed only a 5% increase in compressed size compared to k-d trees.

- **Resolution levels.** From top to bottom, every odd-even split creates a new, coarser *resolution level*, which consists of nodes in the even-indexed subtree. A hybrid tree with L resolution levels contains a sequence of exactly $L - 1$ odd-even splits, at nodes found by traversing down the left child $L - 2$ times from the root (see Figure 7.5a for an example with $L = 3$). L can be automatically set so that the chain of odd-even splits ends when no particles or cells are left to split. Assuming that left children are always visited first, DT on hybrid trees visits the resolution levels from coarse to fine, producing a “breadth-first” walk of space similar to BT on k-d trees but with much a smaller memory footprint.

  Although hybrid trees are designed with DT in mind, they also support BT, noting that BT is best used only within each resolution level and not across resolution levels (note that nodes at the same *depth level* may belong to different *resolution levels*, see, *e.g.*, Figure 7.5a). My proposed hybrid tree is also only one of the many possible combinations of k-d and odd-even splits, which may be useful for different purposes.

- **Particle indexing.** From the perspective of particle indexing using their interleaved quantized coordinates, hybrid trees correspond to the hierarchical Z (HZ) ordering [223] of particles. An HZ ordering sorts the particles first by their resolution level, then by their index within the level. This is done by swappping the least significant one bit in a particle's Morton code (whose position from the LSB determines the particle's resolution level) and the bits to its left (which constitute the particle's intra-level index). Figure 7.4 gives an example of this scheme. The HZ indexing scheme was first proposed by Pascucci and Frank [223] and generalized by Hoang et al. [115] for multiresolution decomposition of regular grids. Here, I adapt the scheme to construct a multiresolution particle tree.

- **Refinement bits.** Refinement bits are bits that further locate each particle in its corresponding cell (see Figure 7.5, gray nodes at the bottom). All refinement bits at the same tree depth form a *bit plane*. The number of refinement bit planes vary across datasets. For some, there are no refinement bits, *i.e.*, particles are specified with precision low enough that the particles are exactly located just by the grid that separates them. In the other spectrum, scientific simulation data are often dominated by refinement bits due to the particles being specified with relatively high precision compared to their density. For hybrid trees, the refinement bits are stored in depth-first order: particles are completely refined one by one, in the (depth-first) order that they appear in the tree. For example, in Figure 7.5a, the refinement bit stream is $10(a)11(b)00(c)10(d)01(e)11(f)00(g)00(h)10(i)10(j)11(k)$.

- **Coding costs.** As in Section 7.1.1.2, let $T(n, G)$ denote the number of bits needed to locate n particles in G grid cells using a k-d tree. For a hybrid tree, the number of bits to code a subtree under node $(\mathbf{G}, n)$ is $(\log_2 n) + T_l(n/2, G/2) + T_r(n/2, G/2)$. The term $T_r(n/2, G/2)$ (for the right subtree) is just $T(n/2, G/2)$ since the right subtree is always a k-d tree. The term $T_l(n/2, G/2)$ (for the left subtree) can again be decomposed into $\log_2(n/2) + T_l(n/4, G/4) + T(n/4, G/4)$. Following the recurrence to the end and ignoring the various $\log_2(n/2^i)$ terms that are insignificant, it can be seen that the cost of encoding the whole hybrid tree is approximately $\sum_i T(n/2^i, G/2^i)$. Since $T(n, G)$ is approximately linear in n (see Section 7.1.1.2), $T(n, G) = 2T(n/2, G/2)$. The sum $\sum_i T(n/2^i, G/2^i)$ therefore is approximately just $T(n, G)$, meaning the coding cost of a hybrid tree is approximately the same as that of a k-d tree for the same input (the difference is of order $O(\log_2 n \log_2(G/n))$ bits which is essentially the number of resolution levels multiplied by the cost to cull the empty cells on each level). This analysis also shows that if a hybrid tree is traversed with DT, so that the resolution levels are visited from coarse to fine, the (partial) coding cost doubles after each resolution level as the number of particles, n, presumably also doubles.

- **Reconstruction error.** To quantify the reconstruction error for each original particle, I find the nearest particle to it among the reconstructed particles. I give an upper bound for the reconstruction error in both cases: BT on a k-d tree (BT-kd) and DT on a hybrid tree (DT-hybrid). Suppose the two particles form two opposite corners of a box of dimensions $d_x \times d_y \times d_z$, the values of d_x, d_y, d_z can be bounded using k, understood to be either the number of tree depths not yet traversed (for BT-kd), or the number of resolution levels not yet traversed (for DT-hybrid). For both cases, it is guaranteed that $d_x d_y d_z \leq w_x w_y w_z 2^k$, with w_x, w_y, w_z being the dimensions of each cell at the leaf level. From the analysis in the **coding costs** paragraph, it is known that the total coding cost for the k-d tree is approximately the same as that for the hybrid tree. Moreover, this cost doubles after each tree depth level (for the k-d tree) and after each resolution level (for the hybrid tree). Therefore, BT-kd and DT-hybrid tree have similar coding costs as well as similar reconstruction error bounds. Note that when all particles are located to their respective cells, the reconstruction error is bounded by the dimensions of a cell *i.e.*, $w_x \times w_y \times w_z$, and each refinement bit plane reduces this bound by half.

  In terms of reconstruction error, the main difference between the two schemes is that DT-hybrid puts the reconstructed particles exactly where the corresponding original particles are, whereas BT-kd puts them in the middle of the bounding boxes at the traversal front. Depending on the dataset, one choice may be preferred over another (Section 7.4). Because both schemes have approximately the same total coding costs but DT-hybrid partially reconstructs particles to a higher precision, it also tends to generate significantly fewer particles compared to BT-kd when both are stopped midway at the same decoding bit budget.

- **Memory footprint with DT.** I do not explicitly construct the tree in memory, as node values are simply encoded to and decoded from a bit stream, following a certain traversal order. Therefore, only the size of the data structures used for traversal, and not that of the tree itself, counts toward the memory footprint. Let G denote the total number of grid cells and N the total number of particles. Using DT, a hybrid tree can be traversed using a stack whose size is the bounded by the height of the tree, which is $\log_2 G + 1$. For each element in the stack, $\log_2 (N+1)$ bits are needed to keep track of the number of particles. The other information required for traversal, namely a node's associated grid, its resolution level, the dimension of splitting, and the type of split can all be deduced from the path connecting the node to the tree's root, encoded with $\log_2 G + 1$ bits. The encoder (but not decoder) would also need to keep track of the range of particles that each node encompasses, for a total of $\log_2 (G+1)$ additional bits per node. In short, the memory footprint of the encoder is $(\log_2 G + 1)^2 + \log_2 (N+1)$ bits, while that of the decoder is $(\log_2 G + 1)^2$ bits. Even when $N = G = 2^64 - 1$, both the encoder and the decoder require trivial amounts of memory (less than 600 bytes).

- **Hybrid tree encoding.** I give the pseudocode for a hybrid tree encoder in 7.2. Compared to 7.1, this algorithm is written in recursive form for simplicity, includes details on when to use odd-even splits, and how the algorithm switches to encoding empty cells if the current grid is dense in particles (lines 9 – 11). A new EncodeRefinementBits step is invoked to output a particle's in-cell refinement bits (*i.e.,* gray nodes in Figure 7.5). Finally, I further specify the actual low-level encoding method to be the commonly used truncated binary coding [202, 284].

Algorithm 7.2 Depth-first, recursive hybrid tree encoding. Inputs: a set of particles P residing in a grid **G**, split type s which is either ODD-EVEN or K-D (it is ODD-EVEN initially),

dimension of splitting D, and bitstream BS.

```
1: function DtHybridEncode($P$, $\mathbf{G}$, S, D, BS)
    ▹ step 1: encode in-cell refinement bits
2:     if $G = 1$ then                                         ▹ one particle in a single cell
3:         EncodeRefinementBits($P$, $\mathbf{G}$, D, BS)
4:         return
    ▹ step 2: split the current node
5:     $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2 \leftarrow$ Partition($P$, $\mathbf{G}$, S, D)
    ▹ step 3: encode the current node
6:     $n \leftarrow |P|$                                       ▹ number of particles in current node
7:     $n_1 \leftarrow |P_1|$                                   ▹ number of particles in the left child
8:     if $G - n < n$ then                                     ▹ grid is dense in particles
9:         $n \leftarrow G - n$                                 ▹ encode number of empty cells
10:        $n_1 \leftarrow |\mathbf{G}_1| - n_1$
11:    TruncatedBinaryEncode($n_1, n$, BS)
    ▹ step 4: recurse
12:    D $\leftarrow$ Next(D)                                  ▹ next dimension of splitting
13:    if $|P_1| > 0$ then                                     ▹ left child
14:        DtHybridEncode($P_1, \mathbf{G_1}$, S, D, BS)
15:    if $|P_2| > 0$ then                                     ▹ right child
16:        S $\leftarrow$ K-D                                  ▹ by definition of hybrid-trees
17:        DtHybridEncode($P_2, \mathbf{G_2}$, S, D, BS)
18:
19: function Partition($P$, $\mathbf{G}$, S, D)
20:    $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2 \leftarrow \varnothing$
21:    if S = ODD-EVEN then
22:        $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2 \leftarrow$ OddEvenPartition($P, \mathbf{G}$, D)
23:    else
24:        $P_1, P_2\mathbf{G}_1, \mathbf{G}_2 \leftarrow$ KdPartition($P, \mathbf{G}$, D)
25:    return $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2$
```

7.1.2 Block-hybrid trees

A problem wit hybrid trees is that each resolution level is still traversed region-by-region, resulting in uneven error distribution. To mitigate this problem, I split the whole tree into multiple blocks (subtrees) and interleave their traversals to reduce error more uniformly. The resulting *block-hybrid tree* contains multiple blocks that can be decoded independently. By adaptively allocating bits across blocks, I can lower the overall reconstruction error, or prioritize certain blocks for the task at hand. Furthermore, blocks can be randomly accessed or decoded in parallel.

To construct a block-hybrid tree, I first use k-d splits to form a *coarse* portion (a k-d subtree at the top), then combine k-d splits with odd-even splits to form a *medium* portion (several hybrid subtrees), and finally use the in-cell refinement bits to form a *fine* portion. Each leaf of the coarse-portion k-d subtree creates a hybrid sub-tree, or *block*. The coarse and medium portions together refine the full grid until at least the cell level (*i.e.,* no leaf node contains more than one particle). The fine portion further locates individual particles within the respective cells. Figure 7.5b shows an example of a block-hybrid built for the running example with 11 particles in 2D. From a particle indexing perspective, block-hybrid trees use hierarchical-Z indexing for the middle portion, and forward Morton for the rest (see Figure 7.4).

- **Refinement bits.** Since one main goal of block-hybrid trees is to distribute reconstruction error more uniformly in space, I store the in-cell refinement bits verbatim in bit plane order, *i.e.,* in breadth-first instead of depth-first order as done for hybrid trees. In particular, each bit plane contains one refinement bit for each particle in the block, in the order that a medium-phase DT visits the particle. To decode each refinement bit, a bounding box that encompasses the current particle needs to be subdivided. To avoid buffering such bounding boxes for later refinement in typical breadth-first manner, I compute them on-the-fly from the current position of each particle and the dimensions of grid cells at the current tree depth. Therefore, no extra memory is needed in addition to an array storing the positions of the decoded particles, which is presumably always present.

- **Flexible decoding.** A block-hybrid tree, once encoded, can be decoded in different ways; in particular, the blocks can be decoded independently and to different extents. To support independent decoding, the compressed bit streams for individual blocks are stored separately (see Figure 7.6). Decoding of higher resolution particles can also be skipped in favor of more refinement bits for lower resolution ones. Such a strategy, which trades resolution for precision, may be desired if the number of output particles needs to be limited due to resource constraints. Since blocks are encoded in independent bit streams, a decoder can freely jump to any block to continue decoding/traversal if needed. The user can also supply a scoring function to rank blocks during traversal. In Subsection 7.2.2, I introduce one such function, which interleaves traversal of blocks to lower the average reconstruction error. Other criteria are possible, for example, during rendering, certain blocks may be prioritized if they are closer to the camera, or since they are known to contain features of interest.

7.2 Tree traversal

To achieve better progressive reconstruction than BT and DT, the traversal should be more adaptive, *i.e.,* nodes with a potentially low cost of traversal (in terms of number of bits to decode) and high gains (in terms of reduction of error) ought to be prioritized. I introduce two such adaptive orders: *adaptive traversal* (AT) for k-d/hybrid trees and *block-adaptive traversal* (BAT) for block-hybrid trees.

7.2.1 Adaptive traversal

For k-d trees, I generalize the container c in 7.1 from either a stack or a queue to a priority queue, which allows us to perform rate-distortion optimization during traversal, *i.e.,* prioritizing nodes that are more important with respect to some error metric and coding cost. Concretely, the importance score of a given node $(\mathbf{B}, n)$ is

$$\frac{n(d/2)^2}{\log_2(n+1)}, \tag{7.1}$$

where d is the length of $\mathbf{B}$ along the current axis of splitting. The denominator captures the cost of decoding the current node, while the $(d/2)^2$ term captures the (squared) error reduction per-particle obtained by decoding this node, assuming the extreme case where all n particles fall into either the left or the right child. Intuitively, spatially larger and denser nodes are prioritized so that reconstruction errors are reduced for more particles. I expect AT with this heuristic to work best (compared to BT) when the particles are highly nonuniformly distributed, and therefore the importance scores of same-depth nodes are notably different.

- **Alternative score function.** My proposed importance score is simple yet works well in practice to improve the rate-distortion trade-off over BT for a wide range of datasets (see Section 7.4). Regardless, this score is still a heuristic and thus is not guaranteed to work for all datasets. I also demonstrate modifications to the importance function by reducing the emphasis on node density (*i.e.,* by removing n from the numerator), which I have observed to work better for particles representing a surface. I anticipate, that in future work, many more importance functions can be devised depending on the data and task at hand, but all should be supported by AT.

7.2.2 Block-adaptive traversal

Although AT improves on BT in reconstruction quality, it has a similarly high memory footprint in practice (see Subsection 7.4.3). Furthermore, AT works with individual nodes and not blocks, so it cannot be used as is to efficiently traverse a block-hybrid tree. Here, I generalize AT to *block-adaptive traversal* (BAT), which is also data-adaptive but works with entire blocks of nodes, and has asymptotically constant memory footprint similarly to that of DT.

With BAT, the coarse portion of a block-hybrid tree is traversed with either BT or AT. Traversal of the medium portion only begins after traversal of the coarse portion completes. The medium portion is traversed in iterations in a data-dependent round-robin manner. Each iteration consists of two steps: first, a block is picked to traverse using a priority queue that ranks blocks based on some criterion, then, the chosen block is traversed using DT. The block at the top of the priority queue is traversed for either a certain number of decoded bytes or a certain number of particles, then its priority is updated in the queue, and the process repeats with the next iteration.

- **Heuristic for ranking blocks.** The ranking of blocks is handled by a user-supplied scoring function; here I propose one. Between two partially decoded blocks, I always prioritize the one at a coarser resolution level. If the two blocks are at the same resolution level, the one with a smaller value of n^*_l/n_l is prioritized, where n_l is the total number of particles in the block on resolution level l, and n^*_l is the number of those already visited by the per-block DT. The idea behind this heuristic is to distribute reconstruction error across blocks as uniformly as possible, so that the average error is reduced.

- **Reconstruction error.** By construction, nodes on the same resolution level are associated with subgrids with the same internal spacing (*i.e.,* spatial distances in x, y, z between neighboring cells). This spacing gives an upper bound on the reconstruction error, since particles that fall in between neighboring cells in the current subgrid have not been reconstructed (they belong to finer resolution levels). Therefore, forcing the blocks to be refined to the same resolution level effectively forces approximately the same upper bound for reconstruction error everywhere. Once the blocks are at the same resolution level, the ratio n^*_l/n_l indicates how much of the given level has been traversed.

- **Memory footprint.** Given a tree of height H, the memory footprint of BAT is controlled by H_c, the height of the coarse k-d subtree. Since there are at most 2^{H_c-1} blocks and each block contains a **s**of size at most $H - H_c$, the total number of elements in the different containers is bound by 2^{H_c-1} (**q**) $+2^{H_c-1}(H - H_c)$ (**s**s) + 2^{H_c-1} (priority queue). In contrast, the size of the queue for BT, if used exclusively for the whole hierarchy, is bounded above by 2^{H-1}, which is often several orders of magnitude larger than 2^{H_c-1}, since a typical H_c is only half of H. In practice, the H_c chosen should be large enough so that the error is more uniformly distributed and that random access to the blocks is more fine-grained, but also small enough to not turn BAT into BT and also to not create too many blocks.

- **Block-adaptive traversal.** The pseudocode for BAT is given in 7.3. B denotes the current block being traversed, with B.l being the block's resolution level, B.n its total number of particles and B.n^* its number of visited particles, as described above. In the CoarseTraverse and MediumTraverse functions, N denotes the current node at the traversal front, with N.l storing its resolution level and N.n its number of particles. The

FineTraverse function performs fine-phase (breadth-first) traversal, which decodes the in-cell refinement bits.

Algorithm 7.3 Block-adaptive traversal. Inputs: a priority queue of blocks q, coarse bit stream bs.

```
1:  function BlockAdaptiveTraverse(q, bs)
2:      if q.IsEmpty then                                   ▷ still in coarse traversal phase
3:          CoarseTraverse(cs)
4:      while not q.IsEmpty do
5:          b ← q.Pop                                       ▷ b is the block with the largest error
6:          if b.$n^*$ < b.$n$ then                          ▷ not all of b's particles visited
7:              MediumTraverse(b)
8:              q.Push(b)
9:          else                                            ▷ all particles visited, do fine traversal phase
10:             FineTraverse(b)
11:             if not AllBitsRead(b.bs) then
12:                 q.Push(b)
13:
    ▷ Coarse-phase traversal: Algorithm 1 with a queue container, bitstream bs and the
    following callback.
14: function CoarseTraverse(bs)
    ▷ LeafFunc                                              ▷ from each leaf we create a block b
15:     b.$l$ ← 0                                           ▷ with resolution level 0,
16:     b.$n$ ← n.$n$                                       ▷ and appropriate number of particles,
17:     b.$n^*$ ← 0                                         ▷ and number of visited particles
18:     b.st ← empty stack
19:     b.st.Push(n)                                        ▷ n is the root node of block b
20:     q.Push(b)
21:
    ▷ Medium-phase traversal: Algorithm 1 with b.st as the (stack) container, b.bs as the
    bitstream, and the following callbacks.
22: function MediumTraverse(b)                              ▷ DT of block b
```

```
    ▷ LeafFunc                                              ▷ each leaf is a particle
        $b.n^* \leftarrow b.n^* + 1$                        ▷ a new particle visited
        $b.n^*_l \leftarrow b.n^*_l + 1$
    ▷ InnerFunc
        if $n.l \neq b.l$ then                              ▷ reached a finer resolution level
            $b.l \leftarrow n.l$                            ▷ update the current resolution level
            $b.n_l \leftarrow n.n$                          ▷ update number of particles
            $b.n^*_l \leftarrow 0$                          ▷ reset number of visited particles

    ▷ Fine-phase traversal: for simplicity, always read an entire bit plane of the input block b.
function FineTraverse(b)
    for each particle $p \in b.P$ do
        bit ← ReadOneBit(b.bs)
        p ← Refine(p, bit)
    $b.l \leftarrow b.l + 1$
```

7.3 Encoding node values

During decompression, at a node $(\mathbf{G}, n)$, a decoder needs to decode n_1 with the knowledge of n and the fact that $0 \leq n_1 \leq n$. The state-of-the-art method [57] uses arithmetic coding [200] or truncated binary coding [202,284], assuming that n_1 is uniformly distributed in $\{0, \ldots, n\}$. However, this assumption is often not true in practice, and thus better encoding methods are possible. I present two such methods here that better predict n_1, namely a nonstatistical *binomial coding* scheme and a statistical *odd-even context coding* scheme, targeting two extreme particle distributions: uniform and highly structured.

