Numerical Methods for Non-Uniform Data Sources

by

Kevin Michael Doherty

B.S., University of Michigan, 2014

M.A., George Washington University, 2017

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Applied Mathematics

2024

Committee Members:

Stephen Becker, Chair

Prof. Alireza Doostan

Prof. Ian Grooms

Prof. William Kleiber

Prof. John Evans

Doherty, Kevin Michael (Ph.D., Applied Mathematics)

Numerical Methods for Non-Uniform Data Sources

Thesis directed by Prof. Stephen Becker

This thesis surveys and creates methods to allow for a mathematically consistent treatment of non-uniform data sources in machine learning and data compression. These methods are fundamentally rooted in numerical methods for quadrature and interpolation, but are leveraged with appropriate computational tools and techniques to adapt to their respective domains and problem types.

Dedication

To my Mom and Dad, who have imbued me with the grit I needed for this degree. To my sister Megan, whose goofy jokes have always added much needed levity. To my friends, without whom I wouldn't have a dedication to write. Finally, to Cait for all of your love and support.

Acknowledgements

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003968 as well as Department of Energy Advanced Scientific Computing Research Awards DE-SC0022283 and DE-SC0023346. We thank John Evans, Jeff Hadley, Kenneth Jansen, Angran Li, and Cooper Simpson from the University of Colorado Boulder for their helpful discussions surrounding this work. We also thank the PSAAP III team from Stanford University for its helpful discussions regarding state of the art partial differential equation simulations.

Contents

Chapter

Appendix

Tables

Table

Figures

Figure

Chapter 1

Introduction

Non-uniform data sources are found throughout climate modeling, advanced partial differential equation simulation and remote sensing. These sources can generate large volumes of data, and hold important scientific insights about our world. However, non-uniformity can destroy some of the regular structure that standard numerical techniques in machine learning and other fields rely upon. This thesis covers advances in numerical methods for non-uniform data sources. We call a data source non-uniform when it lacks the regular spacing in it's coordinate directions. We will work with data sources that have a non-uniform spatial discretization but have a tensor product structure, which we call a non-uniform grid, and also with those without a tensor product structure, which we call a non-uniform mesh. Much of this thesis will focus on data compression, although in some cases there are other applications as well, which we will address when appropriate.

Non-uniformly discretized data arise in many scientific applications. These discretizations can be chosen to better approximate functions of certain classes, to minimize computational cost or may arise naturally due to experimental sampling. Important examples of numerical methods that exhibit non-uniform structure are sparse grids [120], finite element methods [88], radial basis function interpolation [114], Gaussian quadrature [16] and Chebyshev interpolation [44]. These methods can be the origin of non-uniform data, if used in the setting of a partial differential equation (PDE) simulation, or be used to manipulate non-uniform data. Our data may live in 2D, 3D or in even higher dimensional spaces, depending

on the nature of the problem. We will focus on the application of data compression for these non-uniform data sources.

Data Compression Data compression began as a field concerned with the communication of information. Band-limited channels corrupted by noise [79] limited the information rate of transmitted messages. In 1948, this motivated Shannon to consider the structure of a message itself [95], and whether a message could be conveyed in fewer symbols with data efficient representations.

The famous Shannon Source Coding Theorem [95] its concerned with a random source, X, from which we can produce a sequence of i.i.d. variables $X_1, X_2, \ldots, X_n$. These variables take discrete values in a finite set of symbols $\mathcal{X}$. This random source can then be evaluated for Shannon entropy, $H(X)$, which takes the value,

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2(p(x)) \tag{1.1}$$

here $p(x)$ is the probability mass function and $\mathcal{X}$ is the set of possible values of X. We can assign each symbol a bit string of fixed length, which allows for a naive encoding into $n\lceil \log_2(|\mathcal{X}|) \rceil$ bits, or $\lceil \log_2(|\mathcal{X}|) \rceil$ bits **on average**. Furthermore, $H(X) \leq \log_2(|\mathcal{X}|)$, with equality achieved when the distribution over the symbols is uniform. This hints at the connection between encoding and entropy. The precise connection is in the Shannon Source Coding Theorem [95]:

Theorem 1. *Given a random source X with entropy $H(X)$, it is possible to encode the i.i.d. sequence $X_1, X_2, \ldots, X_n$ with $n(H(X) + \epsilon)$ bits for $\epsilon > 0$. This original sequence is then recoverable from the encoding with probability $1 - \epsilon$.*

A year later, Fano [26] devised a code which aimed to minimize the number of symbols required to transmit a message with low entropy but was only asymptotically optimal. Huffman perfected this code structure in 1952 [45][1] , and demonstrated that it is optimal for

[1] As a term paper for Fano's class, so the story goes.

a message set with given frequencies. Since then there have been a couple different entropy encoding schemes which can match or (marginally) exceed the efficiency Huffman's optimal code, in particular Arithmetic Range Coding [57] and Asymmetric Numeral Systems [23].

These entropy encoding techniques all deal directly with the compression of messages encoded as symbols, but the data we are interested in is not always so neatly packaged. Take the example of a PDE simulation, which may consist of petabytes of highly correlated spatio-temporal patterns, all governed by a small system of equations.These simulations—if directly interpreted as messages (with no regard for their spatio-temporal information)—would not have their low entropy readily exploited by entropy encoders shown above. In the remainder of this thesis we will explore the usage of linear and non-linear maps that aim to make spatio-temporal patterns more compressible, and in particular we will focus on non-uniform data sources, whose spatio-temporal patterns are harder to exploit than their counterparts on uniform grids.

Content Guide The second chapter of this thesis covers a neural network layer called "QuadConv" and was published in the Journal of Computational Physics [21]. The third chapter outlines improvements to "QuadConv" that make it a more competitive and efficient neural network layer. The fourth chapter introduces a more traditional data compression framework called `Mesh-Float-Zip` which competes with state of the art scientific data compression tools.

Chapter 2

Quadrature Convolutions

Kevin Doherty

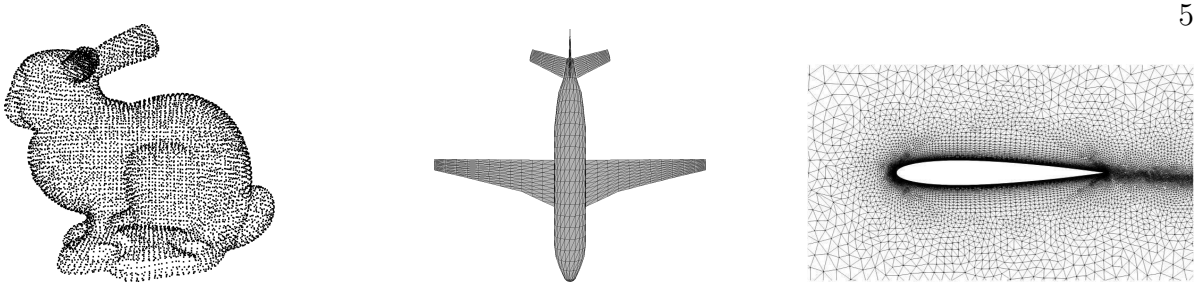
Cooper Simpson

Stephen Becker

Alireza Doostan

2.1 Abstract

We present a new convolution layer for deep learning architectures which we call QuadConv — an approximation to continuous convolution via quadrature. Our operator is developed explicitly for use on non-uniform, mesh-based data, and accomplishes this by learning a continuous kernel that can be sampled at arbitrary locations. Moreover, the construction of our operator admits an efficient implementation which we detail and construct. As an experimental validation of our operator, we consider the task of compressing partial differential equation (PDE) simulation data from fixed meshes. We show that QuadConv can match the performance of standard discrete convolutions on uniform grid data by comparing a QuadConv autoencoder (QCAE) to a standard convolutional autoencoder (CAE). Further, we show that the QCAE can maintain this accuracy even on non-uniform data. In both cases, QuadConv also outperforms alternative unstructured convolution methods such as graph convolution.



(a) 3D point cloud [100]. (b) Surface mesh [11]. (c) NACA 0012 mesh [98].

Figure 2.1: Examples of non-uniform data.

2.2 Introduction

Discrete convolutions are one of the canonical operations used in numerous deep learning applications such as image classification, object detection, semantic segmentation, etc. They have been proven to be effective in extracting important features from data, and they possess a number of desirable properties such as translation equivariance. In particular, when used with compactly supported kernels of fixed size, this facilitates the extraction of local features from the data and provides a significant boost to computational efficiency compared to a (square) fully-connected layer, reducing time complexity from quadratic in the input size to linear and memory from quadratic to a constant [37]. The effects of this are significant: each training epoch is faster, less overall training is required, and often results in better generalization with fewer parameters. However, these standard discrete convolutions rely on the assumption that the data is defined on a uniform grid. This limitation is unfortunate, as there are a host of settings where convolutions may be effective, but the relevant data is non-uniform; Fig. 2.1 shows a few representative examples.

In this paper we introduce a quadrature-based discrete convolution operator suitable for arbitrary meshes, which we call QuadConv. Our construction is based on the continuous definition of convolution, which, for two functions $f, g : \mathbb{R}^D \to \mathbb{R}$, is given by

$$(f * g)(\boldsymbol{y}) = \int_{\mathbb{R}^D} f(\boldsymbol{x}) \cdot g(\boldsymbol{y} - \boldsymbol{x}) \, d\boldsymbol{x}. \tag{2.1}$$

A variety of conditions on f and g can guarantee this integral is well defined, e.g., Young's Inequality, or, of particular relevance here, if f and g are compactly supported. Note that we use the term quadrature to refer to a weighted sum approximation of an integral. Some authors make a distinction between one dimensional quadrature and higher dimensional cubature which we will not employ. In general, our method is applicable to data on a mesh, which is a set of nodes with arbitrary non-intersecting connections. Within this setting, we will consider two particular sub-types of data: uniform grids and non-uniform meshes. The latter explicitly requires the nodes to be non-uniformly distributed in space. Our proposed method is mathematically quite simple, but, as we will see, is non-trivial to implement efficiently.

We will denote vectors using lowercase bold font (e.g., $\boldsymbol{x}$), and matrices or operators using uppercase bold font (e.g., $\boldsymbol{X}$). We will often refer to g as the kernel and f as the data. This distinction is important in this context, as the data, f, is given (and hence known only on the mesh nodes) and the kernel, g, is a learned map. Where appropriate, continuous functions will be referred to with an argument from their domain in parenthesis, $f(\cdot)$ or $f(\boldsymbol{x})$, and its discrete counterpart will be referred to as $f(\boldsymbol{x}_i)$, or, if the arguments are clear from context, with an index f_i.

To summarize our contributions, we propose a novel convolution operator for deep learning applications that is suitable for data on a non-uniform mesh. In addition to this, we discuss the practical implementation of our operator, showing that it is a computationally feasible approach. Lastly, we present the application of our work to autoencoder based data compression. We show that it matches the effectiveness of standard convolutions on uniform grid data, and performs equally well on non-uniform data.

In the remainder of this section we focus on motivating our approach and discussing the related literature. Section 2.3 will then introduce our mathematical formulation, with Section 2.3.1 describing the practical implementation. In Section 2.4 we apply our method to a number of datasets on uniform grids and non-uniform meshes. Section 2.5 summarizes

our work and presents possible directions for future research.

2.2.1 Motivation

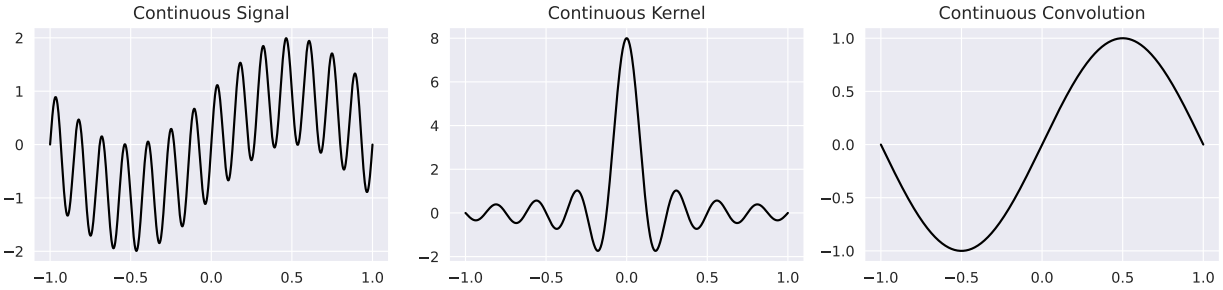
The standard convolution operator used in deep learning is a particular discretization of Eq. (2.1) operating on D-dimensional tensors. For example, for a single output and input channel, the one dimensional form is given as follows:

$$(f * g)_j = \sum_i f_i \cdot g_{j-i} \tag{2.2}$$

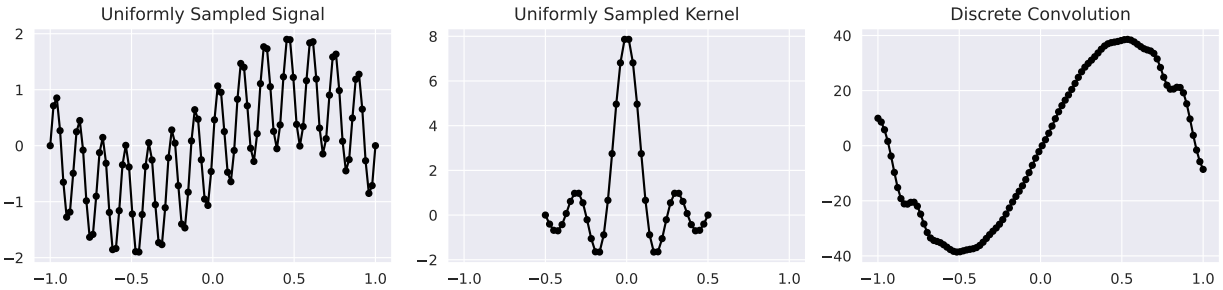
along with appropriate boundary conditions. This discretization is mathematically justified as long as the points are equally spaced, and is drawn from the definition of convolution for functions defined on $\mathbb{Z}$, which is the same as Eq. (2.2) up to notation. The following example will demonstrate how using Eq. (2.2) in the mathematically justified setting can yield accurate results, how this accuracy breaks down on non-uniformly spaced points, and how our proposed method fixes these issues. To that end, consider the following 1D functions:

$$f(x) = \sin(\pi x) + \sin(14\pi x) \qquad \text{and} \qquad g(x) = \frac{8\sin(8\pi x)}{\pi x},$$

where f is a signal composed of low and high frequency sine waves, and g is the ideal low-pass filter whose action under convolution will remove the higher frequency from f. In Fig. 2.2a we can see the result of this convolution computed analytically according to Eq. (2.1).