7.3.1 Binomial Coding

For data that exhibit approximately uniformly spatial distribution of particles, n_1 is not uniformly distributed in $\{0, \ldots, n\}$ but is more likely to be close to $\frac{n}{2}$ — a property that I will exploit to improve the encoding. Given a node with n particles, there are 2^n possible configurations (each of the n particles can fall in either of the two child nodes with probability $\frac{1}{2}$), and there are $\binom{n}{n_1}$ ways for the left child node under consideration to contain exactly n_1 particles out of the n particles of the parent. Therefore, n_1 follows the binomial distribution with parameters n and $\frac{1}{2}$ (see Figure 7.7), *i.e.*, $P(n_1|n) = \binom{n}{n_1}2^{-n} = B(n, \frac{1}{2})$.

- **Arithmetic coding for small n.** For small values of n, this binomial distribution can be effectively coded using arithmetic coding [200] with a precomputed binomial table. My arithmetic coder supports integer probabilities whose sum is at most 2^{31}, which means the distribution $B(n, \frac{1}{2})$ is exactly modeled for $n \leq 30$. For every $n \in \{1, \dots, 30\}$, I precompute a table where the entries are $\binom{n}{n_1}$ for every $n_1 \in \{0, \dots, n\}$ (I scale $P(n_1|n)$ by 2^n to represent the probabilities with integers). I then compute a prefix sum on each table to obtain a (scaled) cumulative distribution function (CDF) ready to be used by the arithmetic coder.

- **Binary-search coding for large n.** When $n > 30$, arithmetic coding with exact probabilities fail because the arithmetic coder uses 32-bit values for its internal states, which have insufficient precision to distinguish all possible values of $P(n_1|n)$, since the scaled CDF grows exponentially with n, *i.e.*, $\sum_{n_1=0}^{n} \binom{n}{n_1} = 2^n$. Note that simply using 64-bit internal states would not solve the problem, due to potential integer overflows under multiplications. Instead, I leverage the *de Moivre-Laplace* theorem [61] to approximate the binomial distribution with a Normal distribution for large n, *i.e.*, $B(n,p) \simeq N(np, np(1-p))$, where N is the Normal distribution with mean $\mu = np$ and variance $\sigma^2 = np(1-p)$). When $p \approx \frac{1}{2}$, *i.e.*, assuming an approximately uniform distribution of particles, the theorem states that $P(n_1|n)$ follows $N(\frac{n}{2}, \frac{n}{4})$.

  Denoting the CDF of $N(\frac{n}{2}, \frac{n}{4})$ as F, I use a binary search that locates n_1 by halving F in the search range $[a_i, b_i]$ for each iteration i, outputting a bit to indicate which half contains n_1. The point of division can be computed using the inverse of F, namely $F^{-1}(x) = \frac{n}{2} + \sqrt{\frac{n}{2}}\,\mathrm{erf}^{-1}(2x-1)$, where erf is the error function. Initially, $[a_0, b_0] = [0, n]$, and the search stops when either the value is found (*i.e.*, $n_1 \leq a_i < b_i < n_1 + 1$ for some i) or the range stops converging, indicating that we run out of numerical precision. In the latter case, assuming equal probabilities for all values in $[a_i, b_i]$, $n_1 - a_i$ can be encoded using truncated binary coding. I use a *mid-short* (or *centered-minimal*) code [202] that assigns shorter codewords for values near the middle of $[0,\dots,b_i - a_i]$. The pseudocode for the binomial encoder is given by 7.4 in the Appendix.

- **Code size gain over truncated binary coding.** The theoretical gain achievable with binomial coding can be assessed using the entropy of the binomial distribution, *i.e.,* $H \simeq \frac{1}{2}\log_2(2\pi e n p(1-p))$ (the full derivation is in Appendix 7.3.1). For $p = \frac{1}{2}$, I get $H \simeq \frac{1}{2}\log_2(2\pi \mathrm{e}\frac{n}{4}) \approx 1 + \frac{1}{2}\log_2 n$. The normalized entropy (dividing H by $\log_2 n$) thus approaches $\frac{1}{2}$ as n tends to infinity and 1 as n tends to 1. In contrast, the normalized entropy of the uniform distribution is always 1, which means that in the best case (when n is very large) binomial coding reduces the code length by half compared to uniform arithmetic coding or truncated binary coding. This gain makes binomial coding attractive for large data. Finally, binomial coding also works well with odd-even splits, because such splits tend to produce approximately equal numbers of particles on the two sides, regardless of the actual particle distribution.

- **Binomial coding and entropy.** To get a better understanding of the theoretical gain achievable with binomial coding, I look at the entropy of the binomial distribution, H, which is

$$H = -\sum_{k=0}^{n}\binom{n}{k}p^k(1-p)^{n-k}\log_2\left(\binom{n}{k}p^k(1-p)^{n-k}\right)$$

I use the de Moivre-Laplace theorem to get

$$H \simeq -\int_{-\infty}^{\infty}\frac{dx}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}\log_2\left(\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}\right)$$

$$\simeq -\int_{-\infty}^{\infty}\frac{dx}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}\left(-\log_2(\sigma\sqrt{2\pi}) - \frac{(x-\mu)^2}{2\sigma^2}\log_2 e\right)$$

By the definitions of a normal distribution and its variance, we have, respectively, $\int_{-\infty}^{\infty}\frac{dx}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} = 1$ and $\int_{-\infty}^{\infty}\frac{dx}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}(x-\mu)^2 = \sigma^2$. Therefore,

$$H \simeq \log_2(\sigma\sqrt{2\pi}) + \frac{1}{2}\log_2 e = \frac{1}{2}\log_2(2\pi e\sigma^2)$$
$$= \frac{1}{2}\log_2(2\pi e n p(1-p))$$

Algorithm 7.4 Binomial coding. Inputs: CDF tables CDFS, arithmetic bit stream BS_a, truncated binary bit stream BS_b, number of particles in the current node n, number of particles in the left child n_1.

```
1:  function BinomialEncode(CDFS, $\text{BS}_a$, $\text{BS}_b$, $n, n_1$)
2:      $\mu \leftarrow n/2$
3:      $\sigma^2 \leftarrow n/4$
4:      small $\leftarrow n \leq 30$
5:      while True do
6:          $a \leftarrow \lceil a \rceil$
7:          $b \leftarrow \lfloor b \rfloor$
8:          if $a = b$ then                                ▷ $n_1 = a = b$, no need to encode
9:              return
10:         $n \leftarrow b - a$
11:         if small then                                  ▷ exact binomial modeling
12:             ArithmeticEncode($n_1 - a$, CDFS[$n$], $\text{BS}_a$)
13:             return
14:         $f_a \leftarrow F_{\mu,\sigma^2}(a)$                   ▷ $F_{\mu,\sigma^2}$ is the CDF of $N(\mu, \sigma^2)$
15:         $f_b \leftarrow F_{\mu,\sigma^2}(b)$
16:         $m \leftarrow F^{-1}_{\mu,\sigma^2}((f_a + f_b)\ /2)$
17:         if $m = a$ or $m = b$ then                    ▷ ran out of precision
18:             TruncatedBinaryEncode($n, n_1 - a$, $\text{BS}_b$)
19:             return
20:         if $n_1 < m$ then
21:             WriteBit($\text{BS}_b$, 0)
22:             $b \leftarrow m$
23:         else
24:             WriteBit($\text{BS}_b$, 1)
25:             $a \leftarrow m$
```

7.3.2 Odd-even context coding

For datasets where the particles are not approximately uniformly distributed – and thus binomial coding does not apply – I propose a prediction scheme based on DT on a hybrid tree to improve compression. Since the even (left) and odd (right) subtrees under an odd-even split interleave spatially, they can have very similar distributions of particles (Figure 7.8). I can therefore leverage their spatial correlations and use one to predict the other. An odd-even split creates an odd and an even subtree, denoted as $\mathbf{T}_o$ and $\bar{\mathbf{T}}_e$, respectively. I use different notations to indicate that $\mathbf{T}_o$ is always a k-d tree, while $\bar{\mathbf{T}}_e$ is almost always a hybrid tree by definition. Using DT, $\bar{\mathbf{T}}_e$ is trqversed and coded first, and then used as a reference to predict $\mathbf{T}_o$.

- **Lock-step traversal.** Since $\bar{\mathbf{T}}_e$ and $\mathbf{T}_o$ are different kinds of tree, I first need to transform $\bar{\mathbf{T}}_e$ to a k-d tree $\mathbf{T}_e$. It is done by invoking a k-d tree building routine on the cells to which the particles of $\bar{\mathbf{T}}_e$ have been located. The k-d trees $\mathbf{T}_e$ and $\mathbf{T}_o$ can now be traversed in lockstep using DT to maintain spatial correlations between respective nodes at the traversal front. After $\mathbf{T}_o$ is fully coded, the hybrid subtree combining $\mathbf{T}_e$ and $\mathbf{T}_o$ is converted to a k-d tree to serve as the reference for the next resolution level (Figure 7.9). 7.5 in the Appendix gives the full pseudocode for the odd-even context encoder. Note that I never explicitly create and store any of $\bar{\mathbf{T}}_e$, $\mathbf{T}_e$ and $\mathbf{T}_o$ in memory. The conceptual transformation of $\bar{\mathbf{T}}_e$ from an odd-even to a k-d subtree can be performed inplace (by partitioning the array storing the input particles) and inline (computing node values for $\mathbf{T}_e$ on-the-fly as $\mathbf{T}_o$ is traversed).

- **Context coding.** During the lockstep DT, the traversal front typically contains two nodes: $(\mathbf{G}_s, s)$ on $\mathbf{T}_e$ and $(\mathbf{G}, n)$ on $\mathbf{T}_o$ (see Figure 7.9). If the number of particles in the left child of $(\mathbf{G}_s, s)$ and $(\mathbf{G}, n)$ are s_1 and n_1, respectively, then n, s, and s_1 are known, while n_1 needs to be coded. To predict n_1 using n, s and s_1, I leverage context-based arithmetic coding [200], in which the knowledge of a vector $\mathbf{c}_1$ (the context) helps narrow down the possible values for n_1. I do not use $\mathbf{c}_1 = [n, s, s_1]$ to encode n_1 directly, since any of these numbers can be so large that keeping track of all possible contexts is impractical. Instead, I work with the log values, namely $m = \lfloor \log_2 (n+1) \rfloor, m_1 = \lfloor \log_2 (n_1+1) \rfloor, r = \lfloor \log_2 (s+1) \rfloor$, and $r_1 = \lfloor \log_2 (s_1+1) \rfloor$.

The use of log values also make all contexts more reliable, since each context now appears enough times to be statistically significant. However, in place of n_1, we now must encode both m_1 and $m_2 = \lfloor \log_2 (n_2 + 1) \rfloor$, with n_2 being the number of particles in the right child of the current node (*i.e.,* $n_2 = n - n_1$). The reason is that in general m_2 in general cannot be inferred from m and m_1, except in a few special cases, namely when $m = m_1 = 1, m_2 = 0$, and when $m_1 = 0, m_2 = m$.

- **Context update** To encode m_1, the context vector $\mathbf{c}_1$ contains more information than just $[m, r, r_1]$. In particular, $\mathbf{c}_1 = [m, r, r_1, r_2, l, h]$, with r_2 being the log of the number of particles in the right child of the reference node (*i.e.,* $r_2 = \lfloor \log_2 (s - s_1 + 1) \rfloor$), l being the current node's resolution level and h being its current tree depth. I use a *context table* H to maintain and update the conditional probabilities $P(m_1|\mathbf{c}_1)$ on the fly for all combinations of m_1 and $\mathbf{c}_1$ encountered during traversal. H is a hashtable, that, when indexed with a key $\mathbf{c}_1$, return a frequency array that gives the conditional distribution of m_1 given $\mathbf{c}_1$, *i.e.,* $P(m_1|\mathbf{c}_1) = \mathrm{H}[\mathbf{c}_1][m_1] \ / \sum_i \mathrm{H}[\mathbf{c}_1][i]$.

  During traversal and coding, $\mathrm{H}[\mathbf{c}_1][m_1]$ is incremented whenever the $(\mathbf{c}_1, m_1)$ pair occurs. However, since $\mathbf{T}_e$ and $\mathbf{T}_o$ in general have different shapes, a full context may not exist, in which case I fall back to the shorter context $[m, l, h]$ for m_1. When a ($\mathbf{c}_1$, m_1) pair occurs for the first time, $\mathrm{H}[\mathbf{c}_1][m_1] = 0$ and thus m_1 cannot be coded using $\mathbf{c}_1$. I instead encode an empty symbol at index -1 with frequency 1 (*i.e.,* $\mathrm{H}[\mathbf{c}_1][-1] = 1$) to signify to the decoder that $\mathbf{c}_1$ cannot be used, then encode m_1 with uniform probability *i.e.,* $1/(m+1)$. At the same time, $\mathrm{H}[\mathbf{c}_1][m_1]$ is still incremented to avoid this zero-probability problem the next time the same $(\mathbf{c}_1, m_1)$ pair occurs. Finally, m_2 is also encoded with a context, which combines $\mathbf{c}_1$ and m_1, since m_1 is already known before m_2 is decoded.

- **Odd-even context coding.** Algorithm 7.5 gives the pseudocode for the odd-even context encoder, including details on how to perform the lockstep traversal inline and inplace.

Algorithm 7.5 Odd-even context encoding. The inputs P, D, $\mathbf{G}$, S, BS$_a$, BS$_b$ are the same as in previous algorithms. l and h are, respectively, the resolution level and tree depth of the

current node during the recursive DT. R is a sorted array containing particles of the current reference subtree, and $\mathbf{G}_R$ is the corresponding subgrid where these particles reside.

```
1: function OeEncode($R$, $\mathbf{G}_R$, $P$, $\mathbf{G}$, d, s, $l$, $h$, bs$_a$, bs$_b$)
   ▹ first two steps are the same as 7.2 ...
   ▹ step 3: encode the current node
2:     $m \leftarrow \log_2(n+1)$
3:     $m_1 \leftarrow \log_2(n_1+1)$
4:     $m_2 \leftarrow \log_2(n-n_1+1)$
5:     $R_1, R_2, \mathbf{G}_{R_1}, \mathbf{G}_{R_2} \leftarrow$ KdPartition($R$,$\mathbf{G}_R$,d)
6:     $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2 \leftarrow$ Partition($P$,$\mathbf{G}$,s,d)          ▹ see Alg. 2
7:     if $|R| > 0$ then                                   ▹ reference node is present
8:         $r \leftarrow \log_2(|R|+1)$
9:         $r_1 \leftarrow \log_2(|R_1|+1)$
10:        $r_2 \leftarrow \log_2(|R_2|+1)$
11:        $\mathbf{c}_1 \leftarrow [m, r, r_1, r_2, l, h]$
12:        ContextEncode($m_1$, $\mathbf{c}_1$, bs$_a$, bs$_b$)
13:        if CannotInfer$m_2$From($m, m_1$) then
14:            $\mathbf{c}_2 \leftarrow [m, r, r_1, r_2, l, h, m_1]$
15:            ContextEncode($m_2$, $\mathbf{c}_2$, bs$_a$, bs$_b$)
16:    else                                   ▹ no reference node, minimal context
17:        $\mathbf{c}_1 \leftarrow [m, l, h]$
18:        ContextEncode($m_1$, $\mathbf{c}_1$, bs$_a$, bs$_b$)
19:        if CannotInfer$m_2$From($m, m_1$) then
20:            $\mathbf{c}_2 \leftarrow [m, l, h, m_1]$
21:            ContextEncode($m_2$, $\mathbf{c}_2$, bs$_a$, bs$_b$)
   ▹ step 4: recurse
22:    d $\leftarrow$ Next(d)                               ▹ next axis of splitting
23:    $h \leftarrow h+1$                                   ▹ next tree depth
24:    if $|P_1| > 0$ then                                  ▹ left child
25:        $l_n \leftarrow l + ($s = odd-even$)$                ▹ next resolution level
26:        $R_1 \leftarrow ($s = odd-even$)$ ? $\varnothing$
```

```
27:         $\text{OeEncode}(R_1, \mathbf{G}_{R_1}, P_1, \mathbf{G}_1, \textsc{d}, \textsc{s}, l_n, h, \textsc{bs}_a, \textsc{bs}_b)$
28:     if $|P_2| > 0$ then                                           ▷ right child
29:         $(R_2, \mathbf{G}_{R_2}) \leftarrow (\textsc{s} \neq \textsc{k-d})\ ?\ (R_1, \mathbf{G}_{R_1})$
30:         $\text{OeEncode}(R_2, \mathbf{G}_{R_2}, P_2, \mathbf{G}_2, \textsc{d}, \textsc{s}, l, h, \textsc{bs}_a, \textsc{bs}_b)$
31:

    ▷ encode $m_1$ using context $\mathbf{c}_1$ into bit streams $\textsc{bs}_a$, $\textsc{bs}_b$
32: function $\text{ContextEncode}(m_1, \mathbf{c}_1, \textsc{bs}_a, \textsc{bs}_b)$
33:     if $\text{H}[\mathbf{c}_1][m_1] > 0$ then                           ▷ context can be used
34:         $\text{ArithmeticEncode}(m_1, \text{H}[\mathbf{c}_1], \textsc{bs}_a)$
35:     else                                ▷ context cannot be used due to 0 probability
36:         $\text{H}[\mathbf{c}_1][-1] \leftarrow 1$
37:         $\text{ArithmeticEncode}(-1, \text{H}[\mathbf{c}_1], \textsc{bs}_a)$
38:         $\text{TruncatedBinaryEncode}(m_1, \mathbf{c}_1.m, \textsc{bs}_b)$
39:     $\text{H}[\mathbf{c}_1][m_1] \leftarrow \text{H}[\mathbf{c}_1][m_1] + 1$
```

7.4 Evaluation

I evaluate the efficacy of the proposed solutions through various experiments. In the discussion that follows, both “BT on k-d tree” and “DT on k-d tree” are the baseline DG [57] methods; all other traversal-tree combinations are my contributions. I quantify the reduction in data as *bits-per-particle* (bpp), measured by dividing the number of bits decoded by the total number of particles originally. Particle are always specified using 32-bit floating point coordinates, which are then quantized to 32-bit (96 bpp) integers prior to experiments. To generate an approximation when a traversal stops midway, for each node $(\mathbf{G}, n)$ at the traversal front, one (random) particle is generated within $\mathbf{G}$. I use |c| to refer to the size of container(s) used for traversal, in terms of number of elements.

I use both the standard peak-signal-to-noise ratio (PSNR) and rendered images, when appropriate, to assess the quality of partial reconstructions. PSNR is defined as $20\log_{10}(W/E)$, where E is the root mean square point-wise distance between every reference particle and its closest reconstructed particle, and W is the maximum dimension of the bounding box for the reference particles. A PSNR or 50 dB means that E is about $\frac{W}{300}$, and an improvement of 1 dB corresponds to a reduction of E by 10 percent.

7.4.1 Adaptive traversal of k-d trees

AT (with the proposed scoring heuristic, Equation 7.1) on k-d trees improves the rate-distortion trade-off over BT on k-d trees for a wide range of datasets (see Figure 7.10). I do not include DT in the same figure since the root-mean-square error for DT is often exceptionally high due to whole regions missing, rendering L_2-norm-based quality metric such as PSNR less meaningful. Visual demonstration of the differences between low-bit-rate reconstructions using BT and AT is provided in Figure 7.11 (see the first green-highlighted column pair). I render at low bit rates the outputs of the various traversal and tree combinations with OSPRay [295]. The bit rates are chosen so that visual differences among the combinations are most apparent. For the *girl* dataset, AT (a3) provides a better covering of space compared to BT (a2), which follows a strict order on each tree depth level, creating a visible seam where the resolution changes. The same phenomenon occurs for *fissure* (comparing b2 and b3). For *soldier*, although less noticeable, AT (d3) generates a smoother surface as well. For *cosmic web*, AT (f3) captures the points of interest — clusters of particles (galaxies) — better by favoring densely packed nodes. Overall, by being more data-adaptive, AT can provide significant improvements over BT, both visually and quantitatively (in PSNR).

- **Alternative AT.** The default scoring function for AT (Equation 7.1) does not always work well for all datasets. For example, the rendering of the *coal* dataset (which contains simulated coal particles) in Figure 7.12 contains occlusion because particles on the "surface" are given more importance. Because of occlusion, however, the majority of particles in dense tree nodes are hidden from view, but these are also nodes that the scoring function deems important. To improve visual quality, I instead use an alternative scoring function, removing n from the numerator, to prevent an overemphasis on dense nodes. The result is a reconstruction with lower PSNR but improved visual quality (*i.e.,* more similar to the reference, compare Figure 7.12b and Figure 7.12c), indicating that PSNR does not always capture visual quality. When particles are intended to be viewed as surfaces, my alternative scoring function often produces better visualizations, because nodes containing surface particles are given higher priority, even though they tend to be more sparse.

7.4.2 Traversals of hybrid and block-hybrid trees

In Figure 7.11, I encode six datasets with different characteristics (rows) and decode them using five combinations of tree and traversal orders (columns) discussed in the chapter. For each row, all columns are decoded at the same bit rate, but note that the number of decoded particles can be different for each method. It can be seen that DT on the hybrid tree is able to recover coarse reconstructions of the whole space instead of very fine reconstructions of only parts of the data, as is the case with DT on a k-d tree (see the second green-highlighted column pair). Compared to BT on k-d tree (DG), DT on hyrid tree tend to produce better results for dense surface data (*girl*, *soldier*), and worse results for sparse scientific data. A specially difficult case for DT on hybrid tree is *molecule*, where the distribution is very sparse but the particles are specified with high precision. In such cases, refining a coarse subset of particles to high precision is not useful (see Figure 7.11 (c5)).

For most datasets, BAT on block-hybrid tree often improves upon DT on hybrid tree visually by distributing error more uniformly throughout space. This observation is most visible when comparing (a5) with (a6), (c5) with (c6), and (e5) with (e6). Interestingly, in terms of PSNR, BAT on block-hybrid tree tends to perform worse than BT or AT on k-d trees and sometimes even DT on hybrid trees. Visually, however, BAT typically outperforms all other methods (most strikingly in the case of *molecule*), often producing a less blocky look on densely sampled surfaces compared to BT or AT (see *girl* or *soldier*). BAT can also fail visually (*cosmic web*) compared to DT on hybrid tree (see (f5) and (f6)) since when dense regions are clearly preferred, uniform refinement is not a good strategy. Finally, my hybrid and block-hybrid trees often generate significantly fewer particles at the same bit rates compared to BT on k-d trees (see *fissure*, *dam break*, and *cosmic web*), which should benefit downstream processing tasks. The most striking example can be seen by comparing (e6) with (e2), where BAT produces an almost identical-looking approximation to BT using only one-eighth the number of particles.

7.4.3 Speed and memory footprint

Figure 7.13 (a, b) shows that DT on any tree and BAT on block-hybrid tree achieve a constant memory footprint, whereas AT and BT require orders of magnitude more memory. Compared to DT and BAT, BT and AT also become slower very quickly. Compared to BT, my AT requires the same memory footprint and is slower, but can improve reconstruction quality by a good margin (as discussed in Subsection 7.4.1). The decode time for BAT grows faster than that of DT (on both k-d and hybrid trees), and its memory footprint is also higher, while still being asymptotically constant (Figure 7.13, middle). The trade-off is higher reconstruction quality (Figure 7.11). Notwithstanding its lack of features compared to BAT on block-hybrid tree, perhaps the best trade-off is had with DT on hybrid tree, which vastly improves reconstruction quality over DT on k-d tree almost for free. Based on these results, I recommend AT on k-d trees for small data and BAT on block-hybrid trees for large data, with AT limited to only the coarse k-d portion at the top.

I test the scalability of BAT on block-hybrid tree against the state-of-the-art octree compressor, MPEG [257], using the TMC3 [2] reference implementation. I encode eight datasets in increasing numbers of particles, and record the encoding time and memory usage of both methods. Figure 7.14 shows that my block-based encoder is several times faster than MPEG's encoder and, at the same time, uses an order of magnitude less memory for the larger datasets. Furthermore, my method's time requirement and memory footprint grow at much slower rates. For decoding, a fair comparison is difficult to obtain since MPEG decodes and outputs one block at a time, whereas my method maintains all the states necessary for simultaneous progressive decoding of all blocks (important for cross-block bit allocation). Nevertheless, MPEG crashes while decoding the largest dataset in this experiment, which consists 400 M particles.

I also encode a dataset with almost one billion particles (*detonation-large*, with 968M particles) using the block-hybrid tree, and then progressively decode and render three approximations from that same encoding (Figure 1.6). Rendering is done with OSPRay [295] after a subset of particles of the original 968M particles is decoded in each case. Since OSPRay constructs its own acceleration data structure for rendering which inflates the memory requirement, without reducing the number of particles, the original dataset could not be rendered on this machine with 64 GB RAM (it was previously rendered using 3 TB of RAM [296]). With block-hybrid tree, high-quality reconstructions are possible at significantly lower particle counts, decoded progressively using a constant memory footprint (50 MB of RAM).

7.4.4 Binomial coding

In Figure 7.15, I plot rate-distortion curves for both truncated binary coding [202,284] and my binomial coding using BT on k-d trees. I use three real-world datasets with approximately uniform distributions of particles, and a synthetic dataset where the distribution is truly uniform. It can be seen that, at the same data quality, binomial coding consistently improves the compression ratio by a factor between 10 and 20 percent. Conversely, at the same compression ratio, binomial coding on average improves the PSNR by about 0.5 dB. Figure 7.16 visually demonstrates the difference in data quality between binomial and truncated binary coding, using the *fissure* dataset, which shows that even a seemingly small difference of 0.68 dB can translate to a significant visual difference. The most difficult dataset to compress is unsurprisingly the *random* one.

To further evaluate the coding efficiency of my implementation, I compress the synthetic dataset consisting of randomly generated particles, and compare the size of the compressed bitstream to a theoretically calculated code size. The theoretical code size is calculated by summing the theoretically smallest number of bits needed to encode n_1 under every k-d tree node $(\mathbf{G}, n)$, assuming n_1 is binomially distributed given n. Figure 7.17 shows that my binomial coding implementation achieves code sizes that are virtually the same as the theoretically calculated ones across all tree depths, meaning the average code size for each tree node is close to the entropy of the binomial distribution. The same figure also shows that this (normalized) entropy approaches 0.5 as n gets larger toward the root of the tree, and 1 as n approaches 1 toward the leafs, consistent with my analysis in Subsection 7.3.1. This result is encouraging for increasingly larger (and denser) datasets of the future, since progressive refinements will stop more toward the root, resulting in better compression for binomial coding, approaching a reduction ratio of 0.5 compared to truncated binary coding. In terms of performance, binomial coding runs about 1.5 times slower than truncated binary coding.

7.4.5 Odd-even context coding

To test the efficacy of the odd-even context coding method, I compress several datasets with two methods: truncated binary coding and context coding. For this comparison, I always use DT on hybrid trees because odd-even context coding is designed to work with this combination. On a hybrid tree, DT visits the resolution levels from coarse to fine; I therefore record the ratio between the two bitstreams as each resolution level is processed, and plot these ratios in Figure 7.18 (a ratio less than 1 means context coding is better). The figure shows that context coding improves on truncated binary coding in compression ratio for several datasets, compressing up to 40% better (for *dancer*), and in several cases up to 20% better at the last resolution level (lossless). Context coding also works better as the resolution gets finer, likely because sibling subtrees under an odd-even split are more correlated at finer resolution levels due to them being less spatially separated. On the other hand, as this distance increases toward coarser resolution levels, the correlation reduces, and thus compression suffers.

Figure 7.18 also shows that my context coding does not work as well for some datasets, but is never significantly worse than truncated binary coding. I distinguish two kinds of datasets: densely sampled surfaces (dashed lines) and sparse but precise particles in scientific simulations (solid lines). It can be seen that the proposed scheme works better for the surface datasets, where the particles form very distinct shapes and there are enough particles that the shapes are relatively well preserved by odd-even subsampling. Such datasets contain densely distributed particles, therefore they have significantly fewer in-cell refinement bits. Since such bits tend to be more random, datasets with fewer of them often compress better. In contrast, the scientific simulation datasets are more difficult to compress because they are dominated by in-cell refinement bits, due to the particles being relatively sparse but stored with high precision.

There is one nonsurface dataset (*detonation*) for which my context-based scheme also works well. Here, the particles mostly follow a very regular arrangement as they represent arrays of explosives, and context modeling can exploit such global repetitions. Using the *dancer* dataset, Figure 7.19 demonstrates that context coding can result in significant improvements in PSNR over truncated binary coding for progressive decompression. Visually, the improvements in PSNR translate to better reconstructed surface at low bit rates with significantly fewer artifacts (Figure 7.20). In experiments, my implementation of odd-even context coding is often two times slower (for the encoder), and between three to eight times slower (for the decoder) than truncated binary coding. The extra cost mostly comes from the re-partitioning of the particles in the even subtree, which (as expected) doubles the computational cost for the encoder. Without odd-even context coding, the decoder is about four times faster than the encoder since it does not have to partition an array of particles to obtain node values (instead, node values are decoded from the bit stream). With odd-even context coding, which adds an extra partitioning step, both the encoder and the decoder run at similar speeds.

7.4.6 (Near) lossless compression ratio

I compare near lossless compression ratios (lossless with respect to quantized particles) among four methods: DG [57], my proposed techniques, MPEG [257], and LASZip [127] for several datasets in Table 7.1. To achieve the best compression, I use k-d trees with binomial coding for *crystal, molecule, salt, fissure, detonation* and *random-80*, block-hybrid trees with odd-even context coding for *girl, dancer*, and *sodier*, and hybrid trees with truncated binary coding for the rest. For lossless compression of point clouds, LASZip is an industry standard, and MPEG represents the state-of-the-art in compression ratio. Table 7.1 shows that my methods mostly achieve comparable lossless compression ratios against that of DG, which means the use of the odd-even splits does not degrade compression (while achieving much better quality-memory trade-offs as previously shown). For dense surface datasets (*girl*, *dancer*, *soldier*), my odd-even context coding results in significantly better lossless compression ratios over DG. LASZip produces the worst compression ratios among all methods in most cases, while MPEG compresses the best with its sophisticated context modeling. Unsurprisingly, MPEG also performs the best for the dense surface datasets (*girl*, *dancer*, *soldier*), since it is designed specifically for this kind of data. For many of the coarse-but-precise scientific datasets (*molecule*, *salt*, *dam break*, *coal* and *cosmic web*), however, MPEG's compression ratios are no better than ours.

detonation contains highly regular, repeating particle arrangements, which MPEG and LASZip take advantage of, whereas DG and ours do not. However, with additional dictionary-based compression, my method's compression ratio increases from 3.85 to 10.3, comparable to that of LASZip's. *random-80* is a synthetically generated dataset where a random 80% of the grid cells contain particles. Since my grid-based approach scales gracefully from sparse to dense data by switching to coding empty cells when particles are densely distributed, it compresses twice better than DG and four times better than LASZip, whereas MPEG simply crashes. Most scientific datasets in practice are sparse relative to the grid size, but future data will likely become denser as more particles are captured and simulated.

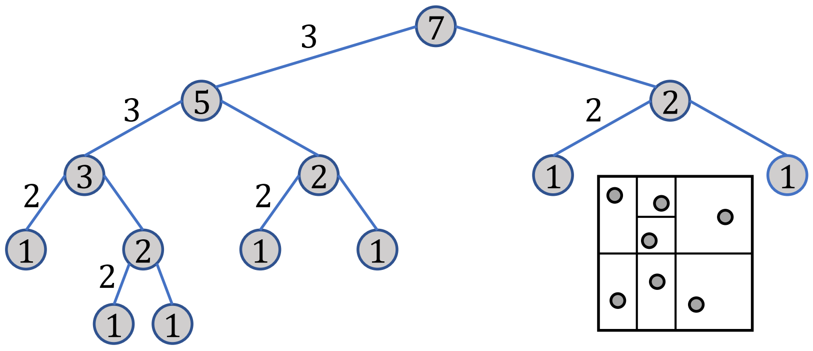


Figure 7.1: A k-d tree built for 7 particles in 2D (bottom right). For simplicity, the subdivision stops when the particles are all separated. Each node contains the number of particles in its bounding box. Numbers on the edges specify the number of bits required to encode the corresponding left children nodes (right children are inferred). The numbers written to the bit stream are (in BT order): 7, 5, 3, 1, 1, 1, 1, using a total of 14 bits.

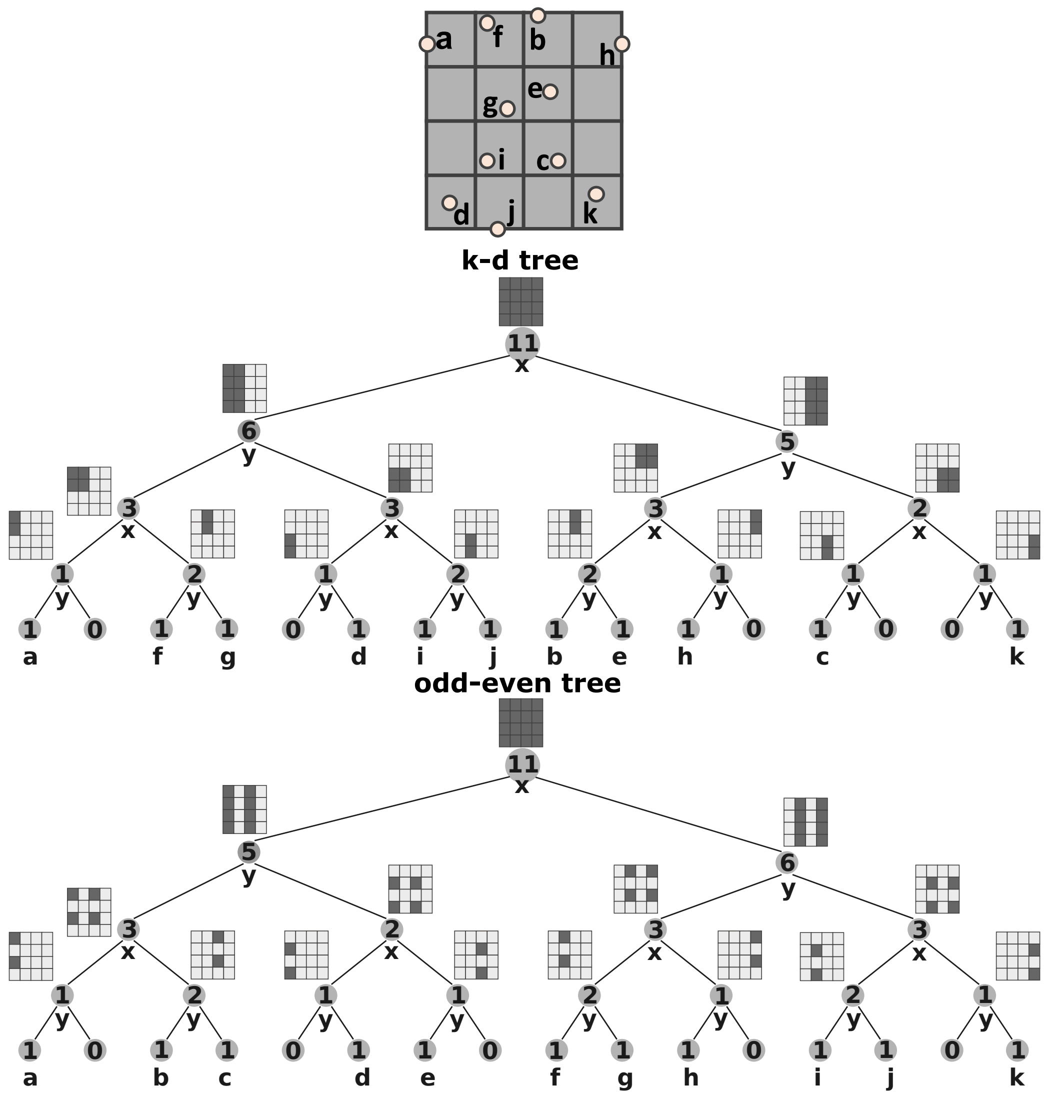
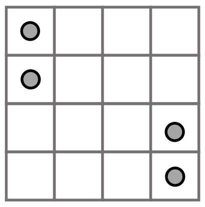
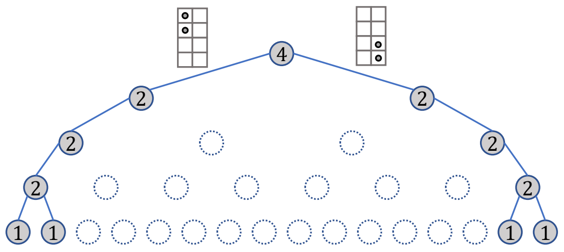
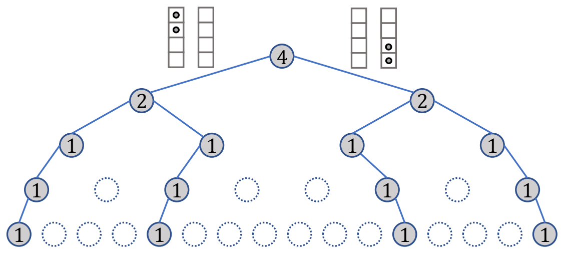


Figure 7.2: An example with 11 particles (a to k) on a 4^2 grid, from which I build a k-d tree (left) and an odd-even tree (right). The odd-even splits partition space by interleaving odd-indexed and even-indexed grid cells at each tree level. For simplicity, the trees are built only until every particle is located to its own cell. The numbers encoded in the bit stream (using DT) are $\{11, 6, 3, 1, 1(a), 1(f), 1, 0, 1(i), 3, 2, 1(b), 1(h), 1, 1(c), 0\}$ for the k-d tree, and $\{11, 5, 3, 1, 1(a), 1(b), 1, 0, 1(e), 3, 2, 1(f), 1(h), 2, 1(i), 0\}$ for the odd-even tree.


(a) 4 particles in 2D


(b) The particles encoded as a k-d tree. The total number of nodes is 11.


(c) The particles encoded as an odd-even tree. The total number of nodes is 15.

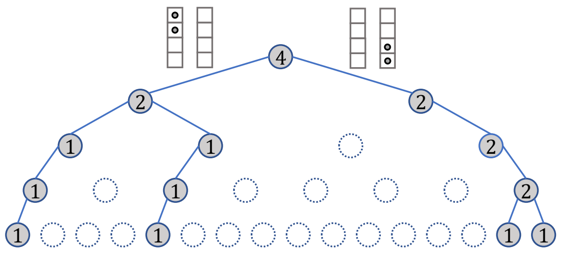

(d) The particles encoded as a hybrid tree. The total number of nodes is 13.

Figure 7.3: An example that demonstrates how odd-even trees (right) compress not as well as k-d trees (left), and how hybrid trees (middle) alleviate this problem. The odd-even tree uses odd-even splits exclusively, whereas the hybrid tree only uses odd-even splits on the path connecting the root to the left-most leaf node. Odd-even trees do not compress well since they create too many nodes to fully locate the particles among all cells.