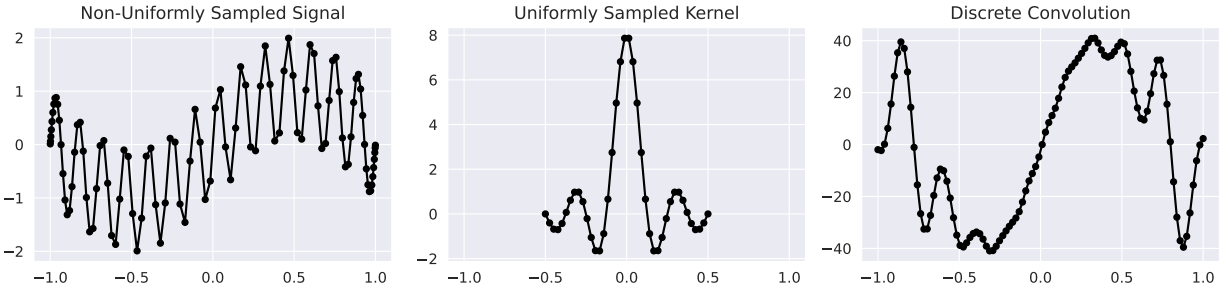


(a) Analytic convolution of continuous kernel and signal.

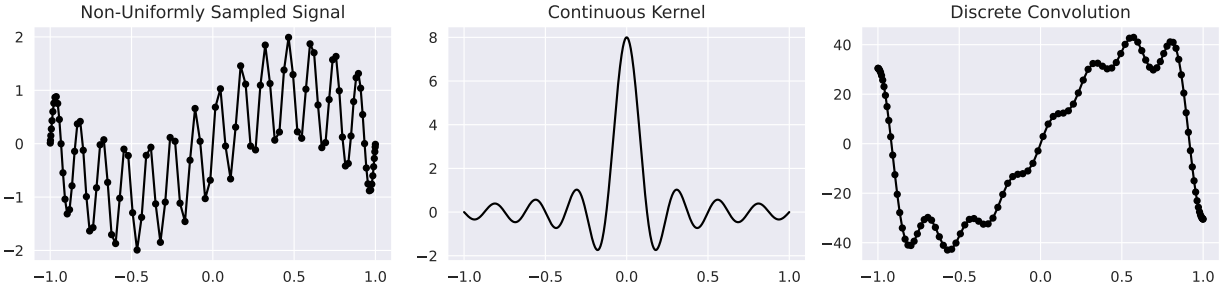


(b) Discrete convolution of uniformly sampled signal and kernel.

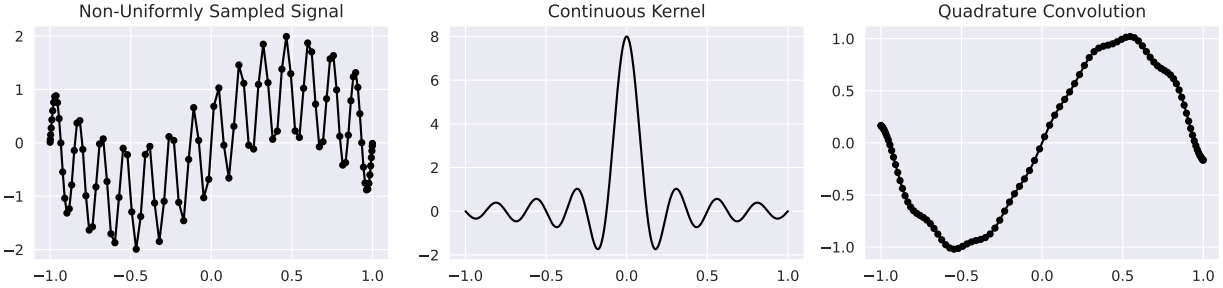
Figure 2.2: Comparison of continuous and discrete convolution.



(a) Discrete convolution with uniformly sampled kernel.



(b) Discrete convolution with continuous kernel.



(c) Quadrature-based convolution with continuous kernel.

Figure 2.3: Comparison of discrete convolution and quadrature-based convolution on a non-uniformly sampled signal.

In Fig. 2.2b, one can observe the normal usage of the discrete convolution, Eq. (2.2), with uniformly spaced grid points, and note that its result closely matches the continuous baseline. Figure 2.3, on the other hand, considers the results of a number of convolution type operations when the input signal is sampled in a non-uniform manner. Figure 2.3a

visualizes the results of discrete convolution if we sample the data at non-uniform locations, but continue to use the uniformly sampled kernel. This result only vaguely resembles the analytic output, in part because the kernel isn't sampled at the correct locations. The discrete convolution of Fig. 2.3b employs a continuous kernel that can be sampled wherever necessary and performs markedly better. That is to say, the continuous kernel can be evaluated at any point $y - x_i$, where we have chosen the output points y to be uniform for visualization purposes. However, the use of a continuous kernel alone is not sufficient to accurately approximate the analytic convolution. Instead, in this work, we propose modeling the kernel, g, as a continuous function and performing the calculation of discrete convolution as a quadrature approximation of Eq. (2.1). This yields the results we see in Fig. 2.3c. As we detail in Section 2.3, the method we propose includes quadrature weights ρ_i in Eq. (2.2) along with a continuous kernel $g(\cdot)$, so that our 1D equivalent would be the following:

$$(f * g)(y) \approx \sum_i \rho_i \cdot f(x_i) \cdot g(y - x_i). \tag{2.3}$$

The combination of the continuous kernel and the reduction of integration error gives us the best approximation of the operation we wish to perform. Although this example presumes a fixed continuous kernel, as opposed to a learned kernel, it demonstrates the fundamental problem of naively applying convolutional kernels to discrete data on non-uniform points.

2.2.2 Related Work

Graph convolutions Graph convolutions are perhaps the most widely used convolution method for non-uniform data, applicable when the data has a readily available adjacency structure. For a full review of the relevant methods we refer the interested reader to [117]. In general, these methods are either spectral or spatial. Spectral graph convolutions perform convolution in the Fourier domain, computing the convolution as a point-wise product of two signals, whereas spatial graph convolutions work in the spatial domain directly. Graph convolutions do not always have clear connections to standard convolution;

in particular, one cannot down-sample the input directly. Thus, graph convolutional neural networks (GCNNs) rely on pooling methods for down-sampling the number of nodes in an arbitrary graph. Significant effort has been put into graph pooling operators from a variety of perspectives [66], but note that not all of these have an associated unpooling operator. Many methods remove edges [41] or nodes [86, 32] via a (possibly trainable) score function based on graph features. Other methods create a new down-sampled graph by clustering groups of nodes [103], where the cluster assignment can be accomplished in numerous ways.

Fundamentally, since graph convolutions use only adjacency structure, they do not exploit the full knowledge of non-uniform data. For example, recall the 1D example of the previous section. In 1D, all grids (uniform or not) of the same number of nodes will have the same adjacency structure, so graph based methods could not distinguish uniform from non-uniform samples. This makes them less suitable for spatially embedded data, where the coordinates of the points can be used in addition to the graph structure. In Section 2.4 we will compare against graph convolutions using standard max-pooling by leveraging information outside of the adjacency structure. Effectively, this provides a best case scenario for GCN, and avoids employing more complicated pooling methods.

Point cloud convolutions There are numerous approaches to convolution for spatially embedded data, owing to the availability of point cloud data from LiDAR measurements [39]. Some of these methods voxelize the input data and perform convolution on the resulting voxels [116]. This often results in significant sparsity which can be taken advantage of for reasonable computational complexity. However, it eliminates the application to many non-uniform problems, if, for example, maintaining variable point density is desirable, as is commonly done to resolve boundary layer effects in fluid problems. The PointNet architecture [84] pioneered the use of shared Multi-Layer Perceptrons (MLPs) operating directly on the points themselves. Hybrid methods, such as [104], have shown great success by combining a voxelized convolution branch with a point-based branch. Other methods may perform convolution directly on the points by various methods such as utilizing MLPs to define the

convolutional kernels over all the necessary locations [10] or changing the definition of convolution to adapt to the irregular domain [118, 56]. We believe the most similar of these approaches to our work is PointConv [115], which attempts to approximate the integral via Monte Carlo integration. However, we approximate the integral via quadrature and since typical meshes are in three dimensions or less, quadrature is typically superior to Monte Carlo. In general, point cloud methods allow for spatial locations to change from sample to sample; in contrast, our method explicitly takes advantage of a static mesh.

Continuous convolutions Continuous convolution operators have been investigated for some time, motivated by a variety of use cases including non-uniform input data, arbitrary kernel sizes, and multi-resolution learning. These methods vary widely in their choice of continuous domain, how their kernel is parameterized, and how the operator is constructed. The 2017 work of [96] generates kernels for graph convolution via continuous edge labels. The use of an MLP that maps from coordinates of spatially embedded data to parameterize the kernel was introduced in [112]. This approach is also used in [89, 90, 91] to construct arbitrarily sized kernels for efficiently modeling long range dependencies. Other parameterization approaches for the kernel have also been considered. SplineCNN [28] uses a set of B-Spline basis functions. S4ND [78] constructs a global kernel via a Kronecker product of sampled one-dimensional convolution kernels created as a linear combination of hand-picked basis functions (e.g., sine and cosine). The work of [53] learns both the locations and values of a discretely sampled kernel. The kernel value at an arbitrary query point is then constructed based its local samples. During the preparation of this paper, the work of [15] was made available, which also represents the kernel as an MLP and learns it from data. The authors seek to develop an operation which resembles standard convolutions as much as possible, while still being applicable to non-uniform data. The key difference between the continuous convolution operators discussed so far and our own is the addition of quadrature to weight our finite sum.

General applications There are many applications where the density of points is

directly related to the underlying data and can be used to achieve more accurate convolution integrals than the standard convolution discretization. Non-uniform meshes are prevalent in PDE simulations, where nodes may be concentrated in areas where the PDE solution has a large gradient, such as near the boundary.

Compression of scientific data The simulation of PDEs can create immense amounts of data, which then requires compression in order to store on disk for later scientific usage. While more classical methods exist for both lossless [65] and lossy compression [64, 17] for these types of datasets, they are largely limited to uniform grids or lose efficiency when applied to non-uniform data. Compression of PDE simulation data with convolutional autoencoders, has proven to be effective on uniform grids achieving high compression ratios (in excess of 100×) with low reconstruction errors [33, 74].

Other scientific applications In addition to data compression, convolutional neural networks have been utilized in PDE super-resolution methods [101] and reduced-order modeling [75, 59]. All of these techniques could benefit from a generalization of standard convolution to the non-uniform meshes commonly found in PDE simulation data.

2.3 Methods

We will begin by discussing the construction of our discrete convolution operator, and then describe the details associated with a practical implementation. Our approach, QuadConv, will be to approximate Eq. (2.1) via quadrature, so denote the nodes as $\boldsymbol{x}_i$ and the weights as ρ_i, for $i = 1, \ldots, N$, and then consider the following approximation:

$$(f * g)(\boldsymbol{y}) \approx \sum_{i=1}^{N} \rho_i \cdot f(\boldsymbol{x}_i) \cdot g(\boldsymbol{y} - \boldsymbol{x}_i). \tag{2.4}$$

We employ the following form for our kernel function g:

$$g(\boldsymbol{z}) = \text{bump}(\boldsymbol{z}) \cdot h(\boldsymbol{z}; \boldsymbol{\theta}), \tag{2.5}$$

where $\boldsymbol{\theta}$ are learnable parameters for some function $h : \mathbb{R}^D \to \mathbb{R}$. The definition of the bump function is then given below for some $\alpha > 0$:

$$\text{bump}(\boldsymbol{z}) = \begin{cases} \exp\left(1 - \frac{1}{1-(\|\boldsymbol{z}\|/\alpha)^4}\right) & \|\boldsymbol{z}\| < \alpha \\ 0 & \text{else} \end{cases}. \tag{2.6}$$

We observe that α, which is set manually, allows us to control the support of g, and therefore the effective size of the kernel. As discussed earlier, this compact support is desirable because we seek to extract local features from the input data. It also has the important consequence of significantly reducing the computational overhead, which we will see in more detail in Section 2.3.1. When setting α in practice, we can take the same approach as standard discrete convolutional layers where 3×3 kernels are often preferred in 2-dimensions since multiple layers with smaller kernels can replicate the effect of a larger kernel size. Similarly, we prefer smaller compact supports to larger ones which saves computation and allows us the same flexibility when layers are composed. When the domain is non-uniform we set α based on the average number of points inside the compact support.

We have written equation Eq. (2.4) as a single sum based on the direct quadrature approximation of Eq. (2.1), but this can also be viewed as an iterated integral and a corresponding iterated sum. For example, in two dimensions we could let $\boldsymbol{x} = (x_1, x_2)$ and $\boldsymbol{y} = (y_1, y_2)$ to rewrite Eq. (2.4) as:

$$(f * g)(y_1, y_2) \approx \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \rho_{ij} \cdot f(x_{1_i}, x_{2_j}) \cdot g(y_1 - x_{1_i}, y_2 - x_{2_j}), \tag{2.7}$$

where we have quadrature weights ρ_{ij} for $i = 1, \ldots, N_1$ and $j = 1, \ldots, N_2$. In fact, if the points lie on a uniform grid, then using the two-point composite Newton-Cotes quadrature recovers standard convolution. Such a choice of quadrature yields weights $\rho_{ij} = 1/4$, and equates to using the trapezoidal rule along each dimension. This constant weight can then be absorbed into the learned kernel, producing the simple finite sum we see with standard convolutions.

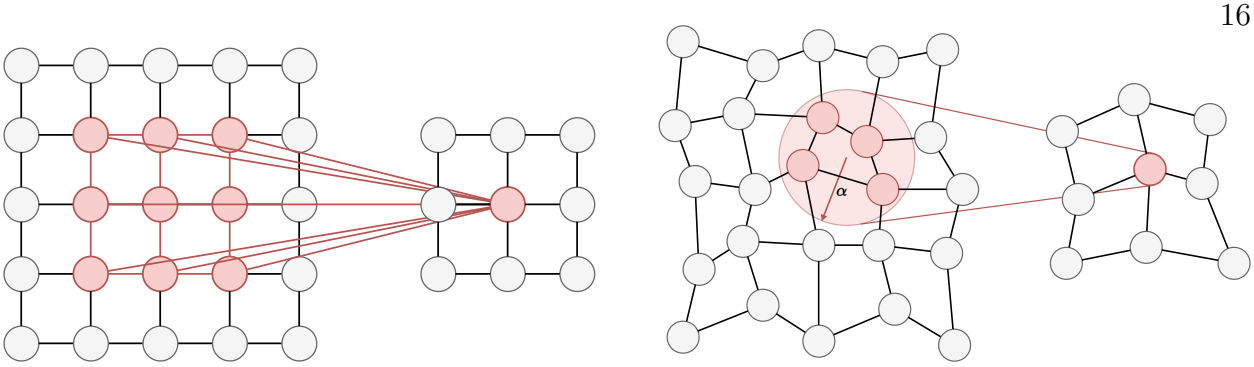
The specifics of the quadrature (i.e., the nodes and weights) is a question so far unaddressed. In general, we consider the nodes to be given as part of the input data. In other words, the quadrature nodes are fixed by the locations at which the input signal f is evaluated. In many scenarios the weights can be readily computed on the fly. For example, if the input points are on a uniform grid, then the operator may use Newton-Cotes weights. For data from FEM based simulations it is possible to construct node based quadrature weights from individual element quadrature evaluations, but this is a somewhat complicated procedure that could incur significant computational overhead and furthermore the usual quadrature construction depends on the kernel. Because we already expect to learn the kernel (i.e., $\boldsymbol{\theta}$) from data, perhaps the easiest option is to learn the quadrature weights as well. When we learn the quadrature weights we ensure they are strictly positive in order to avoid catastrophic cancellation and resulting loss of precision [76].

Figure 2.4 visualizes, in two dimensions, the difference between the standard discrete convolution and our quadrature convolution. The standard method uses a grid based kernel (3×3 in this case) that slides across the spatial dimensions of the domain to compute the output values. However, when one moves to a non-uniform mesh, this operation no longer applies. On the other hand, the quadrature method computes the output values using mesh nodes that lie within the compact support (translucent red circle) of the kernel, and so it is agnostic to the underlying mesh structure.

Analytic convolutions (Eq. (2.1)) are equivariant to continuous translations, and their standard discretization (Eq. (2.2)) inherits this property in a discrete sense. In mathematical terms, if T is some translation operator $(Tf)(\boldsymbol{x}) = f(\boldsymbol{x} - \boldsymbol{t})$, then the following holds:

$$T(f * g) = (Tf) * g = f * (Tg),$$

or, more plainly, the output of the convolution is translated equivalently to the translation of its input. In image classification, for example, this property is desirable because features should be invariant to placement in the image. Because QuadConv approximates contin-



(a) Discrete convolution with 3x3 kernel on uniform grid.

(b) QuadConv with α compact support on non-uniform mesh.

Figure 2.4: Comparison of discrete convolution and QuadConv.

uous convolution, it is approximately equivariant, up to the accuracy of the quadrature. As features move through the domain, the output of the QuadConv layers will vary by approximately the same amount as the original data.

So far, we have only considered scalar valued inputs to the convolution operator, but it is necessary to extend this to a multi-channel (i.e., vector) setting. Standard deep learning convolutions accomplish this by stacking multiple kernels into a filter, and then applying multiple filters to the input data. We will adopt a similar approach. Define $\boldsymbol{G} : \mathbb{R}^D \to \mathbb{R}^{\tilde{C}\times C}$ as a map from a point to a matrix, which we interpret as a stack of filters. Despite the incongruence in vocabulary, we will refer to $\boldsymbol{G}$ as a filter. Now, we consider $\boldsymbol{f} : \mathbb{R}^D \to \mathbb{R}^C$ as our vector valued input, and the quadrature convolution is given as:

$$(\boldsymbol{f} * \boldsymbol{G})(\boldsymbol{y}) = \sum_{i=1}^{N} \rho_i \cdot \boldsymbol{G}(\boldsymbol{y} - \boldsymbol{x}_i) \cdot \boldsymbol{f}(\boldsymbol{x}_i), \tag{2.8}$$

where we can easily observe that the output lies in $\mathbb{R}^{\tilde{C}}$. The filter $\boldsymbol{G}$ is parameterized by $\boldsymbol{\theta}$ according to the multi-channel extension of Eq. (2.5):

$$\boldsymbol{G}(\boldsymbol{z}) = \text{bump}(\boldsymbol{z}) \cdot \boldsymbol{H}(\boldsymbol{z}; \boldsymbol{\theta}), \tag{2.9}$$

where $\boldsymbol{H}(\cdot; \boldsymbol{\theta}) : \mathbb{R}^D \to \mathbb{R}^{\tilde{C}\times C}$. The form of the bump function remains unchanged, and thus we see that $\boldsymbol{G}$ may be close to $\boldsymbol{0}$ depending on the location it is evaluated at. We may also

note that the dimension, D, of the data was arbitrary, so our operator can easily be applied to data in any dimension.

2.3.1 Practical Computation

The methodology we have developed so far is the foundation of our proposed convolution operator, but certain aspects of the implementation are as of yet unclear. Further, specific elements of the computation must be leveraged to achieve any reasonable level of efficiency. We will discuss the practical implementation of the quadrature-based convolution operator, and consider its asymptotic computational complexity.

In general, we consider input in the form of a tuple $(\boldsymbol{X}, \boldsymbol{F})$, where $\boldsymbol{X} \in \mathbb{R}^{D\times N}$ represents the mesh consisting of N points in D-dimensional space, and $\boldsymbol{F} \in \mathbb{R}^{C\times N}$ are the associated features with C channels. For example, a 3D flow field would have three channels; one each for the velocity along each dimension. For a training dataset with T training samples, the input will be $((\boldsymbol{X}, \boldsymbol{F}^t))_{t=1}^{T}$, meaning that all samples share the same mesh. For a practical implementation of the current version of QuadConv we require a single fixed input mesh. If we were to allow the mesh to vary, it would significantly increase the cost of learning quadrature weights for each node, and negate the speedups obtained via caching, which is discussed later in this section.

The QuadConv operator, which we will denote as $\boldsymbol{Q}$, acts in the following manner:

$$(\boldsymbol{X}, \boldsymbol{F}) \xrightarrow{\boldsymbol{Q}} (\boldsymbol{Y}, \tilde{\boldsymbol{F}}),$$

where $\boldsymbol{Y} \in \mathbb{R}^{D\times\tilde{N}}$ are $\tilde{N}$ output points, and $\tilde{\boldsymbol{F}} \in \mathbb{R}^{\tilde{C}\times\tilde{N}}$ are the recovered features with $\tilde{C}$ channels. As with standard convolutions, there are a number of architectural decisions that must be made manually by the user, and each comes with a variety of trade-offs. Similar to standard convolutions, the practitioner is left to specify the number of output channels. Unlike standard convolutions, where the number of output points is determined as a combination of various hyperparameters such as stride and padding, the number of output

points and their coordinates are directly specified when using QuadConv. Due to this, our method easily facilitates up-sampling or down-sampling, i.e., increasing or decreasing the number of points in the domain, and can maintain non-uniform point density. This is due to the continuous kernel being defined at all points inside the domain which maximizes our re-sampling flexibility — this means there are no inherent constraints on the output points of a QuadConv layer. Thus the process by which the output point coordinates are computed is an important decision factor. In the simplest case, assuming the geometry is Cartesian, they can be placed on a uniform grid. A more sophisticated strategy would involve coarsening the underlying mesh via agglomeration, e.g., [13], to construct the output points directly from the input points. The actual computation by $\boldsymbol{Q}$ via Eq. (2.8) is given below:

$$\boldsymbol{Q}[\boldsymbol{X}, \boldsymbol{F}]_j = (\boldsymbol{Y}, \tilde{\boldsymbol{F}})_j = \big(\boldsymbol{y}_j, \sum_{i=1}^{N} \rho_i \cdot \boldsymbol{G}(\boldsymbol{y}_j - \boldsymbol{x}_i) \cdot \boldsymbol{f}_i\big), \tag{2.10}$$

for $j = 1, \ldots, \tilde{N}$. Note that $\boldsymbol{y}_j$ and $\boldsymbol{f}_i$ are the j^{th} and i^{th} columns of $\boldsymbol{Y}$ and $\boldsymbol{F}$, respectively. Recall from Eq. (2.9) that $\boldsymbol{H}$ provides us with the learnable map from a point to a matrix. Our method, in general, is agnostic to the form of this map, but for the sake of efficiency we opt for a single MLP, $\mathbf{G} : \mathbb{R}^{\mathbf{D}} \to \mathbb{R}^{\tilde{\mathbf{C}} \times \mathbf{C}}$. There are many other possible choices, such as using multiple MLPs which are shared among input/output channels or are entirely independent.

As noted earlier, depending on their evaluation location, the approximation $\boldsymbol{G} \approx \mathbf{0}$ may hold for many of the filters, allowing the implementation to take advantage of the sparsity of the sum in Eq. (2.8) in order to avoid undue computation. The set of index pairs i and j for which the function bump$(\boldsymbol{y}_j - \boldsymbol{x}_i)$ is nonzero generates a map of the form $j \mapsto \{i_k\}$, meaning that for each output location index j, there is a set of input indices $\{i_k\}$ that contribute to its value. Figure 2.5 visualizes how this map can be used to implement Eq. (2.10).

This entire process can be executed efficiently in a vectorized manner as the individual operations are independent, and for any given output index j the sum that determines its features is over far fewer indices i than would otherwise be the case, just as in the standard discrete case. Assuming that the input and output locations are static, then the maps

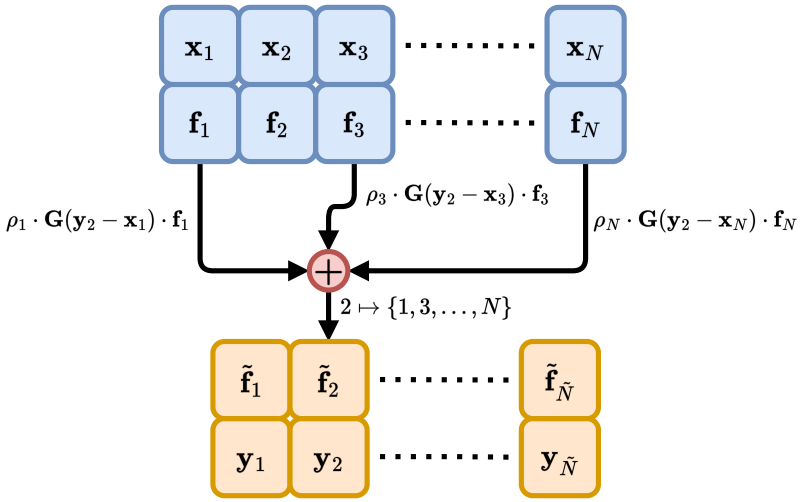


Figure 2.5: QuadConv computation. The output value at index $j = 2$ depends only sparsely on the input values, e.g., there is no dependence on the input value at index $i = 2$.

$j \mapsto \{i_k\}$ can be computed once as a pre-processing step and cached for future use. Doing so avoids the costly construction of the maps themselves, which significantly improves the execution time of the convolution operator.

Table 2.1 shows the computational complexity for the forward pass of QuadConv compared to standard discrete convolutions. For QuadConv, we split the complexity into the construction step for the index map, which needs only be computed once for the entire dataset and hence is amortized, and the actual quadrature convolution computation. This helps to highlight the impact of caching on the overall performance. During training, there is also cost due to the backward pass in automatic differentiation, which we assume is on the same scale as the cost of forward pass.