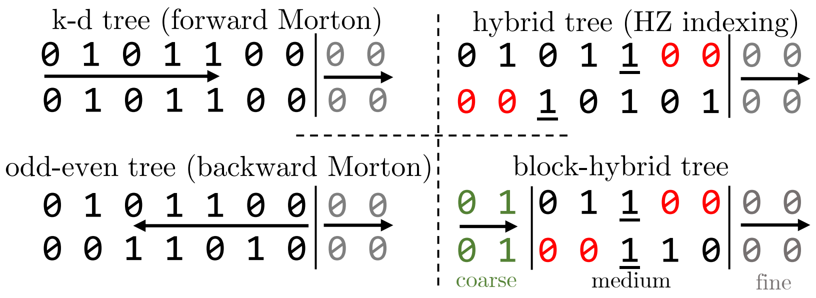
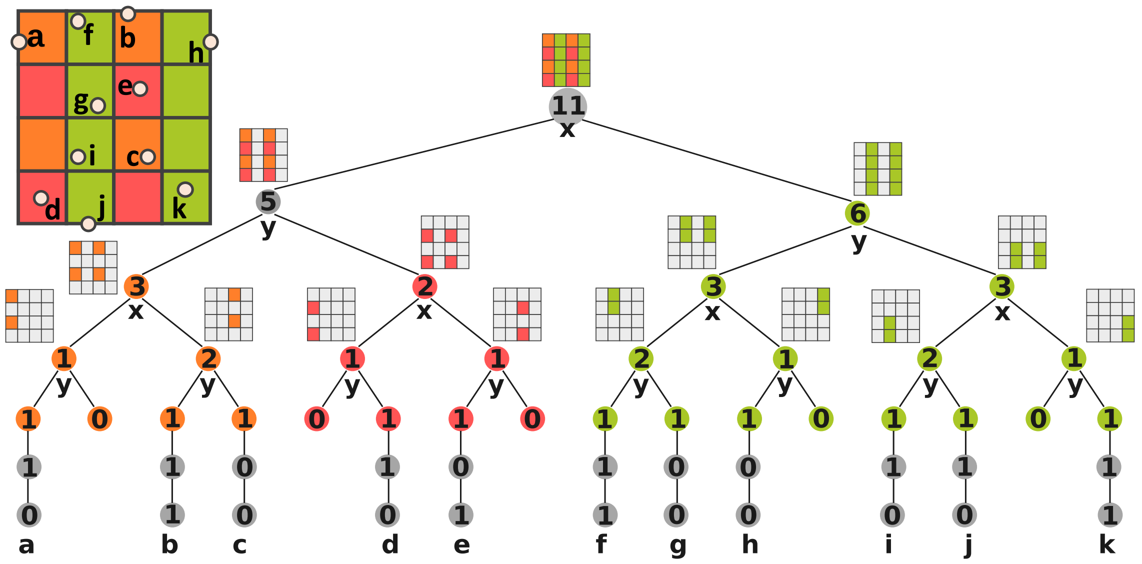
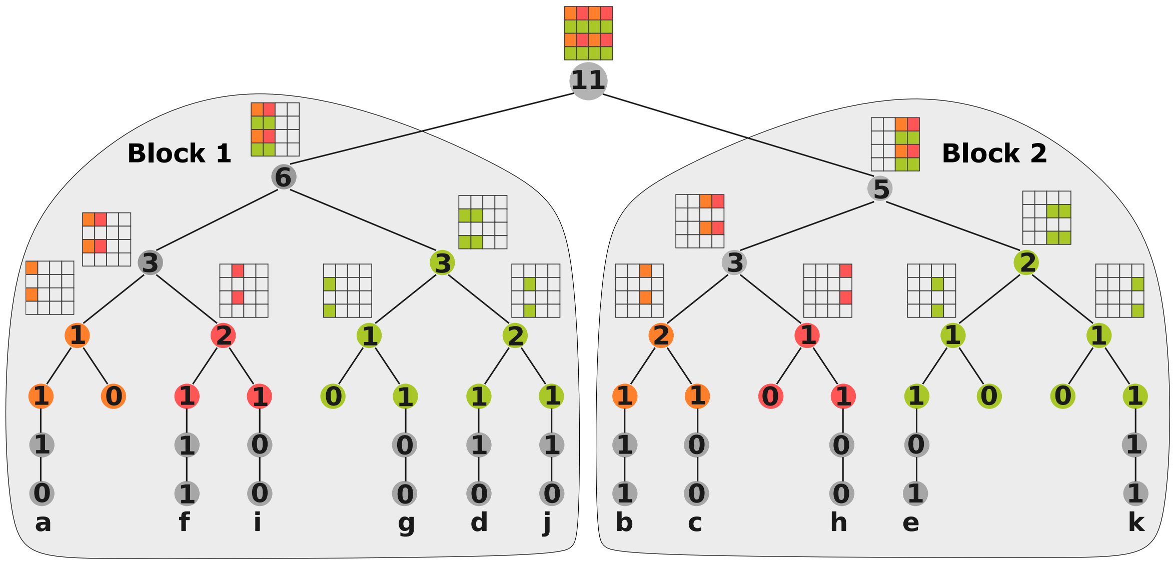


Figure 7.4: A tree implies an ordering of particles following their transformed Morton codes. Input Morton bits are shown the top and arrows indicate directions of the output bits at the bottom. K-d trees and odd-even trees use forward and backward Morton codes. Hybrid trees use HZ indexing [223]. Block-hybrid trees use HZ indexing for the medium portion.



(a) A hybrid tree with three resolution levels, created with two odd-even splits (at the root and its left child)



(b) A block-hybrid tree with two blocks, each of which is a hybrid tree with three resolution levels

Figure 7.5: Hybrid and block-hybrid trees. (a) A *hybrid tree* created using a particular combination of odd-even splits (with different colored child nodes) and the standard k-d splits (same colored child nodes). (b) A *block-hybrid tree* created by exclusive uses of k-d splits at shallow depths and hybrid trees further down. Both trees are constructed for the same 11-particle in Figure 7.2, with additional (conceptual) tree nodes for in-cell refinement bits, shown in gray.

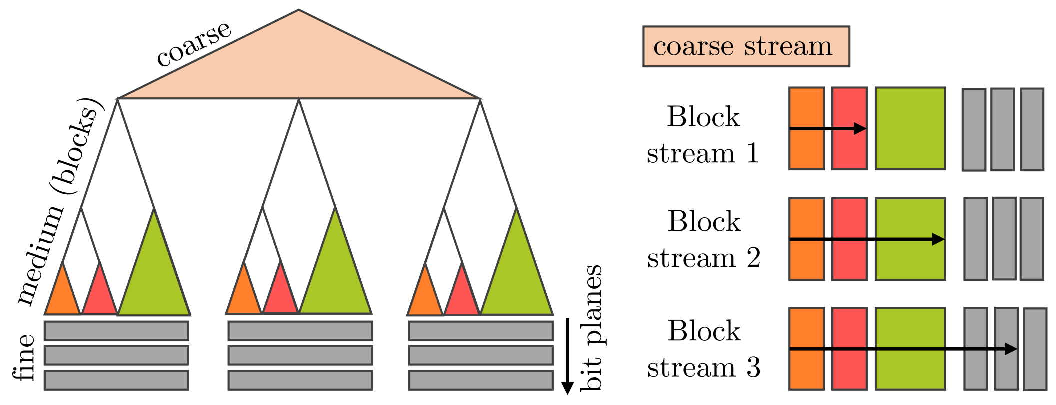


Figure 7.6: Schematic of a block-hybrid tree's bit streams. Left: a block-hybrid tree with subtrees colored by resolution level. Right: the blocks' bit streams are stored separately, so that blocks can be decoded independently, indicated by the arrows. In a block, medium-portion bits are in depth-first order (by resolution level), whereas in-cell refinement bits (gray) are in breadth-first order (by bit plane). At decoding time, any of the resolution levels (colored triangles) can be skipped in favor of more refinement bits for the coarser resolution levels.

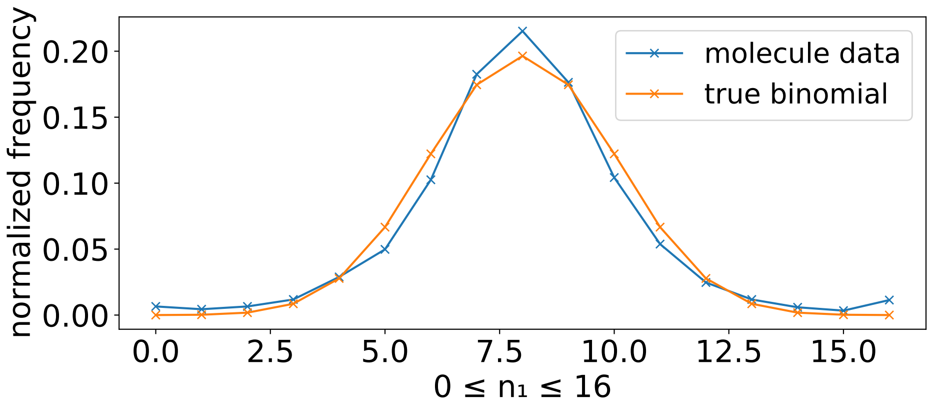


Figure 7.7: (Normalized) frequencies of n_1 (number of particles in the left child), given $n = 16$, for both the *molecule* dataset and a true binomial distribution *i.e.*, $B(16, \frac{1}{2})$. The empirical distribution tracks the theoretical distribution well, showing that n_1 is clearly not uniformly distributed.

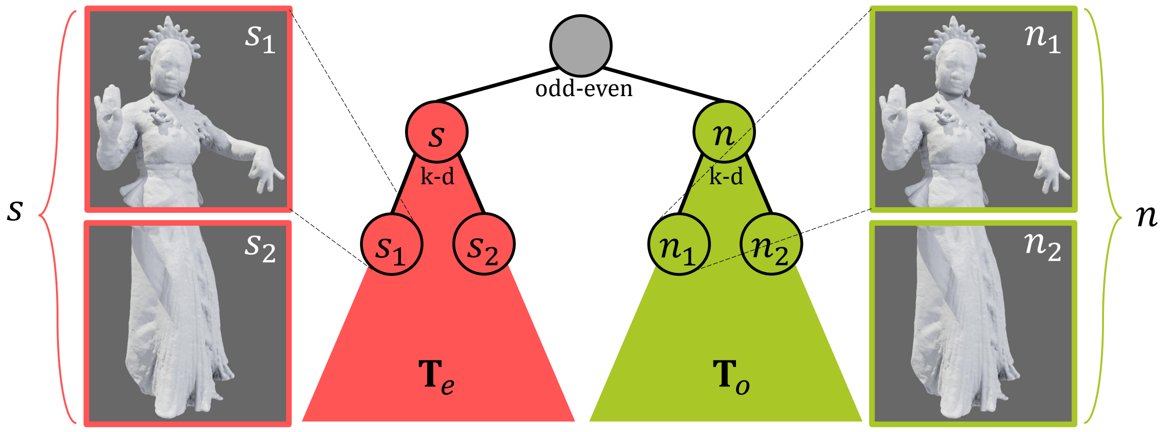


Figure 7.8: Similarity between the odd-even subtrees. The left (red) and right (green) subtrees under an odd-even split can have similar particle distributions, and one can be used to predict the other. Here, n_1 can be inferred from n, s, and s_1 (*e.g.*, $n_1 \approx ns_1/s$).

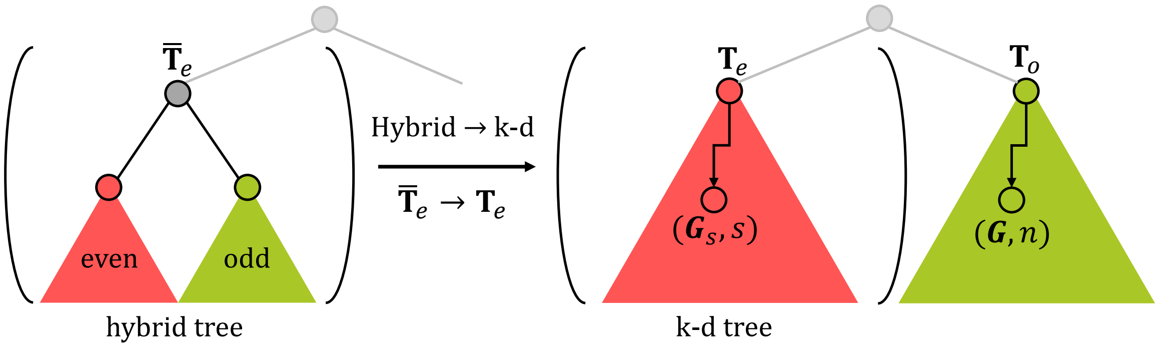


Figure 7.9: The even subtree $\bar{\mathbf{T}}_e$ is transformed from a hybrid tree to a k-d tree $\mathbf{T}_e$. The odd subtree $\mathbf{T}_o$ is coded using a lockstep traversal with $\mathbf{T}_e$. Local information at the two front nodes ($(\mathbf{G}_s, s)$ and $(\mathbf{G}, n)$) are used for context coding. When $\mathbf{T}_o$ is fully coded, it is combined with $\mathbf{T}_e$ and transformed into a k-d tree for next-level prediction.

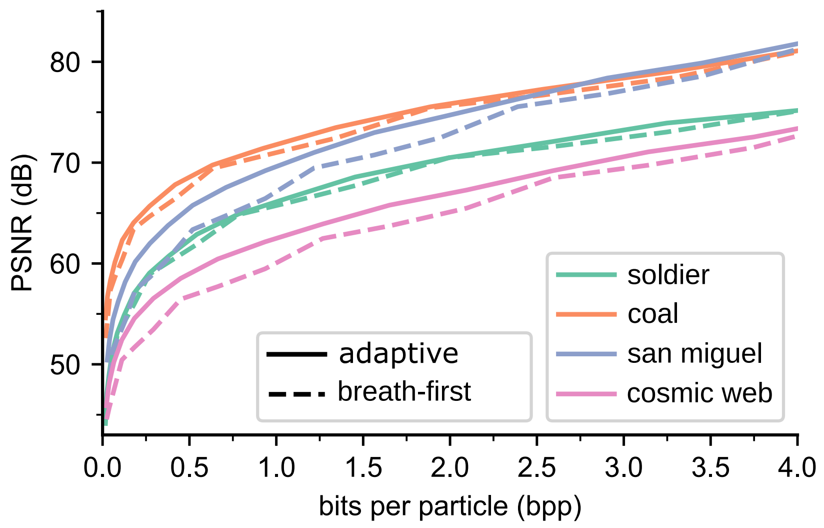


Figure 7.10: Rate-distortion curves for AT and BT on k-d trees. AT not only outperforms BT on all datasets tested, but also produces significantly "smoother" rate-distortion curves.

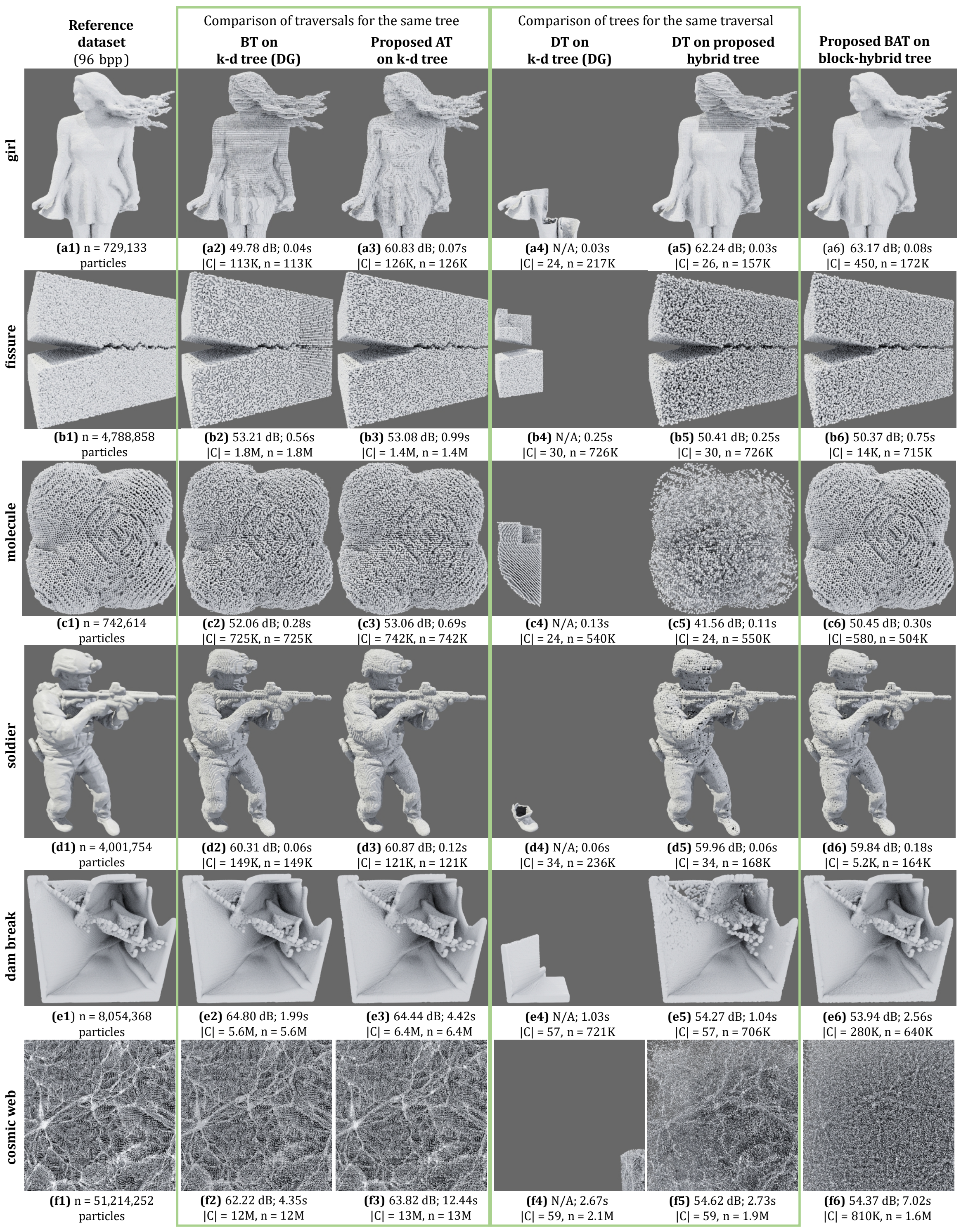

Reference dataset (96 bpp)
Comparison of traversals for the same tree
BT on k-d tree (DG)
Proposed AT on k-d tree
Comparison of trees for the same traversal
DT on k-d tree (DG)
DT on proposed hybrid tree
Proposed BAT on block-hybrid tree
girl
(a1) n = 729,133 particles
(a2) 49.78 dB; 0.04s |C| = 113K, n = 113K
(a3) 60.83 dB; 0.07s |C| = 126K, n = 126K
(a4) N/A; 0.03s |C| = 24, n = 217K
(a5) 62.24 dB; 0.03s |C| = 26, n = 157K
(a6) 63.17 dB; 0.08s |C| = 450, n = 172K
fissure
(b1) n = 4,788,858 particles
(b2) 53.21 dB; 0.56s |C| = 1.8M, n = 1.8M
(b3) 53.08 dB; 0.99s |C| = 1.4M, n = 1.4M
(b4) N/A; 0.25s |C| = 30, n = 726K
(b5) 50.41 dB; 0.25s |C| = 30, n = 726K
(b6) 50.37 dB; 0.75s |C| = 14K, n = 715K
molecule
(c1) n = 742,614 particles
(c2) 52.06 dB; 0.28s |C| = 725K, n = 725K
(c3) 53.06 dB; 0.69s |C| = 742K, n = 742K
(c4) N/A; 0.13s |C| = 24, n = 540K
(c5) 41.56 dB; 0.11s |C| = 24, n = 550K
(c6) 50.45 dB; 0.30s |C| =580, n = 504K
soldier
(d1) n = 4,001,754 particles
(d2) 60.31 dB; 0.06s |C| = 149K, n = 149K
(d3) 60.87 dB; 0.12s |C| = 121K, n = 121K
(d4) N/A; 0.06s |C| = 34, n = 236K
(d5) 59.96 dB; 0.06s |C| = 34, n = 168K
(d6) 59.84 dB; 0.18s |C| = 5.2K, n = 164K
dam break
(e1) n = 8,054,368 particles
(e2) 64.80 dB; 1.99s |C| = 5.6M, n = 5.6M
(e3) 64.44 dB; 4.42s |C| = 6.4M, n = 6.4M
(e4) N/A; 1.03s |C| = 57, n = 721K
(e5) 54.27 dB; 1.04s |C| = 57, n = 706K
(e6) 53.94 dB; 2.56s |C| = 280K, n = 640K
cosmic web
(f1) n = 51,214,252 particles
(f2) 62.22 dB; 4.35s |C| = 12M, n = 12M
(f3) 63.82 dB; 12.44s |C| = 13M, n = 13M
(f4) N/A; 2.67s |C| = 59, n = 2.1M
(f5) 54.62 dB; 2.73s |C| = 59, n = 1.9M
(f6) 54.37 dB; 7.02s |C| = 810K, n = 1.6M

Figure 7.11: Visual comparison of the different traversal-tree combinations (columns) discussed in this chapter for six datasets (rows). The reduced datasets are shown at 1.1 bpp (*girl*), 1.3 bpp (*fissure*), 4.4 bpp (*molecule*), 0.4 bpp (*soldier*), 3.1 bpp (*dam break*), and 1.3 bpp (*cosmic web*).