The variables $\{C, N\}_{in}$ and $\{C, N\}_{out}$ are the input and output channels and points, respectively. We consider a standard convolution with kernel size K, and for QuadConv, we consider a sparsity factor S, the average number of nodes inside the support of the kernel $\boldsymbol{G}$, which is dependent on the radius α of the bump function and the local point density. The variables M_t and M_m denote the time and memory complexity for the chosen filter operation (i.e., MLP). Note that M_m has an implicit weak dependence on C_{in} and C_{out}, as

the output of the filter operation is some $C_{in} \times C_{out}$ matrix. When considering the peak

	Standard	**QuadConv**	
		Convolution	**Index Map**
Time	$KN_{out}C_{in}C_{out}$	$SN_{out}C_{in}C_{out} + M_t SN_{out}$	$N_{in}N_{out}$
	KNC^2	$SNC^2 + M_t SN$	N^2
Peak Memory	$KC_{in}C_{out}$	$M_m + C_{in}C_{out}$	SN_{out}
	KC^2	$M_m + C^2$	SN

Table 2.1: Big-$\mathcal{O}$ comparison of standard discrete convolutions and QuadConv. The second row of the time and memory blocks takes $N_{in} = N_{out} = \mathcal{O}(N)$, and $C_{in} = C_{out} = \mathcal{O}(C)$ in order to facilitate comparisons. The index map only needs to be computed once for a given mesh.

memory consumption, we do not include the requirements for the input or output data, as that is common to all three methods.

One may observe a number of important details from Table 2.1. With respect to QuadConv, we see that sparsity is extremely important. Just as in standard discrete convolution the local support, or sparsity, of the operator reduces the complexity to SN_{out} for a single input/output channel, where typically $S \lesssim 10$, down from $N_{in}N_{out}$ without exploiting sparsity. From the last column, we can see that because the index map construction must look at all input-output point pairs, it is an $\mathcal{O}(N^2)$ operation — producing the SN_{out} indices needed for the forward pass of QuadConv. This is the fundamental trade-off: caching as a pre-processing step saves significant computation time in return for a larger memory footprint.

2.4 Numerical Experiments

We have implemented QuadConv using PyTorch [82] and Lightning [25], available as an open source Github repository [19]. The code and datasets directly associated with this paper, and capable of recreating our results, can be found in [20]. This section shows a

number of experiments that validate our method of convolution as an effective tool in deep learning. In particular, we perform data compression of PDE simulation data using an autoencoding neural network. First, we start with data on a uniform grid (Section 2.4.1) so that we can establish a baseline of performance by comparing our method to standard discrete convolution. We then interpolate this dataset onto a non-uniform mesh and show that QuadConv still performs just as accurately as with the uniform mesh (Section 2.4.2). Finally, we consider a different dataset which was simulated on a non-uniform mesh, and show that our operator continues to facilitate high reconstruction accuracy at large levels of compression (Section 2.4.3).

Figure 2.6 provides a high-level visualization of our neural network architecture. The raw input data is passed through the encoder, which is comprised of a (Quadrature) Convolutional Neural Network ((Q)CNN) that progressively down-samples the number of points and increases the number of channels, and an MLP which outputs the latent representation $\boldsymbol{z}$. This latent vector is the compressed state of the data. To reconstruct, one simply performs the reverse operation using the decoder. An MLP is applied to the latent state $\boldsymbol{z}$, and then a (Q)CNN up-samples the data.

We use a randomized non-contiguous 80/20 split of our data into a training and testing sets. For data compression where all of the data is available at the time of compression, all of the data is available for training and generalization is not important. In more typical machine learning applications the generalization of the model is the most important metric. We evaluate our metrics over the entire dataset (both testing and training), and we also include the maximum error over the entire dataset to capture the generalization of the operators.

Our autoencoder aims to minimize the empirical risk of a regularized mean-squared error loss between $\boldsymbol{F}^t$ and $\tilde{\boldsymbol{F}}^t$,

$$\mathcal{L}(\boldsymbol{F}^t, \tilde{\boldsymbol{F}}^t) = \|\boldsymbol{F}^t - \tilde{\boldsymbol{F}}^t\|_{\mathrm{HS}}^2 + \lambda R(\boldsymbol{F}^t, \tilde{\boldsymbol{F}}^t), \tag{2.11}$$

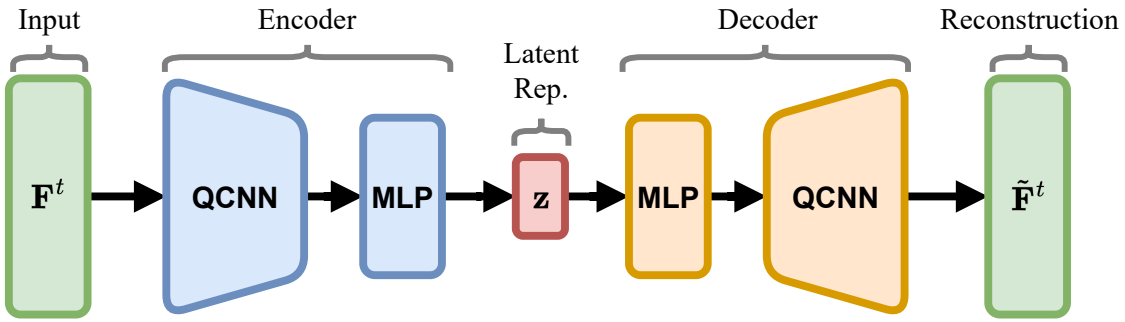


Figure 2.6: Autoencoder structure.

where $t = 1, \ldots, T$ indexes the time dimension of the (time-dependent) PDE simulation and $\|\boldsymbol{F}^t\|_{\text{HS}} = \left(\sum_{ij} |\boldsymbol{F}^t_{ij}|^2\right)^{1/2}$ is the Hilbert-Schmidt (aka Frobenius) norm. Our regularization term $R(\boldsymbol{F}^t, \tilde{\boldsymbol{F}}^t)$ in the uniform grid case is the mean-squared error between finite difference derivatives of $\boldsymbol{F}^t$ and $\tilde{\boldsymbol{F}}^t$ (i.e., the Sobolev norm). In our non-uniform examples the derivatives are more difficult to define so we do not use any regularization. The final error we report is the time-averaged relative Frobenius-norm,

$$\frac{1}{T}\sum_{t=1}^{T} \frac{\|\tilde{\boldsymbol{F}}^t - \boldsymbol{F}^t\|_{\text{HS}}}{\|\boldsymbol{F}^t\|_{\text{HS}}}. \tag{2.12}$$

Our data is represented with 32-bit floating point numbers (single precision) [47], and all training and evaluation was performed with single precision numbers. For the purposes of this discussion, we define the compression ratio as the ratio between the dimension of the raw and compressed data. Since we only compress the data in spatial dimensions, this corresponds to the size reduction of any individual spatial sample. For example, raw data on a 10×10 grid compressed to a latent space in $\mathbb{R}^{10}$ would correspond to a compression ratio of 10. We do not consider the storage of the decoder itself, as this has negligible contribution to the compression ratio as $T \to \infty$; as we'll show in Table 2.3, storage of the encoder and decoder together is just a few megabytes.

All experiments were conducted on 4 V100 GPUs and 1 IBM Power9 CPU with the Adam [54] optimizer. Nearly all experiments used a batch size of 8 under a distributed data parallel training strategy, except for the SplineCNN networks which use a batch size of 1.

The SplineCNN steps are noted with an asterisk in each of the tables that follow but the other experiments each correspond to the same amount of data processed per step.

2.4.1 Uniform Grid Ignition Data

This dataset consists of 450 uniformly sampled time steps of a jet ignition simulation that lie on a uniform spatial grid of size 50×50. The wavefront is fully resolved in time. This dataset is transport dominated for the initial portion of the simulation until the jet flame reaches steady state. Figure 2.7, with a latent state $\boldsymbol{z} \in \mathbb{R}^{50}$, describes the model architecture used on the dataset in question — resulting in a compression ratio of 50. Because the input points are already on a uniform grid, we can use the two-point composite Newton-Cotes quadrature [3] weights for the QuadConv layers. This matches the quadrature interpretation of the CNN which gives a very direct comparison of the QuadConv layer to the standard convolution layer. We refer to that QuadConv experiment as "Static Weights" in Table 2.2. We also perform the same experiment with quadrature weights as parameters of the neural network, with the hope that the training process can learn a quadrature which is more tailored to the dataset (the "Learned Weights" experiment). In our neural network, the max pooling layer handles the downsampling, so the input and output points for the QuadConv layers are identical.

In addition we compare to a few alternatives: a Proper Orthogonal Decomposition (POD) over 50 basis vectors [8], a graph convolutional network (GCN), and a SplineCNN [28] network. We choose to use the particular graph convolution operation presented in [55],

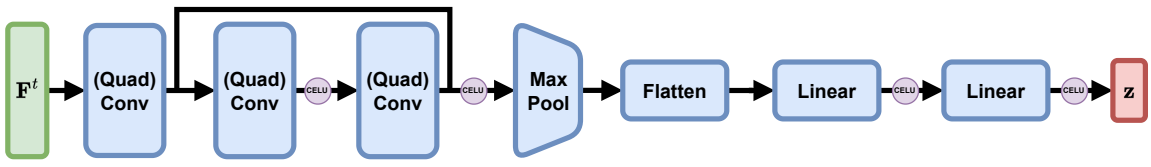


Figure 2.7: Max-pooling based encoder structure. The linear layers comprise the MLP. The decoder is similarly structured, but mirrored to up-sample.

which is an approximation of the full spectral graph convolution operator. In practice, many implementations of graph neural networks exhibit instabilities and large maximum errors, the latter of which we observe in the Table 2.2. SplineCNN utilizes a continuous kernel which operates on the attributes of the edges given in pseudo-coordinates as well as the adjacency structure of the data. In both examples we circumvent the complexity of graph pooling operators and make the comparison more direct by using the max pooling operator as defined on a grid as the pooling operator. Both the GCN and SplineCNN share the same structure as Figures 2.6 and 2.7, also with $\boldsymbol{z} \in \mathbb{R}^{50}$.

Model Type	Average Error	Max Error	Training Time (h)	# of Steps	# of Trainable Parameters
POD (50 Basis Vectors)	1.37%	17.3%	N/A	N/A	N/A
Conv	0.42%	10.7%	1.46	186,000	1,024,718
GraphConv	0.52%	9.1%	2.64	315,360	1,016,206
SplineCNN	0.65%	10.7%	13.85	2,728,080*	1,025,710
QuadConv (Static Weights)	0.63%	**2.3%**	3.91	186,000	1,035,310
QuadConv (Learned Weights)	0.49%	**1.2%**	9.25	290,243	1,040,310

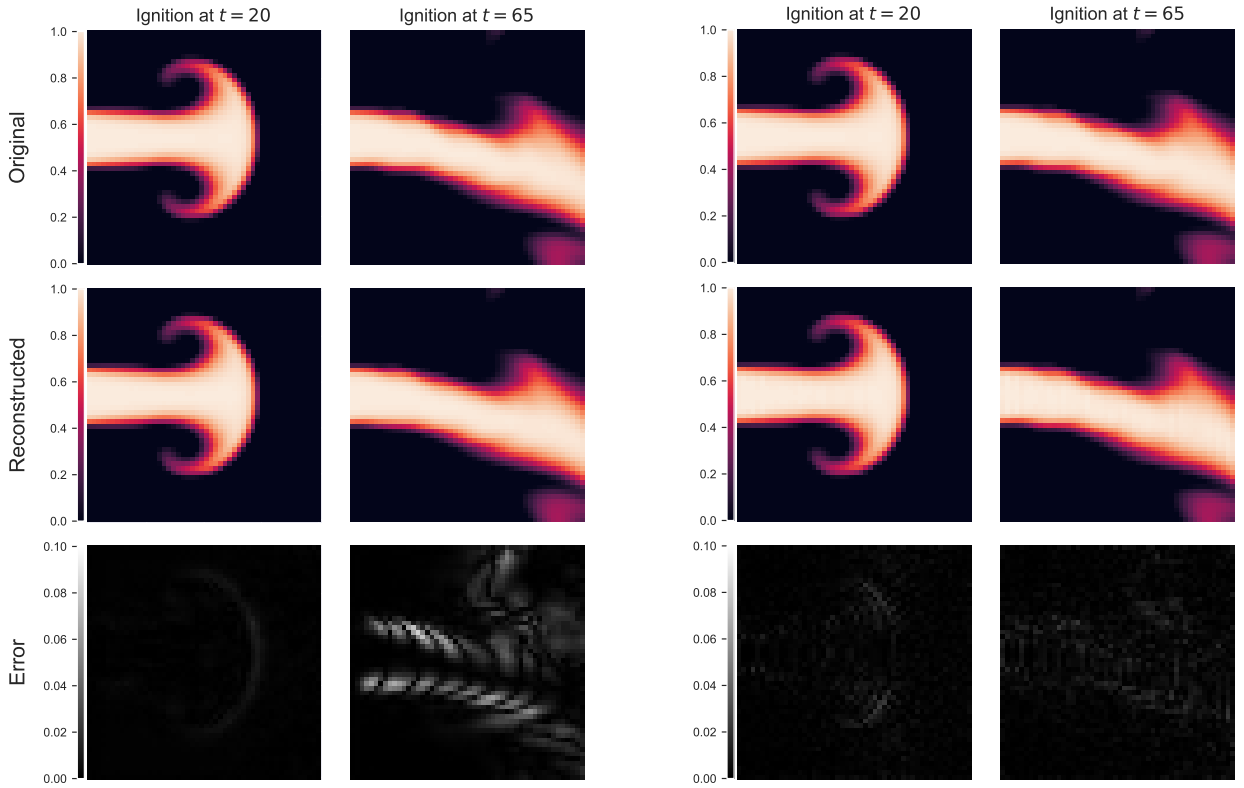
Table 2.2: Autoencoder results for uniform grid ignition data compression at 50× compression.