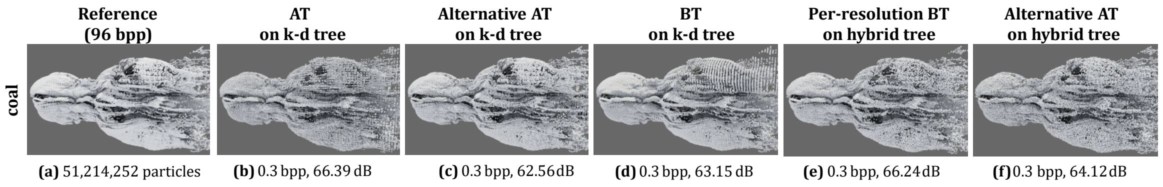
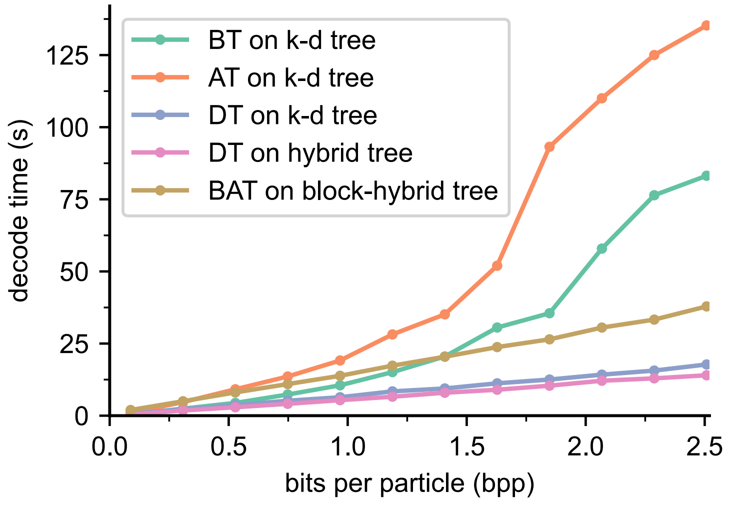
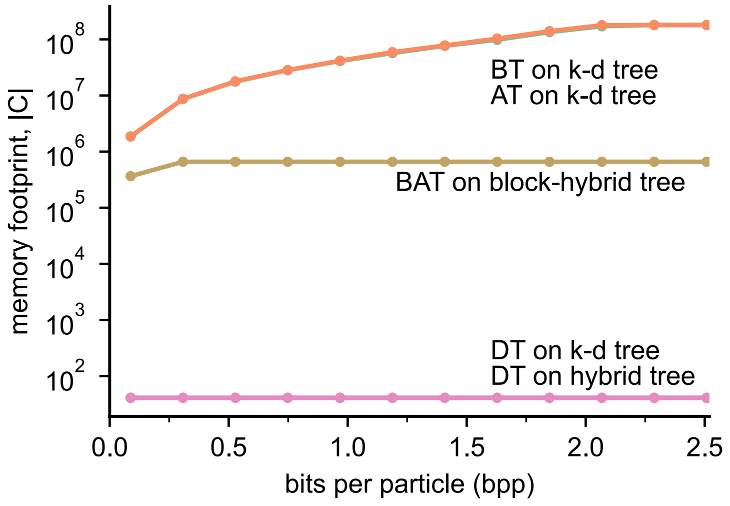


Figure 7.12: Reconstruction results for alternative combinations of traversal orders and trees, including the use of an alternative scoring function for AT to obtain a better reconstruction visually (c), even at a lower PSNR. All reconstructions are at 0.3 bpp. Although not canonical, BT and AT on hybrid trees are very possible combinations, which may sometimes be preferable than BT on k-d trees, as is perhaps the case here.

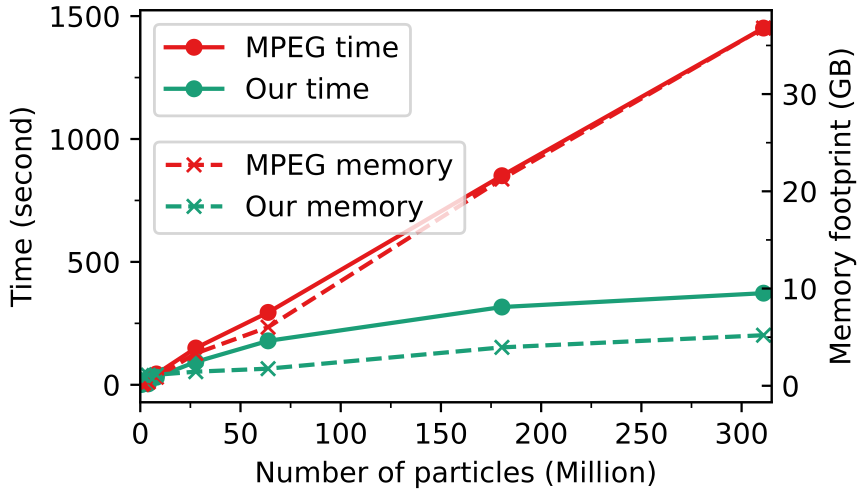


(a) Decode times



(b) Memory footprints

Figure 7.13: Decode times and memory footprints for combinations of trees and traversal methods, plotted for the *detonation* dataset. DT and BAT achieve constant memory footprint and linearly scaled decode time in number of bits, whereas AT and BT require orders of magnitude more memory, and also much faster growing decode time.



(a) Comparison with MPEG

Figure 7.14: Compared to MPEG [257], my block-based encoder (BAT on block-hybrid tree) is almost 5× to 7× less expensive, and my method's time and memory costs also grow at much slower rates.

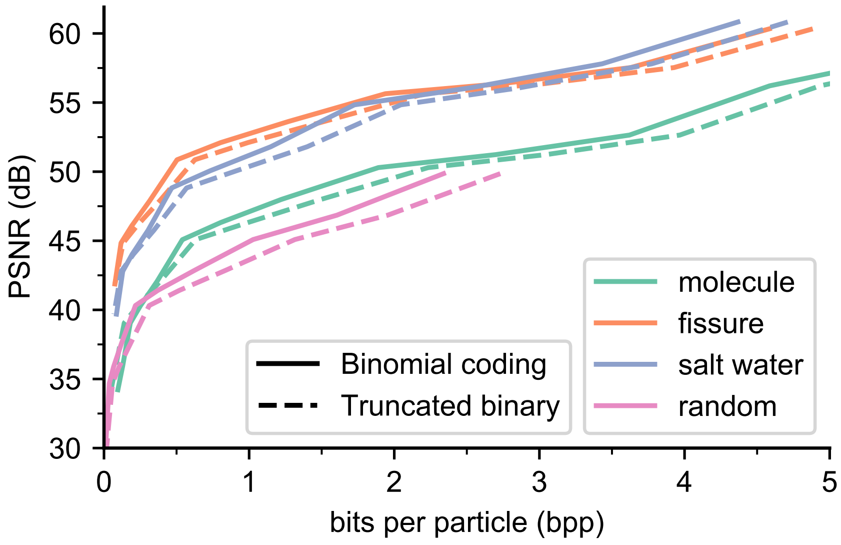
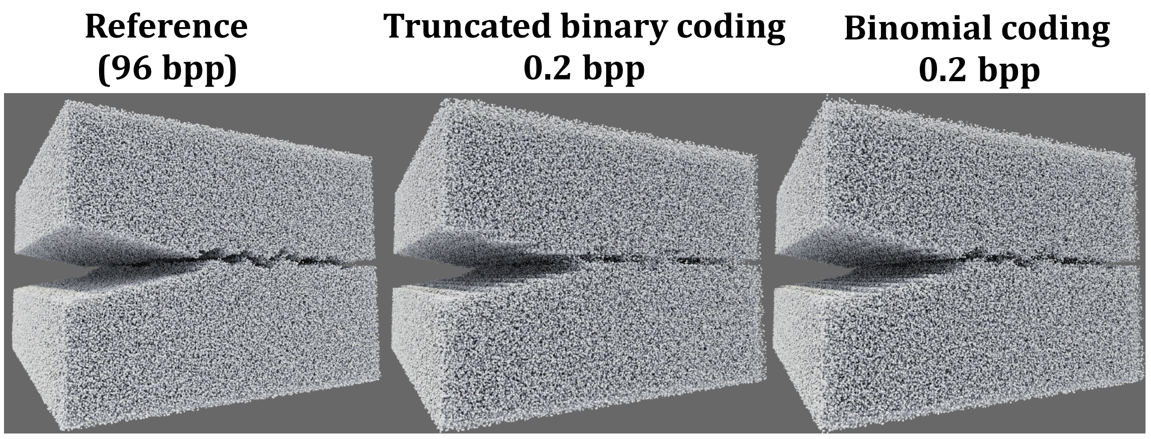


Figure 7.15: Rate-distortion curves demonstrate that my proposed binomial encoding outperforms the standard truncated-binary coding [284] for datasets with approximately uniform distributions of particles. I also include a synthetic *random* dataset, with random particle distribution.



(a) 8,054,368 particles **(b)** 45.78 dB **(c)** 46.46 dB

Figure 7.16: At the same bit rates, binomial coding more faithfully reconstructs features in the original data: for the *fissure* dataset, the shape of the crack is more clearly defined with binomial coding.

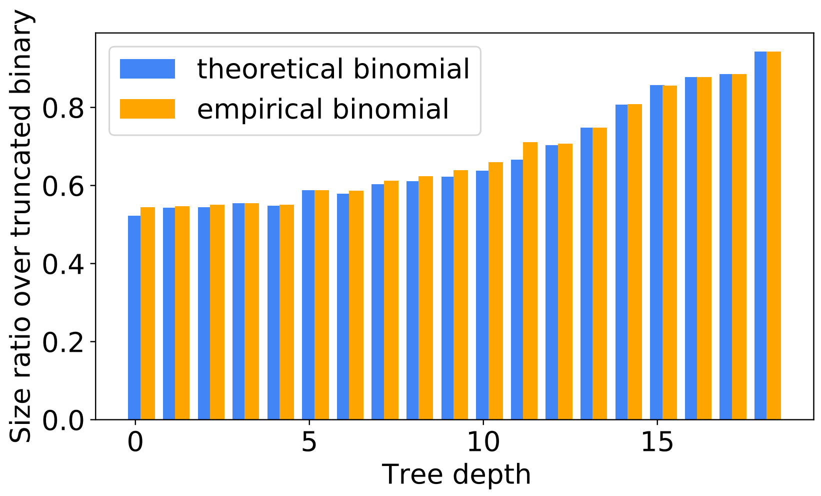


Figure 7.17: Ratios of code sizes (binomial coding over truncated binary coding), both theoretically calculated and empirically measured, for a synthetic dataset with randomly generated particles. My binomial coding implementation achieves almost perfect coding efficiency.

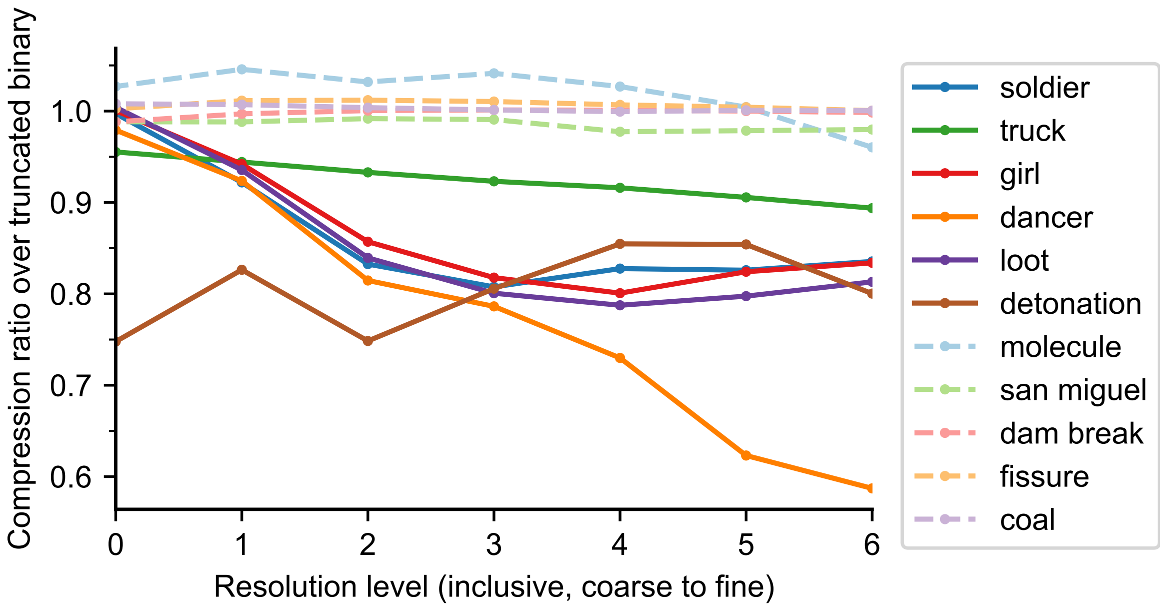


Figure 7.18: Ratio between the compressed code sizes (context coding over truncated binary coding) at progressively finer resolution levels, with the last level corresponding to lossless compression. My context coder works very well for dense surface datasets (solid lines), and less well for high-precision but sparse datasets (dashed lines).

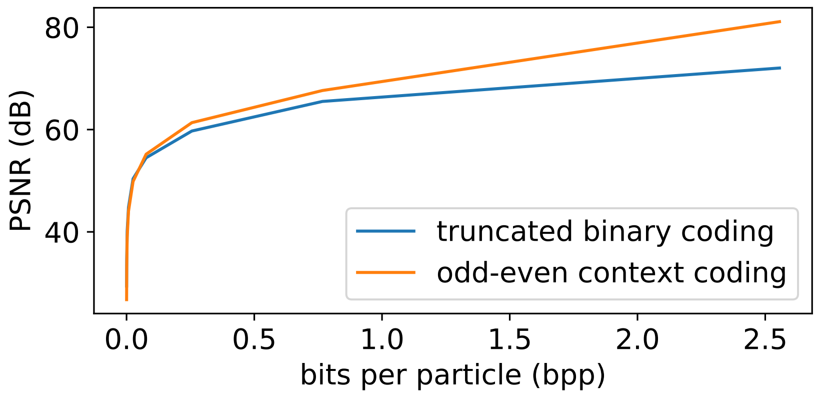


Figure 7.19: Rate-distortion plots for the *dancer* dataset shows that odd-even context coding significantly outperforms truncated binary coding.

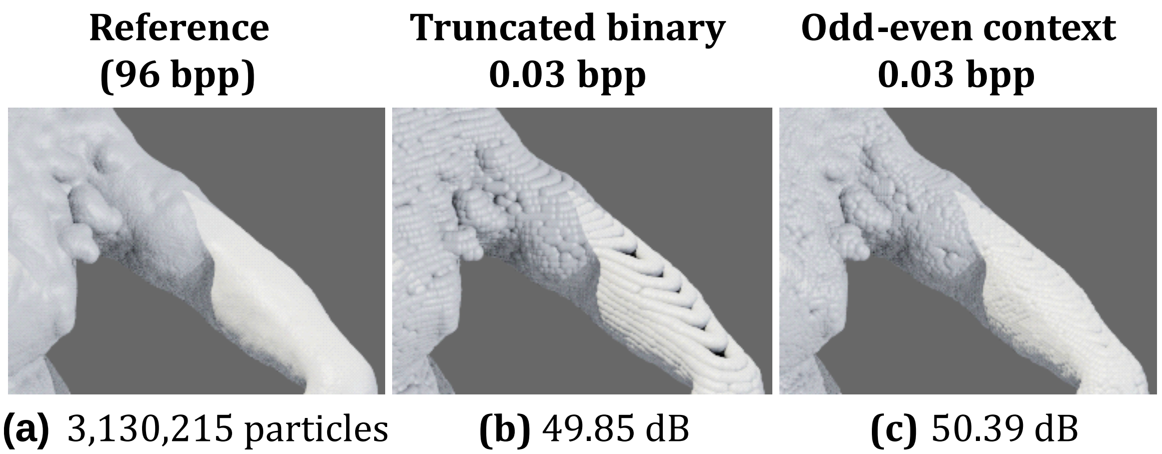


Figure 7.20: Visual comparison between the two coding methods at the same bit rate of 0.03 bpp. Odd-even context coding reproduces the reference data more faithfully (with fewer artifacts).

Table 7.1: Comparison of lossless compression ratios across four compression methods for the several datasets used in my experiments. I give further comments in the text for the datasets marked with *.

datasets	# particles	DG	Ours	MPEG	LASZip
crystal [198]	16K	2.10	2.11	2.44	1.56
girl [142]	729K	13.90	39.57	67.29	9.92
molecule [296]	742K	2.19	2.20	2.18	1.64
salt [5]	1.8M	2.36	2.38	2.33	1.51
fissure [296]	4.7M	2.54	2.56	3.50	2.13
dancer [142]	3.1M	22.81	35.42	65.02	6.89
soldier [142]	4M	13.70	16.14	21.00	5.56
san miguel [192]	3.6M	3.36	3.20	3.46	2.02
dam break [265]	8M	2.71	2.69	2.69	1.85
coal [196]	27M	3.45	3.42	3.36	2.20
cosmic web [296]	51M	3.17	3.12	3.06	1.72
*detonation [21]	180M	3.08	3.85	24.20	10.30
*random-80	1.6M	42.70	95.50	crashed	27.40

CHAPTER 8

CONCLUSION

I modeled the full space of data reductions with a novel tree T_r^p that unifies precision and resolution, and accepts approximations as sub-trees produced by valid cuts of T_r^p. The parameters and trade-offs needed for practical data layouts encoding the tree are provided. In addition, I provided an empirical study on the system-level considerations leading to an efficient design, which I showed to be competitive with the state of the art. My T_r^p captures the essential dependencies among nodes, but not all dependencies. For example, with filters such as wavelets, there exist data dependencies among neighboring nodes, required for inverse filtering (I handled this in my implementation). In addition, how cuts can be economically represented in memory is an open question. For future work, I would like to determine if and how optimal mixtures of reductions can be automatically found for scientific tasks with different notions of errors and costs. Extending the current framework to different grid types, unstructured data, and time-varying data is yet another direction. Finally, studying how applications such as interactive GPU rendering can benefit from a data layout such as the one proposed is important.

For progressive streaming, I focus on the trade-off between two prominent dimensions of data reduction, resolution and precision, with respect to common analysis and visualization tasks. To keep the study tractable while not compromising the generalizability of results, I target fundamental analysis and visualization tasks, with an outlook that these can serve as building blocks for more complex and multiparameter tasks in the future. Although the work focuses on a small set of core tasks, the framework is generic and applies to any well-defined metric, and one future direction is to consider a broader set of tasks.

I presented the first empirical study to demonstrate that combining reduction in precision and resolution can lead to a significant improvement in data quality for the same data budget, and that different tasks might prefer different resolution-versus-precision trade-offs.