	Conv	GraphConv	SplineCNN	QuadConv (Static)	QuadConv (Learned Weights)
Model Size (MB)	3.98	3.95	4.88	4.08 — 13.73 (Cached)	4.08 — 13.73 (Cached)
Inference Time (ms)	2.83	8.98	4.86	8.63	8.55

Table 2.3: Inference time and model size for all autoencoders on a uniform grid.

We can see in Table 2.2 that the CNN and QCNN indeed have very similar average performance, although the QCNNs take longer to train. Learning the quadrature weights provides a slight advantage for the average reconstruction error and a large improvement in the maximum error, though it comes at the cost of a longer training time to converge relative to the static quadrature. We see that the GCN has competitive average reconstruction error but the worst maximum error. While increased training time when using QuadConv is to be expected, at roughly 2.7× for the static quadrature, it is not prohibitively more costly. Furthermore, the implementation of standard discrete convolution in PyTorch [82]

is exceptionally well optimized, while QuadConv has room for improvement. Even at these relatively high compression ratios, we see few readily visible errors in the data in Fig. 2.8 at selected time points. The error visualization at the bottom compresses the color scale by an order of magnitude in order to better expose the error that is present.



(a) CNN compression.

(b) QCNN compression.

Figure 2.8: Comparison of CNN and QCNN ignition data reconstruction. Note that the bottom row has a rescaled color bar.

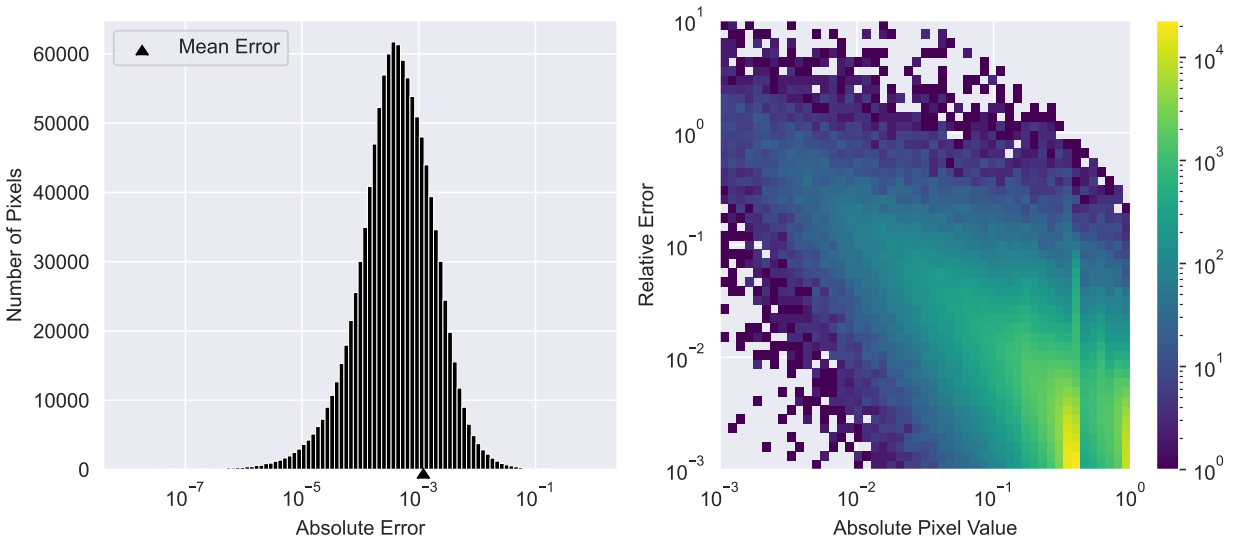


Figure 2.9: Error analysis of CNN for the uniform grid ignition data.

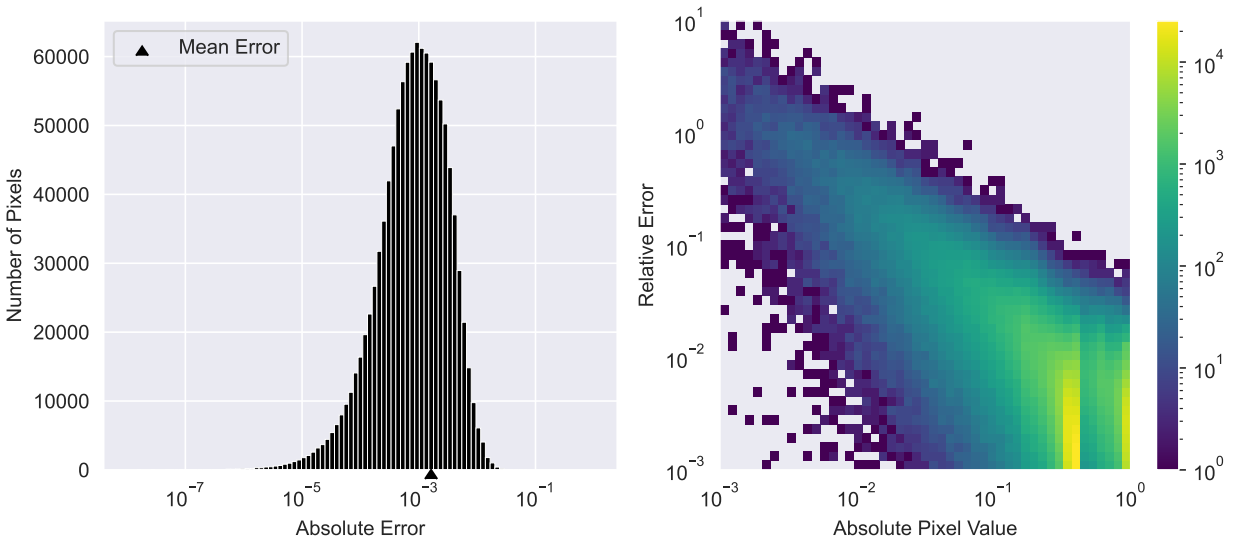


Figure 2.10: Error analysis of QCNN with learned quadrature weights for the uniform grid ignition data.

We conduct a further error analysis to understand relative performance in Figs. 2.9 and 2.10, which more clearly shows the similarity of performance between the CNN and the QCNN. This shows the distribution of the absolute errors are similar, as well as the relative

error histogram, which helps to identify at what absolute pixel values large relative errors occur. The GCN Fig. A.1 and SplineCNN Fig. A.2 have a similar mean absolute error to the CNN and the QCNN but the relative error histograms reveals the larger maximum errors, which can be found in the appendix.

2.4.2 Non-Uniform Mesh Ignition Data

To maintain a benchmark of performance we interpolated the ignition data with 2D splines and evaluated the interpolant on the non-uniform mesh shown in Fig. 2.11a. This mesh contains 2189 points, more concentrated in the middle and the right-side of the domain, and was generated with `dmsh` [94]. In order to evaluate the error introduced during the interpolation and subsequent re-sampling on the non-uniform mesh, we re-interpolated the data from the non-uniform mesh and compared it to the original data from the uniform grid. Using the same relative Frobenius-error averaged over all data samples, we found the interpolation process introduced 0.122% error into the data.

For this experiment, we continue to use a latent state $\boldsymbol{z} \in \mathbb{R}^{50}$, but our model varies slightly from Section 2.4.1 in the first and last QuadConv layers, although Fig. 2.7 still describes the general structure. Because the input data is on a non-uniform mesh, the first QuadConv layer learns the quadrature weights and re-samples to a uniform grid. Doing this also allows us to use the max pooling operation and its adjoint to up-sample and down-sample, respectively. It should be noted that after re-sampling the data to a uniform grid and before sampling back to a non-uniform mesh, one could employ standard 2D convolutions. This is certainly a valid approach, but in this work we are seeking to validate the QuadConv operator itself, and not necessarily trying to optimize the architecture. The rest of the model remains unchanged until the last QuadConv layer in the decoder, where the data is sampled from the uniform grid back to the original non-uniform mesh.

The GCN and SplineCNN in this experiment must similarly adapt their pooling layer to be well-defined on the non-uniform data. We use the KNN pooling operator found in [27]

to interpolate the data from the mesh (after a graph convolution operation) back to a grid. This allows us to use the max pooling operator as defined on a grid just as in the QCNN. We use the KNN unpooling operator in [27] to then move back to the mesh in the second to last layer in the network. While there are other graph pooling operations, this is what we believe to be the most direct comparison possible with readily available software.

We will also compare against two other methods for dealing with the non-uniformity of the mesh data. First, we consider a voxel model, which is identical to the standard convolutional model used in Section 2.4.1 except that the input data is voxelized. That is to say, before the input data is passed through the network, it is aggregated onto a uniform grid. The inverse of this process, de-voxelization, is then applied after processing and before computing evaluation metrics. The second model we consider is a point-voxel model. This pairs the voxel model with a point-based branch where shared MLPs are applied directly to the point features themselves. See [84] and [104] for more details. As we can see in

Model Type	Average Error	Max Error	Training Time (h)	# of Steps	# of Trainable Parameters
Voxel	2.789%	5.18%	0.66	120,000	1,015,222
Point-Voxel	0.956%	13.41%	1.17	120,000	1,021,163
GraphConv	0.730%	21.14%	2.98	450,000	1,016,206
SplineCNN	0.597%	8.54%	19.34	3,600,000*	1,025,710
QuadConv	0.596%	**2.05%**	13.50	450,000	1,037,499

Table 2.4: Results for non-uniform ignition data compression.

Table 2.4 and Fig. 2.11b, the QCNN produces similar accuracy as on the uniform grid. This suggests that our approximation is performing well over the non-uniform data found in the center of the domain, since we do not see an increase in error due to the change in the underlying structure of the data. Error histograms are similar to those in the uniform grid case and can be found in A. While the voxel model trains quite quickly, we can see that it is by far the worst performing method. This should not be altogether surprising, as the de-voxelization process introduces a hard lower bound on the reconstruction accuracy

because mesh points within the same grid cell will necessarily have the same features. On the other hand, the point-voxel model is able to achieve comparable average error to QuadConv, but its maximum error is quite high to due its poor generalization to the test data. This speaks to QuadConv's ability to match and even exceed other state-of-the-art methods for operating on non-uniform data. Importantly, we note that the shared MLPs of the point branch place a limit on how small a point-voxel network can be. Thus, such a method may not be suitable for data compression in a general setting. Finally, the GCN exhibits the same high maximum error as in the previous test case on the grid while performing well on average. While the SplineCNN is the best of all previous methods, its maximum error is still 6% higher than QuadConv.

2.4.3 Non-Uniform Mesh Flow Data

While the ignition examples are informative for verifying the performance of the QuadConv layer in comparison to standard convolution, the non-uniform mesh is not native to the data. We consider here an example of fluid flow around a cylinder at Reynolds number $Re = 100$ in a channel. This PDE simulation was conducted on the non-uniform mesh shown in Fig. 2.12. This dataset has 300 time points and the streamwise velocity at 7613 nodes in the domain.

This data is not as transport dominated as the jet ignition data, and, as a result, it is less challenging to compress. Our model is also substantially changed from the previous experiments. While we have thus far performed a $50\times$ reduction in the data, this example has a latent state $\boldsymbol{z} \in \mathbb{R}^{15}$, which equates to a $500\times$ reduction. This increase in data reduction is due to a different linear layer preceding the latent space, but it still requires effective feature extraction inside the QuadConv layers. Because of the non-Cartesian nature of the data, we no longer use the approach of sampling to a grid and applying max pooling layers. Instead, we use the QuadConv layers to down-sample, where a specified a number of the output points are sampled from the input points by sampling uniformly at random.

The encoder structure is described in Fig. 2.13. The random sampling technique is feasible in this scenario because the point density is fairly constant, but mostly we use it as a tool to show that other agglomeration procedures can be used for selecting non-uniform output points.

Model Type	Average Error	Max Error	Training Time (h)	# of Steps	# of Trainable Parameters
QuadConv	0.762%	2.24%	7.76	80,679	807,304

Table 2.5: Results for non-uniform flow data compression.

As shown in Table 2.5 and Fig. 2.14, the flow data compresses well, with low error and converges considerably faster than the ignition data. This example shows that practical meshes are easily used, and high accuracy can be achieved with a QuadConv based network. The error histograms can be found in A.

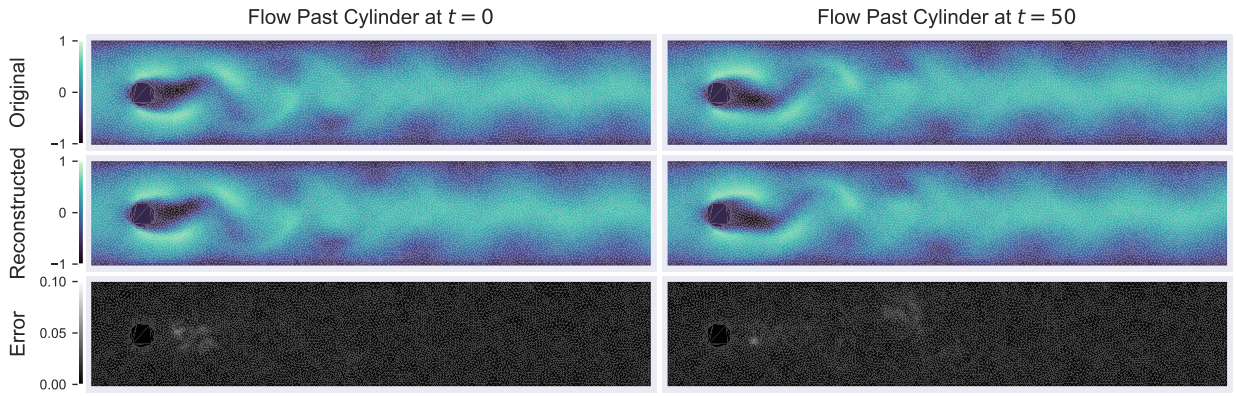


Figure 2.14: QCNN non-uniform flow compression.

From Fig. 2.14, we observe very little degradation in the quality of the solution and no visible changes in the error. In fact, since the linear layers in the network dominate the total number of trainable parameters, our embedding of the data into a smaller space reduces the overall number of parameters in the network and slightly reduces training time.

2.5 Discussion

In this work we have presented a new convolution operator for deep learning applications involving non-uniform data. Our operator approximates a continuous convolution via quadrature, and employs a learned continuous kernel to allow for arbitrary discretizations of the input data. In addition, we discussed an implementation of this operator that yields sufficiently reasonable time complexity for use in modern deep learning settings. Our experiments on both uniform and non-uniform PDE simulation data compression showed that, in practice, this quadrature convolution can match the performance of standard convolutions on uniform grid based data, and performs equally as well on non-uniform mesh data.

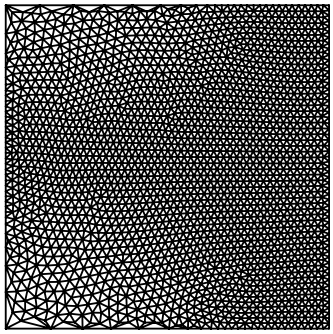
2.5.1 Future Work

The results we have presented here demonstrate the promising performance of the QuadConv layer and QCNN in application, but there are still a number of interesting questions to address as to its performance and sensitivity to hyper-parameters. As discussed in Section 2.3.1, there are a number of ways to parameterize the map from point to filter, and we have only considered one such method here. A more thorough investigation into the alternative approaches is warranted. Also discussed in Section 2.3, and put to use in Section 2.4, is the learning of the quadrature weights for a given set of input points. The accuracy of this learned quadrature itself can be measured separately using known data and kernel function. Understanding this may inspire regularization of the quadrature weights that could enable learning higher accuracy approximations. Mentioned briefly in Section 2.3.1 was the idea of constructing the output points of a QuadConv layer via an agglomeration of the input mesh. This would allow one to maintain properties of the original mesh while still down-sampling. Such a method is more extensible than the random down-sampling used in Section 2.4.3, and would likely lead to better performance for QuadConv on meshes with more complex non-Cartesian geometries. Moreover, our continuous kernel allows us to learn representations of

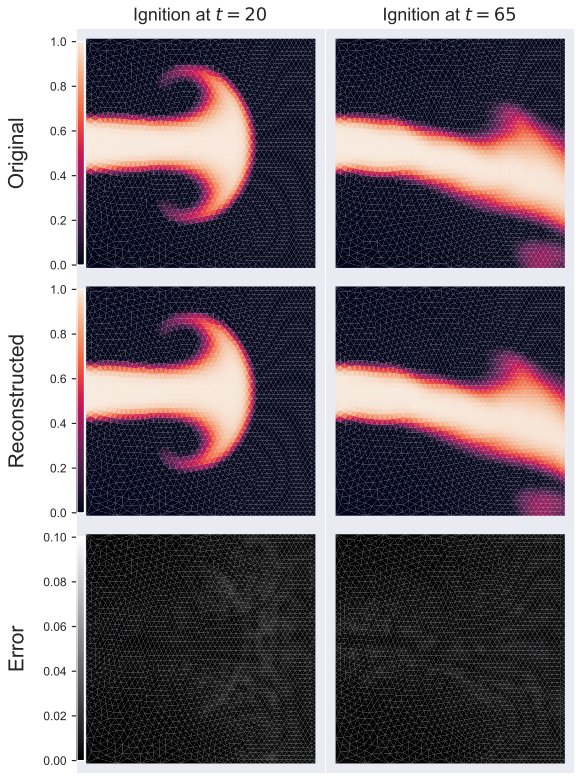
data on a continuous space which enables one to perform multi-level training. This would involve training on different resolutions of data using the same network but with variable quadrature weights in order to continue to approximate the convolution operator well while using the same kernels. Last but not least, developing online, parallel learning schemes to enable pass-efficient, ideally *in situ*, data compression, as in [24, 81], is an important future research direction.

Acknowledgments

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003968 as well as Department of Energy Advanced Scientific Computing Research Awards DE-SC0022283 and DE-SC0023346. AD's work has also been partially supported by AFOSR grant FA9550-20-1-0138. We would also like to thank Kenneth Jansen, John Evans, and Jeff Hadley from the University of Colorado Boulder for their helpful discussions surrounding this work.



(a) Non-uniform ignition mesh.



(b) QCNN compression.

Figure 2.11: QCNN non-uniform ignition data compression.

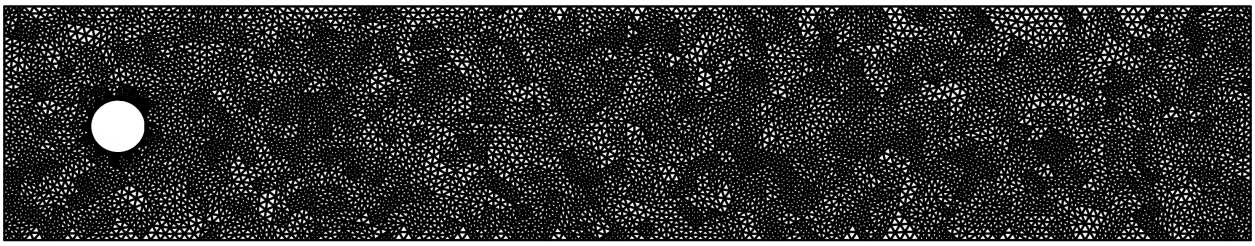


Figure 2.12: Non-uniform flow mesh.

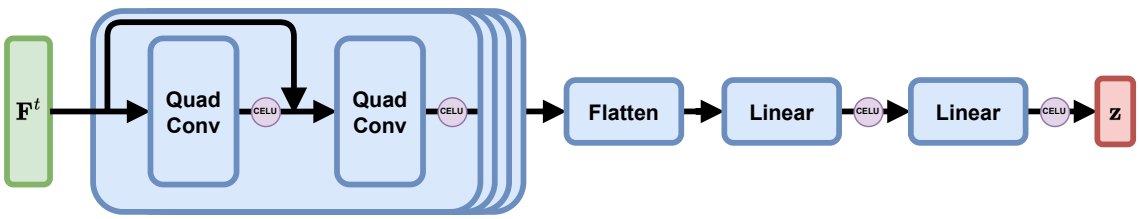


Figure 2.13: Encoder with four QuadConv based blocks. The linear layers comprise the MLP. The decoder is similarly structured, but mirrored to up-sample.

Chapter 3

Refining Quadrature Convolutions

3.1 Introduction

In Section 2.2 we introduced a new neural network layer called QuadConv that aims to generalize convolutional layers to non-uniform meshes by using continuous convolution as a template. QuadConv exhibits low error in comparison to both standard convolution on uniform grids and extends this performance to non-uniform meshes in our example data compression tasks. However there are many aspects of of QuadConv that are either not explored in Chapter 1, or do not compete well with standard convolutions and graph convolutions. This chapter examines some of these shortcomings and makes improvements to QuadConv that make it a more usable and performant neural network layer.

We divide our improvements into a few sections: stability, speed, scaling and ecosystem. The stability of a neural network layer is concerned with its initialization and convergence rate (as measured in number of epochs until convergence). Smart initialization helps with training deep neural networks that have similar operating conditions in each layer. The speed of a neural network layer is its wall clock time for forward and backward propagation. Improvements to speed and convergence rate can make a neural network substantially faster to train. Scaling is the ability to vary the parameter count of a neural network and as a consequence vary the accuracy of a neural network. In a standard convolutional network, this takes the form of varying the number of channels in each layer. Finally, there is the ecosystem of the neural network layer, which consists of the software tools and mathematical

tools that make neural networks easier to construct and test. For a neural network layer to be adopted, these components must all be sufficiently addressed to make application to real world problems practical. We draw upon similar work for standard convolution and graph convolution for motivation and context.

3.1.1 Notation

The notation surrounding neural networks can be dense, and our work focuses on how differences in the data domain effects standard practices, which further complicates notation. Here we establish some common terms to make the following sections easier. The layers in a neural network are each functions which map the layer input to the layer output, we refer to these as the domain and range of the layer. When concerned with spatial data, the dimensions of the points in the domain and range are frequently either $d = 2$ or $d = 3$. The domain ($\mathcal{X}$) and range ($\mathcal{Y}$) are collections of points in $\mathbb{R}^d$. We consider the data to be given by a function $\boldsymbol{f}_k^i(\cdot) : \mathcal{X} \to \mathbb{R}$.

Notation	
Spatial Dimension	d
Domain element	$\boldsymbol{x}_n \in \mathbb{R}^d$
Domain	$\mathcal{X} = \{\boldsymbol{x}_n : n \in \{1, \ldots, N\}\}$
Range element	$\boldsymbol{y}_m \in \mathbb{R}^d$
Range	$\mathcal{Y} = \{\boldsymbol{y}_m : m \in \{1, \ldots, M\}\}$
Input Channels	$k \in \{1, \ldots, K\}$
Output Channels	$j \in \{1, \ldots, J\}$
Data	$\boldsymbol{f}_k(\boldsymbol{x}_n) \in \mathbb{R}$
Data at Layer i	$\boldsymbol{f}_k^i(\boldsymbol{x}_n) \in \mathbb{R}$
QuadConv layer	$Q(\cdot; \boldsymbol{\theta})$

3.1.2 Motivation and Related Work

We address related work and motivation together, since the improvements we hope to make to QuadConv are the things that have made standard convolutions and graph convolutions successful.

Stability The first standard convolutional neural networks (CNNs) to use back-propagation in 1986 [92] did not differ in many ways from their more successful counterparts in the early 2000's. Glorot and Bengio spectulate in their 2010 paper [34] that initialization played a big role in this advancement, and they explored what is now termed Glorot (or Xaviar) initialization. They observed the output of different layers of a network had different distributions, and that as a result the training could become slow when the distributions of the output of the networks became extreme. The initialization strategy that Glorot and Bengio pioneered generalizes to a simple goal that we attempt to replicate for QuadConv.

Given an input $\boldsymbol{f}_k(\mathcal{X}) \in \mathbb{R}^N$, whose entries are i.i.d. drawn from a normal distribution, $\boldsymbol{f}_k(\boldsymbol{x}_n) \sim \mathcal{N}(0,1)$, and a neural network layer parameterized by $\boldsymbol{\theta}$ whose action on $\boldsymbol{f}_k(\mathcal{X})$ is given by $Q(\boldsymbol{f}_k(\mathcal{X}); \boldsymbol{\theta})$, we will aim to initialize $\boldsymbol{\theta}$ such that $Q(\boldsymbol{f}_k(\boldsymbol{x}_n); \boldsymbol{\theta}) \sim \mathcal{N}(0,1)$. This creates a cycle under the action of $Q(\cdot; \boldsymbol{\theta})$, such that successive application of different layers that have this same property maintain the stability of the output distribution.

Speed The ability to train CNNs at scale has been a fundamental driver of their adoption and impact. In 2009, Raina et al. [85] first proposed and demonstrated training CNNs on graphical processing units (GPUs). Since then large scale software projects Tensorflow [1] and PyTorch [82] have made accelerated CNN training accessible to researchers and practitioners. As we saw in Table 2.2, while QuadConv exhibits better errors than a CNN, it takes nearly 8 hours of more wall time to converge, which equates to $4\times$ more time per epoch. The graph convolutional network that we compare to is $3.8\times$ faster per epoch, owing to the significant software development focused on accelerating them [27]. In fact, despite graph convolutional networks being introduced in 2016 in a paper by Kipf and Welling [55], they did not gain significant adoption until 2018 [108], when PyTorch-Geometric made graph convolution faster and more accessible [27]. For many applications this large difference in execution time makes QuadConv infeasible. In order to spur adoption, we address the road blocks that prevent QuadConv from having competitive execution time.

Performance Scaling In Chapter 1, we compared QuadConv to many different convolutional architectures and to facilitate a fair comparison each of the networks we compared to had similar numbers of parameters. Yet missing from this analysis is a study of how QuadConv's performance scales when the number of parameters is increased. CNNs have two standard methods to increase the parameter count: number of filters and depth. For a one-dimensional standard convolutional layer over data $X \in \mathbb{R}^{n \times K}$ and kernel weights $W \in \mathbb{R}^{s \times K \times J}$ with the following form for the j-th output channel ($j \in \{1, \ldots, J\}$):

$$\text{Conv1D}(X, W)_j = \sum_{k=1}^{K} X_k * W_{kj} \tag{3.1}$$

where $*$ denotes one-dimensional discrete convolution, K is the number of input channels and J is the number of output channels. We refer to J as the number of filters, whereas depth increases the number of layers in a network. Increasing the number of filters in convolutional neural networks has long been known to decrease error [80], allowing practitioners to choose parameter counts that fit their memory, computation and error budget. The 2016 paper by He et al. exploring the impact of residual connections on CNNs [43] established techniques to allow depth scaling to decrease errors as well. We explore why QuadConv's architecture changes how it scales when compared to typical CNNs, and improve its design to allow it to scale in a more predictable fashion.