For example, whereas computing histograms requires high precision, computing derivatives benefits more from higher resolution, and function reconstruction and isosurface extraction require a suitable mix of the two (see Figure 6.14). I also showed that common reduction techniques, *e.g.,* those based on $\mathcal{S}_{\text{lvl}}$ and $\mathcal{S}_{\text{mag}}$, do not perform well when leading zero bits are removed (to simulate entropy compression). For each task, the relative ordering of the rate-distortion curves stays largely the same regardless of data sets, although the gaps among them vary depending on the smoothness and noisiness of the data. Compared to data-independent streams, signature-based streams often perform better because they can adapt to the data. They are also amenable for implementation (unlike $\mathcal{S}_{\text{opt}}$), since a signature is negligibly small and thus can be precomputed and stored during preprocessing. It is also interesting to consider per-block signatures instead of a global one.

An important question is whether task- and data-dependent streams provide sufficient advantages over purely data-independent streams. In practice, data would be used for multiple, and not necessarily predefined, tasks, and maintaining multiple streams will likely lead to additional overheads. Here, I consider $\mathcal{S}_{\text{sig}}$ to be the best possible stream that could be realized. Improvements on $\mathcal{S}_{\text{sig}}$ in the resolution-versus-precision space are likely possible, but they are unlikely to be significant. Given these assumptions and the fact that $\mathcal{S}_{\text{sig}}$ in most cases provides very similar results to $\mathcal{S}_{\text{bit}}$ or $\mathcal{S}_{\text{wav}}$, the additional effort (and potential overheads) for task-dependent bit orderings is unlikely to be beneficial. This leaves a significant gap between the best data-independent streams and the optimal stream $\mathcal{S}_{\text{opt}}$. My experiments suggest that the majority of these differences can be attributed to spatial adaptivity (see Figure 6.14). The prototypical example is isosurface computation, where $\mathcal{S}_{\text{opt}}$ can skip all regions that do not affect any portion of the surface. It may be worthwhile in future work to investigate solutions to spatial adaptivity to significantly improve the performance of data-independent streams.

This study can be considered only a first step toward a system of solutions that can optimize storage, network, and I/O bandwidth to suit specific tasks at hand. Ultimately, my results can guide development of new data layouts and file formats for scientific data.

For fair comparisons, I always reconstruct the data at full resolution using wavelets. However, processing and memory costs are important, and it is likely that adaptive representations would be used in practice [49, 93, 205]. In these cases the error of a given approximation depends not only on the available information, but also on the data structure and algorithm being used. For example, trilinear interpolation on a coarse grid might produce different results than wavelet reconstruction on the original mesh. There exist solutions where both interpolations are equivalent [305], but such solutions have not yet been implemented in standard tools. An important future research direction will be to understand the implications of the results presented here for existing toolchains such as VTK.

For particle data, I have presented novel techniques along a tree-based particle compression pipeline, centered around the concept of an *odd-even split*. The presented novel tree construction and traversal techniques achieve a better balance between data quality and resource requirements compared to other state-of-the-art particle compressors. My *adaptive traversal* approach improves over the static breadth-first traversal with respect to a user-defined error heuristic. Compared to k-d trees, my *hybrid trees* enable high-quality depth-first traversal. The *block-hybrid tree* allows not only independent, low-footprint encoding and decoding of blocks, but also higher reconstruction quality compared to all other approaches. The proposed *block-adaptive traversal* approach allows flexible, error-guided reconstructions at decoding time independent of how data is compressed. *binomial coding* and *odd-even context coding* significantly improve the compression ratio for datasets they are designed for by as much as 20% (for uniform distributions) and 40% (for densely sampled surfaces). All of the proposed techniques benefit the encoder and decoder equally. Working together, the contributions amount to a highly flexible and scalable particle compression system, which compares favorably to the state-of-the-art MPEG standard in memory and speed, both in absolute terms and in rates of growth.

Like DG [57], my method does not take advantage of global redundancy, which could be useful to compress certain regular arrangements of particles, albeit at the expense of coding complexity and speed. To realize the odd-even splitting scheme, I need to quantize particle positions to avoid the inaccuracy caused by floating-point operations, but techniques may exist that maintain accuracy without quantization. I also do not tackle compression of attributes other than positions, although odd-even splitting – being based on the lazy wavelet transform – might suggest a wavelet-based compression scheme for attributes. There are opportunities for more in-depth studies of the trade-offs between odd-even and k-d splits, as well as between various possible combinations of tree and traversal types. The idea of odd-even splits may be generalized to octrees, although perhaps with different trade-offs.

For tasks such as such as nearest-neighbor queries, occlusion culling, or empty-space skipping in rendering, it remains to be seen how the odd-even splitting mechanism affects application-level concerns, and to what extent my hybrid and block-hybrid trees can be used for noncompression purposes. For some datasets, neither binomial coding nor odd-even context coding may be applicable. Such datasets tend to contain nonuniform, relatively sparse but precise particles, which are common in scientific simulations. Better coding schemes might be invented to better target these cases, for which I hope the ideas presented here provide good starting points. Finally, it is also important to study task-oriented error metrics/heuristics and their utility to drive either tree construction or tree traversal, or both.

REFERENCES

[1] "Draco: Geometric coding for dynamic voxelized point clouds," https://github.com/google/draco, accessed: 2021-03-31.

[2] "MPEG-PCC-TMC13: Geometrybased point cloud compression," https://github.com/MPEGGroup/mpeg-pcc-tmc13, accessed: 2021-03-31.

[3] "OpenJPEG: An open-source JPEG 2000 codec written in C," http://www.openjpeg.org. [Online]. Available: http://www.openjpeg.org/

[4] "OpenTopography," https://portal.opentopography.org/datasets, accessed: 2021-03-31.

[5] "Scientific visualization contest 2016," https://www.uni-kl.de/sciviscontest/, accessed: 2021-03-31.

[6] M. Aharon, M. Elad, and A. Bruckstein, "K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation," *IEEE Transactions on Signal Processing*, vol. 54, no. 11, pp. 4311–4322, 2006.

[7] N. Ahmed and K. R. Rao, "Walsh-hadamard transform," in *Orthogonal Transforms for Digital Signal Processing*. Springer, 1975, pp. 99–152.

[8] J. Alakuijala, R. Van Asseldonk, S. Boukortt, M. Bruse, I.-M. Comșa, M. Firsching, T. Fischbacher, E. Kliuchnikov, S. Gomez, R. Obryk *et al.*, "JPEG XL next-generation image compression architecture and coding tools," in *Applications of Digital Image Processing XLII*, vol. 11137, 2019, pp. 112–124.

[9] E. Alexiou and T. Ebrahimi, "On the performance of metrics to predict quality in point cloud representations," in *Applications of Digital Image Processing XL*, vol. 10396, 2017, p. 103961H.

[10] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A massively parallel amr code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[11] M. Ament, D. Weiskopf, and H. Carr, "Direct interval volume visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1505–1514, 2010.

[12] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola, "TTHRESH: Tensor compression for multidimensional visual data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 9, pp. 2891–2903, 2019.

[13] R. Ballester-Ripoll and R. Pajarola, "Lossy volume compression using Tucker truncation and thresholding," *The Visual Computer*, vol. 32, no. 11, pp. 1433–1446, 2016.

[14] R. Ballester-Ripoll, S. Suter, and R. Pajarola, "Analysis of tensor approximation for compression-domain volume visualization," *Computers & Graphics*, vol. 47, pp. 34–47, 2015.

[15] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, 1970, pp. 107–141.

[16] S. Belkasim, "Multi-resolution analysis using symmetrized odd and even DCT transforms," in *2011 Data Compression Conference*, 2011, pp. 447–447.

[17] G. Bell, T. Hey, and A. Szalay, "Beyond the data deluge," *Science*, vol. 323, no. 5919, pp. 1297–1298, 2009.

[18] W. R. Bennett, "Spectra of quantized signals," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 446–472, 1948.

[19] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[20] M. Bertram, M. A. Duchaineau, B. Hamann, and K. I. Joy, "Generalized B-spline subdivision-surface wavelets for geometry compression," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 3, pp. 326–338, 2004.

[21] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight, "Extending the UINTAH framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S101–S122, 2016.

[22] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-art in GPU-based large-scale volume visualization," in *Computer Graphics Forum*, vol. 34, no. 8, 2015, pp. 13–37.

[23] G. Beylkin, R. Coifman, and V. Rokhlin, "Fast wavelet transforms and numerical algorithms," in *Fundamental Papers in Wavelet Theory*. Princeton University Press, 2009, pp. 741–783.

[24] A. Bhagatwala, Z. Luo, H. Shen, J. A. Sutton, T. Lu, and J. H. Chen, "Numerical and experimental investigation of turbulent DME jet flames," *Proceedings of the Combustion Institute*, vol. 35, no. 2, pp. 1157–1166, 2015.

[25] H. Bhatia, D. Hoang, N. Morrical, V. Pascucci, P.-T. Bremer, and P. Lindstrom, "AMM: Adaptive multilinear meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 6, pp. 2350–2363, 2022.

[26] A. Bhattacharyya, "On a measure of divergence between two statistical populations defined by their probability distribution," *Bulletin of the Calcutta Mathematical Society*, vol. 35, pp. 99–110, 1943.

[27] M. Botsch, A. Wiratanaya, and L. Kobbelt, "Efficient high quality rendering of point sampled geometry," in *Eurographics Workshop on Rendering (EGRW)*, 2002, pp. 53–64.

[28] T. Boutell, "PNG (portable network graphics) specification version 1.0," Tech. Rep., 1997.

[29] P. J. Burt and E. H. Adelson, "The Laplacian pyramid as a compact image code," in *Readings in Computer Vision*. Elsevier, 1987, pp. 671–679.

[30] S. Byna, A. Uselton, D. Knaak, and H. He, "Trillion particles, 120,000 cores, and 350 TBs: Lessons learned from a hero I/O run on Hopper," in *Cray User Group conference (CUG)*, 2013.

[31] K. Cai, Y. Liu, W. Wang, H. Sun, and E. Wu, "Progressive out-of-core compression based on multi-level adaptive octree," in *ACM Virtual Reality Continuum and Its Applications in Industry (VRCIA)*, 2006, pp. 83–89.

[32] J. Calhoun, F. Cappello, L. N. Olson, M. Snir, and W. D. Gropp, "Exploring the feasibility of lossy compression for pde simulations," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 397–410, 2019.

[33] P. Chebyshev, "Sur l'interpolation," *Oeuvres de P.L. Chebyshev*, vol. 1, p. 541, 1961.

[34] S. Chen, D. Tian, C. Feng, A. Vetro, and J. Kovačević, "Fast resampling of three-dimensional point clouds via graphs," *IEEE Transactions on Signal Processing*, vol. 66, no. 3, pp. 666–681, 2018.

[35] Y. Chen, D. Murherjee, J. Han, A. Grange, Y. Xu, Z. Liu, S. Parker, C. Chen, H. Su, U. Joshi *et al.*, "An overview of core coding tools in the AV1 video codec," in *Picture Coding Symposium (PCS)*, 2018, pp. 41–45.

[36] Y. Cho and W. Pearlman, "Quantifying the coding performance of zerotrees of wavelet coefficients: Degree-*k* zerotree," *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 2425–2431, 2007.

[37] Y. Cho and W. A. Pearlman, "Hierarchical dynamic range coding of wavelet subbands for fast and efficient image decompression," *IEEE Transactions on Image Processing*, vol. 16, no. 8, pp. 2005–2015, 2007.

[38] P. A. Chou, T. Lookabaugh, and R. M. Gray, "Entropy-constrained vector quantization," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 1, pp. 31–42, 1989.

[39] C. Chrysafis, A. Said, A. Drukarev, A. Islam, and W. A. Pearlman, "SBHP-a low complexity wavelet coder," in *Acoustics, Speech, and Signal Processing*, vol. 4, 2000, pp. 2035–2038.

[40] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Batched multi triangulation," in *IEEE Visualization (VIS)*, 2005, pp. 207–214.

[41] P. Cignoni, F. Ganovelli, C. Montani, and R. Scopigno, "Reconstruction of topologically correct and adaptive trilinear isosurfaces," *Computers & Graphics*, vol. 24, no. 3, pp. 399–418, 2000.

[42] G. Cirio, G. Lavoué, and F. Dupont, "A framework for data-driven progressive mesh compression," in *Computer Graphics Theory and Applications (GRAPP)*, 2010, pp. 5–12.

[43] R. J. Clarke, *Transform Coding of Images*. Academic Press Professional, Inc., 1985.

[44] ——, *Digital Compression of Still Images and Video*. Academic Press, Inc., 1995.

[45] J. Clyne, "The multiresolution toolkit: Progressive access for regular gridded data," in *Visualization, Imaging, and Image Processing*, 2003, pp. 152–157.

[46] J. Clyne, P. Mininni, A. Norton, and M. Rast, "Interactive desktop analysis of high resolution simulations: Application to turbulent plume dynamics and current sheet formation," *New Journal of Physics*, vol. 9, no. 8, p. 301, 2007.

[47] A. Cohen, I. Daubechies, and J.-C. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Communications on Pure and Applied Mathematics*, vol. 45, no. 5, pp. 485–560, 1992.

[48] A. W. Cook, W. Cabot, and P. L. Miller, "The mixing transition in Rayleigh–Taylor instability," *Journal of Fluid Mechanics*, vol. 511, pp. 333–362, 2004.

[49] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Symposium on Interactive 3D Graphics and Games*, 2009, pp. 15–22.

[50] D. Cutting and J. Pedersen, "Optimization for dynamic inverted index maintenance," in *ACM SIGIR Research and Development in Information Retrieval*, 1989, pp. 405–411.

[51] I. Daubechies, "Orthonormal bases of compactly supported wavelets," *Communications on Pure and pplied mathematics*, vol. 41, no. 7, pp. 909–996, 1988.

[52] ——, *Ten Lectures on Wavelets*. SIAM, 1992.

[53] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *The Journal of Fourier Analysis and Applications*, vol. 4, no. 3, p. 247–269, 1998.

[54] L. De Fioriani, P. Magillo, and E. Puppo, "Building and traversing a surface at variable resolution," in *IEEE Visualization (VIS)*, 1997, pp. 103–110.

[55] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the best rank-1 and rank-$(r_1, r_2, \ldots, r_n)$ approximation of higher-order tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1324–1342, 2000.

[56] P. Deutsch, "DEFLATE compressed data format specification version 1.3," Tech. Rep., 1996.

[57] O. Devillers and P. Gandoin, "Geometric compression for interactive transmission," in *IEEE Visualization*, 2000, pp. 319–326.

[58] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 730–739.

[59] J. Díaz, F. Marton, and E. Gobbetti, "Interactive spatio-temporal exploration of massive time-varying rectilinear scalar volumes based on a variable bit-rate sparse representation over learned dictionaries," *Computers & Graphics*, vol. 88, pp. 45–56, 2020.

[60] J. Digne, R. Chaine, and S. Valette, "Self-similarity for accurate compression of point sampled surfaces," *Computer Graphics Forum*, vol. 33, no. 2, pp. 155–164, 2014.

[61] Y. Dodge, *The Concise Encyclopedia of Statistics*. Springer, 2008.

[62] D. A. Donzis, P. Yeung, and D. Pekurovsky, "Turbulence simulations on $O(10^4)$ processors," in *TeraGrid*, 2008.

[63] R. Dorfman, "The detection of defective members of large populations," *The Annals of Mathematical Statistics*, vol. 14, no. 4, pp. 436–440, 1943.

[64] O. Dovrat, I. Lang, and S. Avidan, "Learning to sample," in *IEEE/CVF Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 2755–2764.

[65] D. Du, F. K. Hwang, and F. Hwang, *Combinatorial Group Testing and Its Applications*. World Scientific, 1999.

[66] Z. Du, P. Jaromersky, Y. Chiang, and N. Memon, "Out-of-core progressive lossless compression and selective decompression of large triangle meshes," in *Data Compression Conference (DCC)*, 2009, pp. 420–429.

[67] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler *et al.*, "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3217–3227, 2014.

[68] S. Dunne, S. Napel, and B. Rutt, "Fast reprojection of volume data," in *Conference on Visualization in Biomedical Computing*, 1990, pp. 11–12.

[69] C. Egho and T. Vladimirova, "Adaptive hyperspectral image compression using the KLT and integer KLT algorithms," in *NASA/ESA Adaptive Hardware and Systems (AHS)*, 2014, pp. 112–119.

[70] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[71] D. A. Ellsworth, C. E. Henze, and B. C. Nelson, "Interactive visualization of high-dimensional petascale ocean data," in *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, 2017, pp. 36–44.

[72] 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, pp. 29–es.

[73] T. Etiene, L. G. Nonato, C. Scheidegger, J. Tienry, T. J. Peters, V. Pascucci, R. M. Kirby, and C. T. Silva, "Topology verification for isosurface extraction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 6, pp. 952–965, 2011.

[74] T. Etiene, C. Scheidegger, L. G. Nonato, R. M. Kirby, and C. Silva, "Verifiable visualization for isosurface extraction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1227–1234, 2009.

[75] Y. Fan, Y. Huang, and J. Peng, "Point cloud compression based on hierarchical point clustering," in *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, 2013, pp. 1–7.

[76] S. Fleishman, D. Cohen-Or, M. Alexa, and C. T. Silva, "Progressive point set surfaces," *ACM Transactions on Graphics*, vol. 22, no. 4, pp. 997–1011, 2003.

[77] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher, "Large data visualization on distributed memory multi-GPU clusters," in *High Performance Graphics*, 2010, pp. 57–66.

[78] T. Fogal, A. Schiewe, and J. Krüger, "An analysis of scalable GPU-based ray-guided volume rendering," in *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, 2013, pp. 43–51.

[79] M. Folk, A. Cheng, and K. Yates, "HDF5: A file format and I/O library for high performance computing applications," in *Supercomputing*, vol. 99, 1999, pp. 5–33.

[80] E. W. Forgy, "Cluster analysis of multivariate data: Efficiency versus interpretability of classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[81] N. Fout, H. Akiba, K.-L. Ma, A. E. Lefohn, and J. Kniss, "High-quality rendering of compressed volume data formats," in *EuroVis*, 2005, pp. 77–84.

[82] N. Fout and K.-L. Ma, "Transform coding for hardware-accelerated volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1600–1607, 2007.

[83] N. Fout and K. Ma, "An adaptive prediction-based approach to lossless compression of floating-point volume data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2295–2304, 2012.

[84] J. E. Fowler and R. Yagel, "Lossless compression of volume data," in *Symposium on Volume Visualization*, 1994, pp. 43–50.

[85] R. Fraedrich, J. Schneider, and R. Westermann, "Exploring the millennium run - scalable rendering of large-scale cosmological datasets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1251–1258, 2009.

[86] D. C. Garcia, T. A. Fonseca, R. U. Ferreira, and R. L. d. Queiroz, "Geometry coding for dynamic voxelized point clouds using octrees and multiple contexts," *IEEE Transactions on Image Processing*, vol. 29, pp. 313–322, 2020.

[87] J. Garcke, "Sparse grids in a nutshell," in *Sparse Grids and Applications*, 2012, pp. 57–80.

[88] A. Gersho, "Principles of quantization," *IEEE Transactions on Circuits and Systems*, vol. 25, no. 7, pp. 427–436, 1978.

[89] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*. Springer, 1992.

[90] H. Gish and J. Pierce, "Asymptotically efficient quantizing," *IEEE Transactions on Information Theory*, vol. 14, no. 5, pp. 676–683, 1968.

[91] E. Gobbetti, J. A. Iglesias Guitián, and F. Marton, "COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks," in *Computer Graphics Forum*, vol. 31, no. 3pt4, 2012, pp. 1315–1324.

[92] E. Gobbetti and F. Marton, "Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models," *Computers & Graphics*, vol. 28, no. 6, pp. 815–826, 2004.

[93] E. Gobbetti, F. Marton, and J. A. Iglesias Guitián, “A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets,” *The Visual Computer*, vol. 24, no. 7, pp. 797–806, 2008.

[94] T. Golla and R. Klein, “Real-time point cloud compression,” in *IEEE/RSJ Intelligent Robots and Systems (IROS)*, 2015, pp. 5087–5092.

[95] S. Golomb, “Run-length encodings,” *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, 1966.