Ecosystem In order to allow practitioners to more easily use QuadConv to build standard architectures we create additional layers and user interface changes. There are a number of other layers which are not convolutional, but are key to implementing a typical CNN architecture. These layers include: linear layers, pooling layers, and 1×1 convolutional layers. Both linear layers and 1×1 convolutional layers do not explicitly use geometric information, and so require no generalization. However pooling layers are important operators used to downsample data in neural networks and are frequently reliant on the underlying grid structure of the data. The most common pooling layer is max pooling [77] and takes the following form for data $\boldsymbol{f}_k^i(\mathcal{X}) \in \mathbb{R}^N$:

$$\boldsymbol{f}_k^{i+1}(h(\boldsymbol{x}_n)) = \max_{l \in \mathcal{D}(\boldsymbol{x}_n)} \boldsymbol{f}_k^i(\boldsymbol{x}_l) \tag{3.2}$$

here we use $\mathcal{D}(\boldsymbol{x}_n)$ to denote a neighborhood of $\boldsymbol{x}_n$ and $h(\boldsymbol{x}_n) : \mathcal{X} \to \mathcal{Y}$ is a surjective function between sets. We generalize these operators using mesh downsampling techniques. Lastly, we update the interface to QuadConv using new objects that represent the mesh and important operations over this mesh. We will show how this reduces the number of lines of code required to implement standard architectures and makes working with meshes in neural networks an overall smoother experience.

3.2 Methods

First we will re-state the original QuadConv formulation, making it easier for the reader to reference. Whenever possible we will only use a single channel in order to simplify the notation and omit the subscript from $\boldsymbol{f}(\boldsymbol{x}_i)$. The output of the layer is given by

$$(\boldsymbol{f} * \boldsymbol{G})(\boldsymbol{y}) = \sum_{n=1}^{N} \rho_n \cdot \boldsymbol{G}(\boldsymbol{y} - \boldsymbol{x}_n) \cdot \boldsymbol{f}(\boldsymbol{x}_n) \ , \tag{3.3}$$

where the filter $\boldsymbol{G}$ is parameterized by $\boldsymbol{\theta}$ and its compact support is limited by a bump function:

$$\boldsymbol{G}(\boldsymbol{z}) = \text{bump}(\boldsymbol{z}) \cdot \boldsymbol{H}(\boldsymbol{z}; \boldsymbol{\theta}) \ . \tag{3.4}$$

We will replace the above formulation with the following,

$$(\boldsymbol{f} * \boldsymbol{G})(\boldsymbol{y}) = \sum_{n=1}^{N} \rho_n \cdot \mathbf{1}_{\mathcal{N}(\boldsymbol{y})} \cdot \boldsymbol{H}(\boldsymbol{y} - \boldsymbol{x}_n; \boldsymbol{\theta}) \cdot \boldsymbol{f}(\boldsymbol{x}_n) \tag{3.5}$$

where $\mathbf{1}_{\mathcal{N}(\boldsymbol{y})}$ is the indicator function over the neighborhood of $\boldsymbol{y}$.

3.2.1 Stability

The parameters in QuadConv (aside from any bias terms added to the convolution) are inside of the Multi-Layer Perceptron (MLP) $\boldsymbol{G}$, and so improving the initialization and the distributional stability will focus on the construction of the MLP and its integration into the convolutional operator. First, we adopt a new MLP architecture by Sitzmann et al. [97] called a SIREN MLP. This architecture uses a $\sin(\cdot)$ activation function, with linear layers that take the form

$$\boldsymbol{x}^{i+1} = \sin(W\boldsymbol{x}^i + \boldsymbol{b}) \, , \tag{3.6}$$

for $\boldsymbol{x}^i, \boldsymbol{b} \in \mathbb{R}^d$ and $W \in \mathbb{R}^{d \times d}$. The key result from Sitzmann et al. [97] is that if we begin with our input to a SIREN as $\boldsymbol{x}_n^0 \sim \mathcal{U}(-1, 1)$ then we can obtain a cycle of distributions as we move through the operations of a given layer, seen in Fig. 3.1. A specially initialized first layer maps $\boldsymbol{x}_n^0 \sim \mathcal{U}(-1, 1)$ to $\boldsymbol{x}_n^1 \sim \text{Arcsine}(-1, 1)$. Here $\text{Arcsine}(-1, 1)$ is a distribution

over the interval $[-1, 1]$ that is equivalent to a shifted Beta distribution. The first operation applied to $\boldsymbol{x}_n^1$ is the linear map W in Eq. (3.6) which gives us $[W\boldsymbol{x}^1]_n \sim \mathcal{N}(0, 1)$ and Sitzmann et al. shows that after the second layer of the SIREN we have $\boldsymbol{x}_n^2 \sim \text{Arcsine}(-1, 1)$. We terminate this cycle such that the output of our SIREN MLP is $\mathcal{N}(0, 1)$.

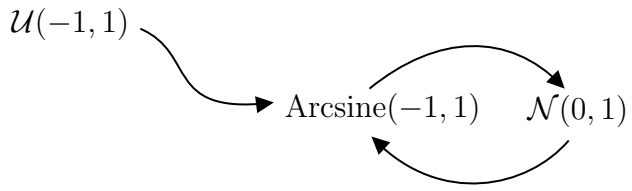


Figure 3.1: The distribution cycle induced by a SIREN MLP

We are hoping to develop an analogous cycle after QuadConv is applied to $\boldsymbol{f}(\boldsymbol{x}_n)$. If we assume that $\boldsymbol{f}(\boldsymbol{x}_n) \sim \mathcal{N}(0, 1)$, we want $Q(\boldsymbol{f}(\boldsymbol{x}); \boldsymbol{\theta})_n \sim \mathcal{N}(0, 1)$. In order to achieve this we want to further control ρ_n and $\boldsymbol{G}(\boldsymbol{z})_n$. Let $\Phi(\cdot)$ be the cumulative distribution function of a standard normal random variable. We use the following convenient property: for $X \sim \mathcal{N}(0, 1)$ and $Y = \Phi(X)$ and any $v \in [0, 1]$,

$$F_Y(v) = P(Y \leq v) = P(\Phi(X) \leq v) = (X \leq \Phi^{-1}(v)) = \Phi(\Phi^{-1}(v)) = v \ , \tag{3.7}$$

thus $Y \sim \mathcal{U}(0, 1)$. We use this fact by passing the output of the SIREN MLP through a pair of transformations. Let the action of our SIREN be denoted by the map $S(\cdot) : \mathcal{U}(-1, 1) \to \mathcal{N}(0, 1)$. If $\boldsymbol{x}_n^0 \sim \mathcal{U}(-1, 1)$, then we have

$$2 \times \Phi(S(\boldsymbol{x}_n^0)) - 1 \sim \mathcal{U}(-1, 1) \ . \tag{3.8}$$

Similarly, we generate the quadrature weights ρ_n via a small SIREN MLP, such that $\rho_n \sim \mathcal{U}(0, 1)$ and we then normalize the quadrature weights according to the following equation,

$$\sum_{n=1}^{N} \rho_n = 1 \ . \tag{3.9}$$

This ensures that our data driven quadrature is still exact for constant functions, and gives us the same distribution properties for our convolutional layer as found in standard convolution.

We will see the effects of these improvements in the results section when we compare to the original formulation of QuadConv.

3.2.2 Speed

In order to improve the speed of the layer, we addressed a handful of different issues that amount to over a 2× improvement in overall performance (Fig. 3.2). First, we eliminate the bump(·) function in the parameterization of the compactly supported kernel. Instead we replace the compact support with a k-nearest neighbor search that determines the compact support according to the kernel size hyperparameter. This saves the operations that would otherwise be associated with the forward and backward passes through the bump(·) function itself. Second, we replace the scatter operation (described in detail in Section 2.3.1) with an in-place scatter that eliminates a copy operation into the final array output. See He et al. for more information on scatter operations [42]. While these are minor changes, they result in hours of training time saved.

	Original QuadConv	Updated QuadConv
Forward Pass	9.47 ms	5.17 ms
Backward Pass	27.30 ms	11.40 ms
Cumulative	36.77 ms	16.57 ms

Figure 3.2: Speed improvements in our new QuadConv formulation: $N = 2189$ with a batch size of 1

3.2.3 Performance Scaling

As we discussed in Section 3.1.2, the scaling associated with standard convolutional neural networks is achieved through number of filters and depth. Depth can be added to any

neural network, regardless of layer structure. Adding filters to a QuadConv layer is more difficult, and so we address that here. For an MLP $\boldsymbol{H}(\boldsymbol{z};\boldsymbol{\theta})$, parameterized by $\boldsymbol{\theta}$, the total number of parameters is determined by the number of hidden layers and the width of those hidden layers. Suppose that $d = 2$ so our input is in $\mathbb{R}^2$, and we have 2 hidden layers of width 8, with an output of dimension C. This yields the following parameters:

Layer	Weights	Bias	Parameter Count
Layer 0	$W_0 \in \mathbb{R}^{8\times 2}$	$\boldsymbol{b}_0 \in \mathbb{R}^8$	$16 + 8$
Layer 1	$W_1 \in \mathbb{R}^{8\times 8}$	$\boldsymbol{b}_1 \in \mathbb{R}^8$	$64 + 8$
Layer 2	$W_2 \in \mathbb{R}^{8\times 8}$	$\boldsymbol{b}_2 \in \mathbb{R}^8$	$64 + 8$
Layer 3	$W_3 \in \mathbb{R}^{C\times 8}$	$\boldsymbol{b}_3 \in \mathbb{R}^C$	$8 \times C + C$

Figure 3.3: Example of parameter counts in an MLP: $168 + (8 \times C + C)$ total parameters

If we scale the number of channels in a QuadConv layer, we only scale the constant C, increasing the number of parameters but not in a useful way. There are two issues with this approach. $\boldsymbol{H}(\boldsymbol{z};\boldsymbol{\theta})$ is now approximating a higher dimensional function but its hidden layers are remaining of fixed size. MLPs are able to fit increasingly complex functions as their hidden layers increase in width, as noted in [83, 109], but keeping hidden layers a constant size does not allow for the increased complexity that more channels demands. The second issue is that $\boldsymbol{H}(\boldsymbol{z};\boldsymbol{\theta})$ is a continuous function, which then creates a relationship between the kernels for each channel, an atypical property for a convolutional layer. We amend both these issues by instead using many MLPs to represent $\boldsymbol{H}(\boldsymbol{z};\boldsymbol{\theta})$, one for each output channel. Thus instead of scaling the number of parameters in $\boldsymbol{H}(\boldsymbol{z};\boldsymbol{\theta})$, we scale the number of MLPs in $\boldsymbol{H}(\boldsymbol{z};\boldsymbol{\theta})$.

3.2.4 Ecosystem

Without the operators that surround the convolutional layer in popular architectures QuadConv is a much less useful layer. Thus we extend max pooling using mesh downsampling, in particular using a technique called Moving Front Non-Uniform Subsampling (MFNUS) recently developed by Lawrence et al. [58]. This subsampling technique creates a set of points $\tilde{\mathcal{X}} \subset \mathcal{X}$ which maintains the non-uniform density of the domain $\mathcal{X}$ and contains fewer points. This allows us to create an entirely grid free neural network, which never relies on implicit geometric information embedded in array shapes. If we examine the general form of max pooling

$$\boldsymbol{f}_k^{i+1}(h(\boldsymbol{x}_n)) = \max_{l \in \mathcal{N}(\boldsymbol{x}_n)} \boldsymbol{f}_k^i(\boldsymbol{x}_l) \tag{3.10}$$

then MFNUS is creating both the neighborhood map $\mathcal{N}(\cdot) : \mathcal{X} \to \mathcal{X}$ and generating the correspondence between the original space and the range of the pooling operator $h(\cdot) : \mathcal{X} \to \tilde{\mathcal{X}}$. We can see a visual representation of this in Fig. 3.4.

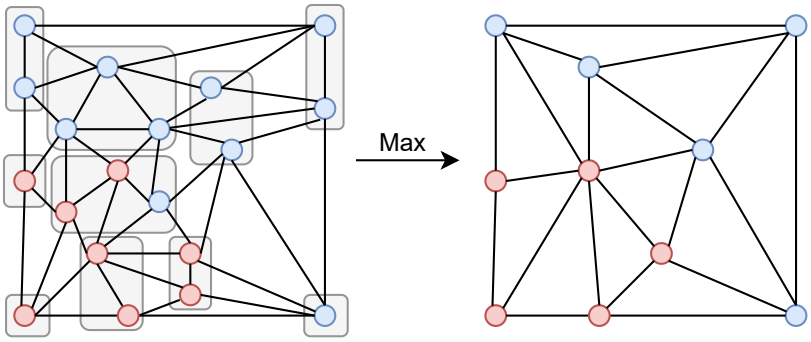


Figure 3.4: A mesh max pooling example, where red values are larger while blue values are smaller.

Beyond the creation of new operators, we have also simplified the interface to QuadConv. Previously, complex behaviors involving mesh downsampling involved a `Handler` object which monitored the mesh currently under usage. This object was globally responsible for the control of the data, which made coding with QuadConv difficult in large networks. Instead we have replaced this with a `Domain` object that allows users to directly control mesh

behavior, and downsample or upsample according to their algorithm of choice. The interface to QuadConv now mirrors convolutional layers in PyTorch and makes coding common architectures simple. We leave an example of building a `ResNetBlock` [43] in Appendix B for the interested reader.