[96] Z. Gong, T. Rogers, J. Jenkins, H. Kolla, S. Ethier, J. Chen, R. Ross, S. Klasky, and N. Samatova, “MLOC: Multi-level layout optimization framework for compressed scientific data exploration with heterogeneous access patterns,” in *International Conference on Parallel Processing (ICPP)*, 2012, pp. 239–248.

[97] A. Gosselin-Ildari, “The deep scaly project, "lampropeltis getula" (on-line), digital morphology,” http://digimorph.org/specimens/Lampropeltis_getula/adult/ (accessed on July 23, 2018), 2006.

[98] P. Goswami, F. Erol, R. Mukhi, R. Pajarola, and E. Gobbetti, “An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees,” *The Visual Computer*, vol. 29, no. 1, pp. 69–83, 2013.

[99] R. Gray, “Vector quantization,” *IEEE ASSP Magazine*, vol. 1, no. 2, pp. 4–29, 1984.

[100] M. H. Gross, O. G. Staadt, and R. Gatti, “Efficient triangular surface approximations using wavelets and quadtree data structures,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 2, pp. 130–143, 1996.

[101] S. Grottel, P. Beck, C. Müller, G. Reina, J. Roth, H. Trebin, and T. Ertl, “Visualization of electrostatic dipoles in molecular dynamics of metal oxides,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2061–2068, 2012.

[102] A. F. Guarda, N. M. Rodrigues, and F. Pereira, “Adaptive deep learning-based point cloud geometry coding,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 15, no. 2, pp. 415–430, 2020.

[103] S. Gumhold, Z. Kami, M. Isenburg, and H.-P. Seidel, “Predictive point-cloud compression,” in *ACM SIGGRAPH Sketches*, 2005, p. 137.

[104] F. Guo, H. Li, W. Daughton, and Y.-H. Liu, “Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection,” *Physical Review Letters*, vol. 113, no. 15, p. 155005, 2014.

[105] S. Guthe and W. Straßer, “Advanced techniques for high-quality multi-resolution volume rendering,” *Computers & Graphics*, vol. 28, no. 1, pp. 51–58, 2004.

[106] S. Guthe, M. Wand, J. Gonser, and W. Straßer, “Interactive rendering of large volume data sets,” in *IEEE Visualization (VIS)*, 2002, pp. 53–60.

[107] A. Haar, *Zur theorie der orthogonalen funktionensysteme.* Georg-August-Universitat, Gottingen., 1909.

[108] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, V. Vishwanath, Z. Lukić, S. Sehrish, and W.-k. Liao, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.

[109] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister, "Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2285–2294, 2012.

[110] C. Harrison, H. Childs, and K. P. Gaither, "Data-parallel mesh connected components labeling and analysis," in *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2011.

[111] E. Hellinger, "Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen." *Journal für die reine und angewandte Mathematik*, vol. 1909, no. 136, pp. 210–271, 1909.

[112] A. J. Hey, S. Tansley, K. M. Tolle *et al.*, *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft research Redmond, WA, 2009, vol. 1.

[113] D. Hoang, H. Bhatia, P. Lindstrom, and V. Pascucci, "High-quality and low-memory-footprint progressive decoding of large-scale particle data," in *IEEE Symposium on Large Data Analysis and Visualization*, 2021, pp. 32–42.

[114] D. Hoang, P. Klacansky, H. Bhatia, P.-T. Bremer, P. Lindstrom, and V. Pascucci, "A study of the trade-off between reducing precision and reducing resolution for data analysis and visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 1193–1203, 2019.

[115] D. Hoang, B. Summa, H. Bhatia, P. Lindstrom, P. Klacansky, W. Usher, P. Bremer, and V. Pascucci, "Efficient and flexible hierarchical data layouts for a unified encoding of scalar field precision and resolution," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 603–613, 2021.

[116] E. S. Hong and R. E. Ladner, "Group testing for image compression," *IEEE Transactions on Image Processing*, vol. 11, no. 8, pp. 901–911, 2002.

[117] M. Hopf, M. Luttenberger, and T. Ertl, "Hierarchical splatting of scattered 4D data," *IEEE Computer Graphics and Applications*, vol. 24, no. 4, pp. 64–72, 2004.

[118] M. Hosseini and C. Timmerer, "Dynamic adaptive point cloud streaming," in *Packet Video Workshop (PV)*, 2018, pp. 25–30.

[119] L. Huang, S. Wang, K. Wong, J. Liu, and R. Urtasun, "Octsqueeze: Octree-structured entropy model for lidar compression," in *IEEE/CVF Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 1313–1323.

[120] Y. Huang, J. Peng, C.-J. Kuo, and M. Gopi, "A generic scheme for progressive point cloud coding," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 2, pp. 440–453, 2008.

[121] E. Hubo, T. Mertens, T. Haber, and P. Bekaert, "The quantized kd-tree: Efficient ray tracing of compressed point clouds," in *IEEE Symposium on Interactive Ray Tracing (RT)*, 2006, pp. 105–113.

[122] ——, "Self-similarity based compression of point set surfaces with application to ray tracing," *Computers & Graphics*, vol. 32, no. 2, pp. 221–234, 2008.

[123] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[124] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, "Out-of-core compression and decompression of large n-dimensional scalar fields," in *Computer Graphics Forum*, vol. 22, no. 3, 2003, pp. 343–348.

[125] I. Ihm and S. Park, "Wavelet-based 3D compression scheme for interactive visualization of very large volume data," in *Computer Graphics Forum*, vol. 18, no. 1, 1999, pp. 3–15.

[126] H. G. Im, P. G. Arias, S. Chaudhuri, and H. A. Uranakara, "Direct numerical simulations of statistically stationary turbulent premixed flames," *Combustion Science and Technology*, vol. 188, no. 8, pp. 1182–1198, 2016.

[127] M. Isenburg, "LASzip: Lossless compression of LiDAR data," *Photogrammetric Engineering & Remote Sensing*, vol. 79, no. 2, 2013.

[128] J. Iverson, C. Kamath, and G. Karypis, "Fast and effective lossy compression algorithms for scientific datasets," in *International Conference on Parallel Processing*, 2012, pp. 843–856.

[129] C. L. Jackins and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.

[130] N. S. Jayant and P. Noll, *Digital coding of waveforms: principles and applications to speech and video*. Prentice Hall, 1984.

[131] S. Jin, P. Grosset, C. M. Biwer, J. Pulido, J. Tian, D. Tao, and J. P. Ahrens, "Understanding GPU-based lossy compression for extreme-scale cosmological simulations," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2020, pp. 105–115.

[132] S. Jin, J. Pulido, P. Grosset, J. Tian, D. Tao, and J. Ahrens, "Adaptive configuration of in situ lossy compression for cosmology simulations via fine-grained rate-quality modeling," in *Symposium on High-Performance Parallel and Distributed Computing*, 2020, p. 45–56.

[133] H. Kalva, "The H.264 video coding standard," *IEEE Multimedia*, vol. 13, no. 4, pp. 86–90, 2006.

[134] M. Keller, D. S. Schimel, W. W. Hargrove, and F. M. Hoffman, "A continental strategy for the national ecological observatory network," *The Ecological Society of America: 282-284*, 2008.

[135] J. E. S. Khalil, A. Munteanu, L. Denis, P. Lambert, and R. V. d. Walle, "Scalable feature-preserving irregular mesh coding," *Computer Graphics Forum*, vol. 36, no. 6, pp. 275–290, 2017.

[136] T.-J. Kim, B. Moon, D. Kim, and S.-E. Yoon, "RACBVHs: random-accessible compressed bounding volume hierarchies," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 2, pp. 273–286, 2010.

[137] D. King and J. Rossignac, "Optimal bit allocation in compressed 3D models," *Computational Geometry*, vol. 14, no. 1, pp. 91–118, 1999.

[138] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[139] P. Komma, J. Fischer, F. Duffner, and D. Bartz, "Lossless volume data compression schemes." in *SimVis*, 2007, pp. 169–182.

[140] L. G. Kraft, "A device for quantizing, grouping, and coding amplitude-modulated pulses," Ph.D. dissertation, Massachusetts Institute of Technology, 1949.

[141] M. Krivokuća, P. A. Chou, and M. Koroteev, "A volumetric approach to point cloud compression – Part II: Geometry compression," *IEEE Transactions on Image Processing*, vol. 29, pp. 2217–2229, 2020.

[142] M. Krivokuća, P. A. Chou, and P. Savill, "8i voxelized surface light field (8ivslf) dataset," *ISO/IEC JTC1/SC29 WG11 (MPEG) input document m42914*, pp. 61–70, 2018.

[143] J. Krüger, J. Schneider, and R. Westermann, "Compression and rendering of iso-surfaces and point sampled geometry," *The Visual Computer*, vol. 22, pp. 517–530, 2006.

[144] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[145] S. Kumar, C. Christensen, J. A. Schmidt, P. Bremer, E. Brugger, V. Vishwanath, P. H. Carns, H. Kolla, R. W. Grout, J. Chen, M. Berzins, G. Scorzelli, and V. Pascucci, "Fast multiresolution reads of massive simulation datasets," in *International Supercomputing Conference (ISC)*, 2014, pp. 314–330.

[146] S. Kumar, J. Edwards, P.-T. Bremer, A. Knoll, C. Christensen, V. Vishwanath, P. Carns, J. A. Schmidt, and V. Pascucci, "Efficient I/O and storage of adaptive-resolution data," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 413–423.

[147] S. Kumar, D. Hoang, S. Petruzza, J. Edwards, and V. Pascucci, "Reducing network congestion and synchronization overhead during aggregation of hierarchical data," in *International Conference on High Performance Computing (HiPC)*, 2017, pp. 223–232.

[148] S. Kumar, V. Vishwanath, P. Carns, J. A. Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M. E. Papka *et al.*, "Efficient data restructuring and aggregation for I/O acceleration in PIDX," in *IEEE High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–11.

[149] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout, "PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets," in *IEEE Cluster Computing (CLUSTER)*, 2011, pp. 103–111.

[150] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data," in *European Conference on Parallel Processing*, 2011, pp. 366–379.

[151] E. LaMar, B. Hamann, and K. I. Joy, "Multiresolution techniques for interactive texture-based volume visualization," in *IEEE Visualization Conference*, 1999, pp. 355–361.

[152] D. Laney, S. Langer, C. Weber, P. Lindstrom, and A. Wegener, "Assessing the effects of data compression in simulations using physically motivated metrics," in *IEEE High Performance Computing, Networking, Storage and Analysis (SC)*, 2013, pp. 1–12.

[153] S. Lasserre, D. Flynn, and S. Qu, "Using neighbouring nodes for the compression of octrees representing the geometry of point clouds," in *ACM Multimedia Systems Conference (MMSys)*, 2019, pp. 145–153.

[154] N. K. Laurance and D. M. Monro, "Embedded DCT coding with significance masking," in *IEEE Acoustics, Speech, and Signal Processing*, vol. 4, 1997, pp. 2717–2720.

[155] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," in *Computer Graphics Forum*, vol. 28, no. 2, 2009, pp. 375–384.

[156] M. Le Muzic, L. Autin, J. Parulek, and I. Viola, "cellVIEW: A tool for illustrative and multi-scale rendering of large biomolecular datasets," in *Eurographics Workshop on Visual Computing for Biomedicine*, vol. 2015, 2015, p. 61.

[157] H. Lee, M. Desbrun, and P. Schröder, "Progressive encoding of complex isosurfaces," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 471–476, 2003.

[158] H. Lee, G. Lavoué, and F. Dupont, "Rate-distortion optimization for progressive compression of 3D mesh with color attributes," *The Visual Computer*, vol. 28, pp. 137–153, 2012.

[159] M. Lee and R. D. Moser, "Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$," *Journal of Fluid Mechanics*, vol. 774, pp. 395–415, 2015.

[160] S. Lefebvre and H. Hoppe, "Compressed random-access trees for spatially coherent data," in *Eurographics Symposium on Rendering Techniques*, 2007, pp. 339–349.

[161] M. Levoy, *Volume Rendering Using the Fourier Projection-Slice Theorem*. Computer Systems Laboratory, Stanford University, 1992.

[162] S. Li, S. Jaroszynski, S. Pearse, L. Orf, and J. Clyne, "VAPOR: A visualization package tailored to analyze simulation data in earth system science," *Atmosphere*, vol. 10, no. 9, p. 488, 2019.

[163] S. Li, S. Sane, L. Orf, P. Mininni, J. Clyne, and H. Childs, "Spatiotemporal wavelet compression for visualization of scientific simulation data," in *IEEE Cluster Computing (CLUSTER)*, 2017, pp. 216–227.

[164] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *IEEE Big Data (BigData)*, 2018, pp. 438–447.

[165] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao *et al.*, "SZ3: A modular framework for composing prediction-based error-bounded lossy compressors," *arXiv preprint arXiv:2111.02925*, 2021.

[166] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *IEEE Transactions on communications*, vol. 28, no. 1, pp. 84–95, 1980.

[167] P. Lindstrom, "ZFP 0.5.5 documentation," accessed: 2021-09-15.

[168] ——, "Out-of-core construction and visualization of multiresolution surfaces," in *Symposium on Interactive 3D Graphics and Games (I3D)*, 2003, pp. 93–102.

[169] ——, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[170] ——, "Reducing data movement using adaptive-rate computing," 2018, https://computation.llnl.gov/sites/default/files/public//llnl-post-728998.pdf (accessed on June 13, 2018). [Online]. Available: https://computation.llnl.gov/sites/default/files/public//llnl-post-728998.pdf

[171] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[172] L. Linsen, V. Pascucci, M. Duchaineau, B. Hamann, and K. Joy, "Wavelet-based multiresolution with n-th-root-of-2," in *Dagstuhl Seminar on Geometric Modeling*, 2004.

[173] H. Liu, H. Yuan, Q. Liu, J. Hou, and J. Liu, "A comprehensive study and comparison of core technologies for MPEG 3D point cloud compression," *IEEE Transactions on Broadcasting*, vol. 66, no. 3, pp. 701–717, 2020.

[174] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield *et al.*, "Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[175] P. Ljung, C. Lundström, and A. Ynnerman, "Multiresolution interblock interpolation in direct volume rendering," 2006.

[176] P. Ljung, C. Lundstrom, A. Ynnerman, and K. Museth, "Transfer function based adaptive decompression for volume rendering of large medical data sets," in *IEEE Symposium on Volume Visualization and Graphics*, 2004, pp. 25–32.

[177] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[178] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible I/O and integration for scientific codes through the adaptable io system (ADIOS)," in *International Workshop on Challenges of Large Applications in Distributed Environments*, 2008, pp. 15–24.

[179] T. D. Lookabaugh and R. M. Gray, "High-resolution quantization theory and the vector quantizer advantage," *IEEE Transactions on Information Theory*, vol. 35, no. 5, pp. 1020–1033, 1989.

[180] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[181] E. B. Lum, K. L. Ma, and J. Clyne, "Texture hardware assisted rendering of time-varying volume data," in *IEEE Visualization (VIS)*, 2001, pp. 263—-270.

[182] A. Maglo, C. Courbet, P. Alliez, and C. Hudelot, "Progressive compression of manifold polygon meshes," *Computers & Graphics*, vol. 36, no. 5, pp. 349–359, 2012.

[183] H. S. Malvar, "Fast progressive wavelet coding," in *Data Compression Conference*, 1999, pp. 336–343.

[184] T. Malzbender, "Fourier volume rendering," *ACM Transactions on Graphics*, vol. 12, no. 3, pp. 233–250, 1993.

[185] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 620–636, 2003.

[186] G. N. N. Martin, "Range encoding: An algorithm for removing redundancy from a digitised message," in *International Conference on Video and Data Recording*, 1979, p. 48.

[187] O. Martinez Rubi, S. Verhoeven, M. van Meersbergen, M. Schütz, P. Oosterom, R. Goncalves, and T. Tijssen, "Taming the beast: Free and open-source massive point cloud web visualization," in *Capturing Reality Forum*, 2015.

[188] F. Marton, M. Agus, and E. Gobbetti, "A framework for GPU-accelerated exploration of massive time-varying rectilinear scalar volumes," in *Computer Graphics Forum*, vol. 38, no. 3, 2019, pp. 53–66.

[189] K. E. Matheson, K. K. Cross, M. M. Nowell, and A. D. Spear, "A multiscale comparison of stochastic open-cell aluminum foam produced via conventional and additive-manufacturing routes," *Materials Science and Engineering: A*, vol. 707, pp. 181–192, 2017.

[190] J. Max, "Quantizing for minimum distortion," *IRE Transactions on Information Theory*, vol. 6, no. 1, pp. 7–12, 1960.

[191] J. D. McCalpin, "Memory bandwidth and system balance in HPC systems," *UT Faculty/Researcher Works*, 2016.

[192] M. McGuire, "Computer graphics archive," https://casual-effects.com/data, accessed: 2021-03-31.

[193] B. McMillan, "Two inequalities implied by unique decipherability," *IRE Transactions on Information Theory*, vol. 2, no. 4, pp. 115–116, 1956.

[194] R. Mekuria, K. Blom, and P. Cesar, "Design, implementation, and evaluation of a point cloud codec for tele-immersive video," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 4, pp. 828–842, 2017.

[195] D. Menemenlis, J.-M. Campin, P. Heimbach, C. Hill, T. Lee, A. Nguyen, M. Schodlok, and H. Zhang, "ECCO2: High resolution global ocean and sea ice data synthesis," *Mercator Ocean Quarterly Newsletter*, vol. 31, no. October, pp. 13–21, 2008.

[196] Q. Meng, A. Humphrey, and M. Berzins, "The UINTAH framework: A unified heterogeneous task scheduling and runtime system," in *SC Companion: High Performance Computing, Networking Storage and Analysis (SCC)*, pp. 2441–2448.

[197] B. Merry, P. Marais, and J. Gain, "Compression of dense and regular point clouds," in *Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (AFRIGRAPH)*, 2006, pp. 15–20.

[198] A. Metere, S. Sarman, T. Oppelstrup, and M. Dzugutov, "Formation of a columnar liquid crystal in a simple one-component system of particles," *Soft Matter*, vol. 11, no. 23, pp. 4606–4613, 2015.

[199] S. Milani, E. Polo, and S. Limuti, "A transform coding strategy for dynamic point clouds," *IEEE Transactions on Image Processing*, vol. 29, pp. 8213–8225, 2020.

[200] A. Moffat, R. Neal, and I. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, p. 256–294, 1998.

[201] A. Moffat and L. Stuiver, "Exploiting clustering in inverted file compression," in *Data Compression Conference*, 1996, pp. 82–91.

[202] ——, "Binary interpolative coding for effective index compression," *Information Retrieval*, vol. 3, no. 1, pp. 25–47, 2000.

[203] P. Moin and K. Mahesh, "Direct numerical simulation: a tool in turbulence research," *Annual Review of Fluid Mechanics*, vol. 30, no. 1, pp. 539–578, 1998.

[204] P. Moran and D. Ellsworth, "Visualization of AMR data with multi-level dual-mesh interpolation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 1862–1871, 2011.

[205] K. Museth, "VDB: High-resolution sparse volumes with dynamic topology," *ACM Transactions on Graphics*, vol. 32, no. 3, pp. 1–22, 2013.

[206] D. T. Nguyen, M. Quach, G. Valenzise, and P. Duhamel, "Lossless coding of point cloud geometry using a deep generative model," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 12, pp. 4617–4629, 2021.

[207] K. G. Nguyen and D. Saupe, "Rapid high quality compression of volume data for visualization," *Computer Graphics Forum*, vol. 20, no. 3, pp. 49–57, 2002.

[208] G. M. Nielson, "On marching cubes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 283–297, 2003.

[209] P. Ning and L. Hesselink, "Vector quantization for volume rendering," in *ACM Workshop on Volume visualization*, 1992, pp. 69–74.

[210] T. Ochotta and D. Saupe, "Image-based surface compression," *Computer Graphics Forum*, vol. 27, no. 6, pp. 1647–1663, 2008.

[211] B. Oliver, J. Pierce, and C. E. Shannon, "The philosophy of PCM," *Proceedings of the IRE*, vol. 36, no. 11, pp. 1324–1331, 1948.

[212] A. Omeltchenko, T. J. Campbell, R. K. Kalia, X. Liu, A. Nakano, and P. Vashishta, "Scalable I/O of large-scale molecular dynamics simulations: A data-compression algorithm," *Computer Physics Communications*, vol. 131, no. 1-2, pp. 78–85, 2000.

[213] E. Ordentlich, M. Weinberger, and G. Seroussi, "A low-complexity modeling approach for embedded coding of wavelet coefficients," in *Data Compression Conference*, 1998, pp. 408–417.

[214] S. J. Orfanidis, *Applied optimum signal processing*. McGraw-Hill, 2018.

[215] B. W. O'shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing Enzo, an AMR cosmology application," in *Adaptive Mesh Refinement-Theory and Applications: Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3–5, 2003*, 2005, pp. 341–349.

[216] R. Pajarola, "Efficient level-of-details for point based rendering," in *International Conference on Computer Graphics and Imaging (IASTED)*, 2003, pp. 141–146.

[217] J. Pantaleoni and D. Luebke, "HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry," in *High Performance Graphics*, 2010, pp. 87–95.

[218] P. Panter and W. Dite, "Quantization distortion in pulse-count modulation with nonuniform spacing of levels," *Proceedings of the IRE*, vol. 39, no. 1, pp. 44–48, 1951.

[219] S. Park and S. Lee, "Multiscale representation and compression of 3D point data," *IEEE Transactions on Multimedia*, vol. 11, no. 1, pp. 177–183, 2009.

[220] R. Parys and G. Knittel, "Interactive large-scale volume rendering," in *High End Visualization Workshop*, vol. 6, no. 9, 2009, p. 13.

[221] ——, "Giga-voxel rendering from compressed data on a display wall," *Journal of WSCG*, vol. 17, pp. 73–80, 2009.

[222] R. Pasco, "Source coding algorithms for fast data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 4, pp. 548–548, 1977.

[223] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2001, pp. 45–45.

[224] V. Pascucci, D. E. Laney, R. J. Frank, G. Scorzelli, L. Linsen, B. Hamann, and F. Gygi, "Real-time monitoring of large scientific simulations," in *Proceedings of the 2003 ACM Symposium on Applied Computing*, 2003, pp. 194–198.

[225] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "BD-CATS: Big data clustering at trillion particle scale," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015, pp. 1–12.

[226] M. Pauly, M. Gross, and L. P. Kobbelt, "Efficient simplification of point-sampled surfaces," in *IEEE Visualization (VIS)*, 2002, pp. 163–170.

[227] W. A. Pearlman, A. Islam, N. Nagaraj, and A. Said, "Efficient, low-complexity image coding with a set-partitioning embedded block coder," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 11, pp. 1219–1235, 2004.

[228] W. A. Pearlman, A. Said *et al.*, "Image wavelet coding systems: Part II of set partition coding and image wavelet coding systems," *Foundations and Trends® in Signal Processing*, vol. 2, no. 3, pp. 181–246, 2008.

[229] ——, "Set partition coding: Part I of set partition coding and image wavelet coding systems," *Foundations and Trends® in Signal Processing*, vol. 2, no. 2, pp. 95–180, 2008.

[230] S.-C. Pei and F.-C. Chen, "Subband image decomposition by mathematical morphology," in *Visual Communications and Image Processing'90: Fifth in a Series*, vol. 1360, 1990, pp. 293–301.

[231] J. Peng, Y. Huang, C.-C. J. Kuo, I. Eckstein, and M. Gopi, "Feature oriented progressive lossless mesh coding," *Computer Graphics Forum*, vol. 29, pp. 2029–2038, 2010.

[232] J. Peng and C. C. J. Kuo, "Octree-based progressive geometry encoder," in *Internet Multimedia Management Systems IV*, vol. 5242, 2003, pp. 301–311.

[233] J. Peng and C.-C. J. Kuo, "Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition," in *ACM Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2005, pp. 609–616.

[234] J. Peng and C.-J. Kuo, "Progressive geometry encoder using octree-based space partitioning," in *IEEE Multimedia and Expo (ICME)*, 2004, pp. 1–4.

[235] F. E. H. Pérez, N. Mukhadiyev, X. Xu, A. Sow, B. J. Lee, R. Sankaran, and H. G. Im, "Direct numerical simulations of reacting flows with detailed chemistry using many-core/GPU acceleration," *Computers & Fluids*, vol. 173, pp. 73–79, 2018.

[236] S. Petruzza, A. Gyulassy, S. Leventhal, J. J. Baglino, M. Czabaj, A. D. Spear, and V. Pascucci, "High-throughput feature extraction for measuring attributes of deforming open-cell foams," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 140–150, 2019.

[237] W. Pratt, W.-H. Chen, and L. Welch, "Slant transform image coding," *IEEE Transactions on Communications*, vol. 22, no. 8, pp. 1075–1093, 1974.

[238] S. Prohaska, A. Hutanu, R. Kahler, and H.-C. Hege, "Interactive exploration of large remote micro-CT scans," in *IEEE Visualization (VIS)*, 2004, pp. 345–352.

[239] J. Pulido, D. Livescu, K. Kanov, R. Burns, C. Canada, J. Ahrens, and B. Hamann, "Remote visual analysis of large turbulence databases at multiple scales," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 115–126, 2018.

[240] I. Ram, M. Elad, and I. Cohen, "Generalized tree-based wavelet transform," *IEEE Transactions on Signal Processing*, vol. 59, no. 9, pp. 4199–4209, 2011.

[241] F. Reichl, M. Treib, and R. Westermann, "Visualization of big SPH simulations via compressed octree grids," in *IEEE Big Data*, 2013, pp. 71–78.

[242] P. d. O. Rente, C. Brites, J. Ascenso, and F. Pereira, "Graph-based static 3D point clouds geometry coding," *IEEE Transactions on Multimedia*, vol. 21, no. 2, pp. 284–299, 2019.

[243] R. F. Rice, "Practical Universal Noiseless Coding," in *Applications of Digital Image Processing III*, vol. 0207, International Society for Optics and Photonics. SPIE, 1979, pp. 247 – 267.

[244] J. J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.

[245] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. Uram, and V. Vishwanath, "Large-scale parallel visualization of particle-based simulations using point sprites and level-of-detail," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2015, pp. 1–10.

[246] F. F. Rodler, "Wavelet based 3D compression with fast random access for very large volume data," in *Pacific Conference on Computer Graphics and Applications*, 1999, pp. 108–117.

[247] Y. Rubner, C. Tomasi, and L. J. Guibas, "A metric for distributions with applications to image databases," in *International Conference on Computer Vision*, 1998, pp. 59–66.

[248] S. Rusinkiewicz and M. Levoy, "QSplat: a multiresolution point rendering system for large meshes," in *ACM Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2000, pp. 343–352.

[249] A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 243–250, 1996.

[250] H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

[251] H. Samet and A. Kochut, "Octree approximation and compression methods," in *3DPVT*, 2002, pp. 460–469.

[252] S. A. Savari and R. G. Gallager, "Generalized Tunstall codes for sources with memory," *IEEE Transactions on Information Theory*, vol. 43, no. 2, pp. 658–668, 1997.

[253] K. Schatz, C. Müller, M. Krone, J. Schneider, G. Reina, and T. Ertl, "Interactive visual exploration of a trillion particles," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2016, pp. 56–64.

[254] R. Schnabel and R. Klein, "Octree-based point-cloud compression," in *Eurographics Point-Based Graphics (SPBG)*, 2006, pp. 111–121.

[255] R. Schnabel, S. Möser, and R. Klein, "A parallelly decodeable compression scheme for efficient point-cloud rendering," in *Eurographics Conference on Point-Based Graphics (SPBG)*, 2007, pp. 214–226.

[256] J. Schneider and R. Westermann, "Compression domain volume rendering," in *IEEE Visualization (VIS)*, 2003, pp. 293–300.

[257] S. Schwarz, M. Preda, V. Baroncini, M. Budagavi, P. Cesar, P. A. Chou, R. A. Cohen, M. Krivokuća, S. Lasserre, Z. Li, J. Llach, K. Mammou, R. Mekuria, O. Nakagami, E. Siahaan, A. Tabatabai, A. M. Tourapis, and V. Zakharchenko, "Emerging MPEG standards for point cloud compression," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 1, pp. 133–148, 2019.

[258] M. Schütz, S. Ohrhallinger, and M. Wimmer, "Fast out-of-core octree generation for massive point clouds," *Computer Graphics Forum*, vol. 39, no. 7, pp. 155–167, 2020.

[259] R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis, "SPGrid: A sparse paged grid structure applied to adaptive smoke simulation," *ACM Transactions on Graphics*, vol. 33, no. 6, pp. 1–12, 2014.

[260] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[261] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Transactions on Signal Processing*, vol. 41, no. 12, pp. 3445–3462, 1993.

[262] Y. Shoham and A. Gersho, "Efficient bit allocation for an arbitrary set of quantizers (speech coding)," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 9, pp. 1445–1453, 1988.

[263] J. M. Singh and P. J. Narayanan, "Progressive decomposition of point clouds without local planes," in *Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP)*, 2006, pp. 364–375.

[264] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG 2000 still image compression standard," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 36–58, 2001.

[265] S. Slattery, S. T. Reeve, C. Junghans, D. Lebrun-Grandié, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, A. Isner, K. Chong, S. Moore, T. Germann, J. Belak, and S. Mniszewski, "Cabana: A performance portable library for particle-based simulations," *Journal of Open Source Software*, vol. 7, no. 72, p. 4115, 2022.

[266] N. Smirnov, "Table for estimating the goodness of fit of empirical distributions," *The Annals of Mathematical Statistics*, vol. 19, no. 2, pp. 279–281, 1948.

[267] J. Smith, G. Petrova, and S. Schaefer, "Progressive encoding and compression of surfaces generated from point cloud data," *Computers & Graphics*, vol. 36, no. 5, pp. 341–348, 2012.

[268] P. Smith, J. Thornock, Y. Wu, S. Smith, B. Isaac, P. Chapman, D. Sloan, D. Turek, Y. Chen, and A. Levasseur, "Oxy-coal power boiler simulation and validation through extreme computing," in *International Conference on Numerical Combustion*, 2015.

[269] B. Stevens, M. Satoh, L. Auger, J. Biercamp, C. S. Bretherton, X. Chen, P. Düben, F. Judt, M. Khairoutdinov, D. Klocke *et al.*, "DYAMOND: The dynamics of the atmospheric general circulation modeled on non-hydrostatic domains," *Progress in Earth and Planetary Science*, vol. 6, no. 1, pp. 1–17, 2019.

[270] E. J. Stollnitz, T. D. DeRose, A. D. DeRose, and D. H. Salesin, *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann, 1996.

[271] G. Strang, "The discrete cosine transform," *SIAM Review*, vol. 41, no. 1, pp. 135–147, 1999.

[272] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, 2012.

[273] B. Summa, G. Scorzelli, M. Jiang, P.-T. Bremer, and V. Pascucci, "Interactive editing of massive imagery made simple: Turning Atlanta into Atlantis," *ACM Transactions on Graphics*, vol. 30, no. 2, pp. 1–13, 2011.

[274] 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," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2135–2143, 2011.

[275] S. K. Suter, M. Makhynia, and R. Pajarola, "Tamresh–tensor approximation multiresolution hierarchy for interactive volume visualization," in *Computer Graphics Forum*, vol. 32, no. 3pt2, 2013, pp. 151–160.

[276] M. Swain and D. Ballard, "Color indexing," *International Journal of Computer Vision*, vol. 7, no. 1, pp. 11–32, 1991.

[277] W. Sweldens, "Wavelets and the lifting scheme: A 5 minute tour," *Journal of Applied Mathematics and Mechanics*, vol. 76, pp. 41–44, 1996.

[278] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1129–1139.

[279] ——, "In-depth exploration of single-snapshot lossy compression techniques for N-body simulations," in *IEEE International Conference on Big Data*, 2017, pp. 486–493.

[280] ——, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1129–1139.

[281] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Transactions on Image Processing*, vol. 9, no. 7, pp. 1158–1170, 2000.

[282] D. S. Taubman and M. W. Marcellin, *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Springer, 2001.

[283] J. Teuhola, "A compression method for clustered bit-vectors," *Information Processing Letters*, vol. 7, no. 6, pp. 308–311, 1978.

[284] ——, "Tournament coding of integer sequences," *The Computer Journal*, vol. 52, no. 3, pp. 368–377, 2009.

[285] D. Thanou, P. A. Chou, and P. Frossard, "Graph-based compression of dynamic 3D point cloud sequences," *IEEE Transactions on Image Processing*, vol. 25, no. 4, pp. 1765–1778, 2016.

[286] T. Totsuka and M. Levoy, "Frequency domain volume rendering," in *Conference on Computer Graphics and Interactive Techniques*, 1993, pp. 271–278.

[287] M. Treib, K. Bürger, F. Reichl, C. Meneveau, A. Szalay, and R. Westermann, "Turbulence visualization at the terascale on desktop PCs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2169–2177, 2012.

[288] W. Usher, X. Huang, S. Petruzza, S. Kumar, S. R. Slattery, S. T. Reeve, F. Wang, C. R. Johnson, and V. Pascucci, "Adaptive spatially aware I/O for multiresolution particle data layouts," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 547–556.

[289] W. Usher, P. Klacansky, F. Federer, P.-T. Bremer, A. Knoll, J. Yarch, A. Angelucci, and V. Pascucci, "A virtual reality visualization tool for neuron tracing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 994–1003, 2017.

[290] S. Valette and R. Prost, "Wavelet-based progressive compression scheme for triangle meshes: wavemesh," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 2, pp. 123–129, 2004.

[291] S. Valette, R. Chaine, and R. Prost, "Progressive lossless mesh compression via incremental parametric refinement," *Computer Graphics Forum*, vol. 28, no. 5, pp. 1301–1310, 2009.

[292] P. van Oosterom and T. Vijlbrief, "The spatial tion code," in *International Symposium on Spatial Data Handling*, 1996, pp. 12–16.

[293] A. Venkat, D. Hoang, A. Gyulassy, P.-T. Bremer, F. Federer, A. Angelucci, and V. Pascucci, "High-quality progressive alignment of large 3D microscopy data," in *2022 IEEE 12th Symposium on Large Data Analysis and Visualization (LDAV)*, 2022, pp. 1–10.

[294] I. Viola, A. Kanitsar, and M. E. Gröller, "GPU-based frequency domain volume rendering," in *Spring Conference on Computer Graphics*, 2004, pp. 55–64.

[295] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gunther, and P. Navratil, "OSPRay - a CPU ray tracing framework for scientific visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 931–940, 2017.

[296] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka, "CPU ray tracing large particle data with balanced P-k-d trees," in *IEEE Visualization (VIS)*, 2015, pp. 57–64.

[297] I. Wald and H.-P. Seidel, "Interactive ray tracing of point-based models," in *Eurographics Symposium on Point-Based Graphics (SPBG)*, 2005, pp. 9–16.

[298] G. K. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[299] F. Wang, I. Wald, Q. Wu, W. Usher, and C. R. Johnson, "CPU isosurface ray tracing of adaptive mesh refinement data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 1142–1151, 2018.

[300] J. Wang, H. Zhu, H. Liu, and Z. Ma, "Lossy point cloud geometry compression via end-to-end learning," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 12, pp. 4909–4923, 2021.

[301] R. Wang, *Introduction to Orthogonal Transforms: With Applications in Data Processing and Analysis*. Cambridge University Press, 2012.

[302] M. Waschbüsch, M. Gross, F. Eberhard, E. Lamboray, and S. Würmlin, "Progressive compression of point-sampled models," in *Eurographics Symposium on Point-Based Graphics (SPBG)*, 2004, pp. 95–103.

[303] G. H. Weber, O. Kreylos, T. J. Ligocki, J. M. Shalf, H. Hagen, B. Hamann, and K. I. Joy, "Extraction of crack-free isosurfaces from adaptive mesh refinement data," in *Hierarchical and Geometrical Methods in Scientific Visualization*, 2003, pp. 19–40.

[304] M. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS," *IEEE Transactions on Image Processing*, vol. 9, no. 8, pp. 1309–1324, 2000.

[305] K. Weiss and P. Lindstrom, "Adaptive multilinear tensor product wavelets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 1, pp. 985–994, 2016.

[306] T. A. Welch, "A technique for high-performance data compression," *IEEE Computer*, vol. 17, no. 06, pp. 8–19, 1984.

[307] R. Westermann, "A multiresolution framework for volume rendering," in *Symposium on Volume Visualization*, 1994, pp. 51–58.

[308] G. Wetekam, D. Staneker, U. Kanus, and M. Wand, "A hardware architecture for multi-resolution volume rendering," in *ACM SIGGRAPH/EUROGRAPHICS Graphics Hardware*, 2005, pp. 45–51.

[309] W. Widanagamaachchi, P.-T. Bremer, C. Sewell, L.-T. Lo, J. Ahrens, and V. Pascuccik, "Data-parallel halo finding with variable linking lengths," in *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, 2014, pp. 27–34.

[310] W. Widanagamaachchi, J. Chen, P. Klacansky, V. Pascucci, H. Kolla, A. Bhagatwala, and P.-T. Bremer, "Tracking features in embedded surfaces: Understanding extinction in turbulent combustion," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2015, pp. 9–16.

[311] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufman, 1999.

[312] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[313] J. Woodring, J. Ahrens, J. Figg, J. Wendelberger, S. Habib, and K. Heitmann, "In-situ sampling of a large-scale particle simulation for interactive visualization and analysis," in *Eurographics Conference on Visualization (EuroVis)*, 2011, pp. 1151–1160.

[314] J. Woodring, J. Ahrens, J. Figg, J. Wendelberger, and K. Heitmann, "In-situ sampling of a large-scale particle simulation for interactive visualization and analysis," *Computer Graphics Forum*, vol. 30, no. 3, pp. 1151–1160, 2011.

[315] J. Woodring, S. Mniszewski, C. Brislawn, D. DeMarle, and J. Ahrens, “Revisiting wavelet compression for large-scale climate data using JPEG 2000 and ensuring data precision,” in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2011, pp. 31–38.

[316] Q. Wu, T. Xia, C. Chen, H.-Y. S. Lin, H. Wang, and Y. Yu, “Hierarchical tensor approximation of multi-dimensional visual data,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 1, pp. 186–199, 2007.

[317] X. Wu, “High-order context modeling and embedded conditional entropy coding of wavelet coefficients for image compression,” in *Signals, Systems and Computers*, vol. 2, 1997, pp. 1378–1382.

[318] B.-L. Yeo and B. Liu, “Volume rendering of DCT-based compressed 3D scalar data,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 1, pp. 29–43, 1995.

[319] C. S. Yoo, E. S. Richardson, R. Sankaran, and J. H. Chen, “A DNS study on the stabilization mechanism of a turbulent lifted ethylene jet flame in highly-heated coflow,” *Proceedings of the Combustion Institute*, vol. 33, no. 1, pp. 1619–1627, 2011.

[320] D. L. Youngs, “Numerical simulation of mixing by Rayleigh–Taylor and Richtmyer–Meshkov instabilities,” *Laser and Particle Beams*, vol. 12, no. 4, pp. 725–750, 1994.

[321] C. Zenger and W. Hackbusch, “Sparse grids,” in *Proceedings of the Research Workshop of the Israel Science Foundation on Multiscale Phenomenon, Modelling and Computation*, 1991, p. 86.

[322] D. Z. Zhang, Q. Zou, W. B. VanderHeyden, and X. Ma, “Material point method applied to multiphase flows,” *Journal of Computational Physics*, vol. 227, no. 6, pp. 3159–3173, 2008.

[323] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, “Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation,” in *IEEE Data Engineering (ICDE)*, 2021, pp. 1643–1654.

[324] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[325] ——, “Compression of individual sequences via variable-rate coding,” *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.