3.3 Results

In order to demonstrate some of QuadConv's improvement after theses changes, we perform the same compression test found in Chapter 1 on the ignition mesh dataset. This test uses the new mesh downsampling with MFNUS, the stability improvements, and our new interface. The number of channels and batch size is also kept the same, however the changes in downsampling results in there being fewer parameters in the new network. Where the original QuadConv network first upsampled the data to a grid, before using standard Max Pooling, this new network can directly downsample the data on a mesh.

Model Type	Average Error	Max Error	Training Time (h)	# of Steps	# of Trainable Parameters
Improved QuadConv	0.621%	4.17%	2.38	100,000	844,000
QuadConv	0.596%	2.05%	13.50	450,000	1,037,499

Table 3.1: Results for non-uniform ignition data compression.

As seen in table Table 3.1, the errors are similar between both networks and the improved version of QuadConv is converging much more quickly both in terms of runtime and in terms of number of steps. Runtime improvements are due to the changes in the code's speed, the number of steps to converge is decreased due to improvements in stability. The new network is running at roughly 42,000 steps per hour of wall time, versus about 33,000 steps per hour for the original version, which is not quite the 2× improvement seen in the layer, since there are other operations taking place in the network. While a reduction of 350,000 steps until convergence is a substantial improvement in the convergence of our network to a low error stationary point.

3.4 Conclusion

In this chapter, we present a number of improvements to the neural network layer that we built in Chapter 1. These improve computational speed, convergence rate (measured in epochs), accuracy and usability. We also present an extension of max pooling that allows us to create entirely grid free neural networks. When evaluated on the same problem as in Chapter 1, we see that the improvements make QuadConv improve its ability to compress data when used in an autoencoder. We hope to use QuadConv in other problems where convolutions are state of the art and non-uniform meshes or point clouds are commonplace in data. We believe that these networks give significant advantages in performance over these data sources, while also opening up exciting research into data dependent quadrature rules, continuous kernels and multi-resolution applications.

Chapter 4

Manifold Harmonic Bases for Compression

Kevin Doherty

Stephen Becker

Alireza Doostan

4.1 Introduction

While in the previous chapters we discuss data compression using neural networks, these are not typical techniques used for data compression. The most common setting for the usage of data compression does not permit extended amounts of training time, and may require extremely high levels of accuracy that neural networks cannot currently provide, cf. Fig. 4.1. More typical methods are those like JPEG [111] or scientifically oriented successors e.g., ZFP [64] and SPERR [61]. Methods in this low error and low compression regime prioritize the speed of compression and decompression, possibly sacrificing compression rate in the process. The methods like JPEG, ZFP and SPERR are commonly referred to as "block coder" style compressors, due to breaking the dataset into blocks before being encoded in a spectral representation [9, 107]. Block sizes can vary widely, depending on the method, ranging from $4 \times \cdots \times 4$ in ZFP to $256 \times \cdots \times 256$ SPERR. These methods frequently use discrete cosine transforms [2] or wavelet transforms [12] in order to accomplish their spectral encoding. In the case of ZFP, this means using a special integer coefficient discrete cosine transform approximation so that multiplication can be accomplished exclusively through bit shifts. In both of these steps, the data to be compressed is implicitly interpreted to be de-

fined over a uniform grid. In the case of JPEG [111], this presents no issue, but when aiming to compress scientific data it is not uncommon to define complex non-Cartesian domains with varying levels of resolution in each section of the domain.

In this chapter we will present a novel block coder called Mesh-Float-Zip (MFZ) which performs spatial compression over spatially non-uniform data. This block coder is similar to ZFP and JPEG in that it uses a linear transformation of the data before entropy encoding. This makes it rigorous, interpretable and error bounded as opposed to the neural methods seen in previous chapters.

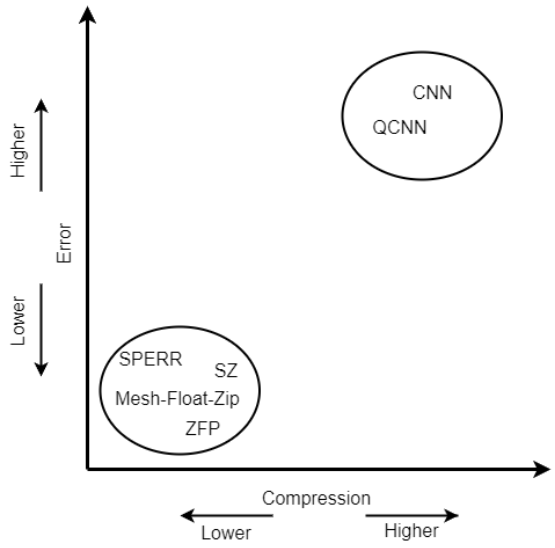


Figure 4.1: A rough comparison of compression techniques seen in this thesis.

Block coder methods begin with two key steps: the partitioning of the data into blocks, and the projection of these blocks onto a suitable spectrum. The blocking of arbitrary meshes is already understood [52], but the generalization of spectral transformation for data compression has not been explored and is the focus of this chapter. We establish a suitable generalization for the spectral encoding of data on a mesh and create a compression tool which replicates the success of other methods over these meshed datasets.

One contribution of this work is in using the Manifold Harmonic Basis [105] as spectral

transformation for data over a mesh, thereby improving data compression for an important class of scientific data. An additional contribution is the method by which we derive the Manifold Harmonic Basis, which is an extension of simple collocation techniques explored by Kansa in 1990 [49], but applied in this case to a Generalized Eigenvalue Problem (GEVP). This improves the speed of our basis generation, and simplifies its generalization to higher dimensions.

First, blocking data which is arranged in a d-dimensional grid is a straight-forward operation accomplished by indexing into the array in which the data is stored. This is because the memory layout of the data often matches the spatial layout of the data. The purpose of the blocking in general is to create spatially congruent regions which can be compressed in parallel, speeding up the compression process and isolating regions of highly compressible data from regions of incompressible data. This means that any algorithm which creates spatially congruent regions of blocks of size N on an arbitrary mesh is suitable for this application. We choose to use METIS [52] in our research as it is a well established and fast graph partitioning algorithm.

Second, we aim to replicate the effect of the spectral encoding on the blocks of data. This is where our contribution to this topic is focused. We choose to apply the Manifold Harmonic Basis (MHB), which was first explored in a paper by Taubin in 1995 [105]. The MHB is a generalization of the Discrete Cosine (or Sine) Transform (DCT or DST) and we will use it to spatially de-correlate data before entropy encoding. This basis has seen extensive usage in graphics processing, frequently being used to perform operations like low-pass filtering of noisy 2D surfaces [110]. Usage is complicated by its definition, since it does not possess a closed form solution over most domains. Rather the Manifold Harmonic Basis consists of eigenfunctions of the following eigenvalue problem

$$-\Delta f = \lambda f \tag{4.1}$$

where $\Delta f = \sum_i \frac{\partial^2 f}{\partial x_i^2}$.

The continuous eigenproblem in Eq. (4.1) takes on different spectral properties depending on the domain of the function in $L^2(\Omega)$, where Ω is the domain under consideration. For a simple domain like $\mathbb{R}$ we have a continuous spectrum from $[0, \infty)$, where each of the eigenfunctions is of the form $f(x) = \sin(\lambda x + c)$ for $x \in \mathbb{R}$ and $c \in \{0, \frac{\pi}{2}\}$. If instead we take the domain to be $\mathbb{R}/\mathbb{Z}$, the spectrum is now $\{2\pi k : k \in \mathbb{N}\}$ where again $f(x) = \sin(\lambda x + c)$ with $x \in \mathbb{R}$ and $c \in \{0, \frac{\pi}{2}\}$. In general, spectral analysis of the Laplace operator is a very deep and well researched field, which is outside the scope of this work but a good reference can be found in [38].

If we take the domain of f to be $\mathbb{R}/\mathbb{Z}$ with boundary conditions $f'(0) = 0$ and $f'(1) = 0$, then discretizing the problem with uniformly spaced points over the interval we recover a DCT. For the appropriate change of boundary conditions over this domain we can in fact recover type I, II, III or IV DCT (or DST) [102]. The Manifold Harmonic Basis provides us with a generalization of these common spectral transformations which can be applied to any mesh, provided we can find one that is discretized on our domain in an appropriate fashion.

The solution of Eq. (4.1) is typically derived through a GEVP involving a Finite Element (FE) Discretization of the Laplace operator. We propose a computationally cheaper collocation method based on [49] which satisfies the necessary conditions to create a Manifold Harmonic Basis. Our method also takes the form of a GEVP, and requires no changes to work in any dimensional space.

In the remainder of this chapter we will further motivate the usage of the Manifold Harmonic Basis, provide a method for deriving it quickly over our data points, integrate METIS and the Manifold Harmonic Basis into a compression chain that is modeled after ZFP [64], and demonstrate effectiveness at recovering and utilizing the spatial correlation that enables efficient compression.

4.1.1 Motivation

Recall that the Shannon Source Coding Theorem (Theorem 1) is concerned with a random source, X, from which we can produce a sequence of i.i.d. variables $X_1, X_2, \ldots, X_n$. These variables take discrete values in a finite set of symbols $\mathcal{X}$. While Theorem 1 holds for i.i.d. random sequences, in practice we are interested in compressing sequences that have a non-trivial covariance structure. We don't have optimal encoding schemes for data that are not i.i.d. and so in order to best utilize things like Huffman encoding, or Asymmetric Numerical Systems, we attempt to diagonalize the covariance structure of the data to make the data uncorrelated (as a proxy for independence). The Karhunen-Loève Transform [87] is the transform that diagonalizes a covariance matrix. In our case the data are a discrete sequence, and so the Karhunen-Loève transform is given by applying the adjoint of the eigenvectors of the (empirical) covariance matrix of the elements of the sequence. Specifically, we use the covariance matrix

$$\Sigma_{ij} = \hat{\mathbb{E}}[(X_i - \mu_X)(X_j - \mu_X)] \tag{4.2}$$

which gives the eigenvalue problem

$$\Sigma e = \lambda e \tag{4.3}$$

and after collecting the eigenvectors e as columns of the matrix E, the Karhunen-Loève transform is

$$X \mapsto E^*(X - \mu_X). \tag{4.4}$$

In the case of uncorrelated and constant variance X_i, the solution is trivial as $\Sigma = \sigma^2 I$ and hence the Karhunen-Loève transform is the (possibly shifted) identity map. However spatial data often has non-trivial covariance structures, so finding a Karhunen-Loève Transform is very important for compression.

The following example of covariance structure is the most important one in data compression, as it is the spatial correlation structure that is assumed for ZFP [64] and JPEG

[111]. Consider a uniform 1D grid, in which neighboring points have a covariance governed by the following expression with a scalar ρ,

$$\Sigma_{ij} = \sigma^2 \rho^{|i-j|} \ . \tag{4.5}$$

This covariance matrix gives rise to a very familiar family of Karhunen-Loève Transforms, namely the 1D DCT and the DST [87, 93].

This motivation is a heuristic that attempts to explain the effectiveness of the DCT and DST transformations, which is ubiquitous in data compression. Our work aims to use a generalization of the DCT and DST in order to provide a de-correlating transform for non-uniform data. This transform, a projection onto the Manifold Harmonic Basis [110], allows us to more effectively de-correlate non-uniform data and thus provide better compression for an important class of scientific data.

4.1.2 Related Work

There are a number of efforts to produce high efficiency error-bounded data compression tools for scientific data. The two most significant efforts are ZFP [64, 18, 30, 40, 63] and SZ [17, 119, 62, 67, 68]. Each effort has produced a number of different versions of the software over the last 5+ years, but we aim to characterize the most significant aspects of these methods.

ZFP ZFP is a block-coder style compression tool, that is designed for data on a uniform grid. Operating on $4 \times \cdots \times 4$ blocks, this algorithm takes a block of data, and transforms it using a DCT to decorrelate the data. This is followed by a sequence of bitwise operations that convert the 4^n numbers into a different kind of floating point representation. The only lossy stage of this process is a zeroing out of a number of low significance bits, based on a user provided threshold for the resulting error. Finally this is followed by an entropy encoding of the bit-planes. The development of ZFP has highlighted its zero mean error distributions [63, 18] and its speed of compression [40]. Due to both the blocking and

the data transformation in ZFP, when compressing non-uniform meshes only the implicit spatial correlation that is contained in the memory layout of the data is leveraged.

SZ SZ is a predictor-corrector framework that was originally based on using curve fitting to interpolate data, and encoding large errors explicitly [17]. Its most recent version, which sacrifices a small amount of speed for increased performance, is SZ3 [62]. SZ3 is a framework of compression stages, beginning with preprocessing (like blocking and data transformations), followed by prediction using neighboring points [46] or deep neural networks [68], then quantization, encoding and a lossless compressor that captures any latent structure after encoding. SZ3 has a number of options for each stage, and adaptively selects the most performant option. As with ZFP, since SZ does not use explicit geometric information, only the implicit spatial correlation that is contained in the memory layout of the data is leveraged.

Other approaches There are other approaches which we will cover with less depth. TTHRESH [5] is a recent tensor decomposition based method that is not sufficiently applicable for non-uniform mesh data. There are also other tensor decomposition based methods that do not apply such as [4] and [50]. MGARD [36] is a multi-grid method, which uses a multi-resolution decomposition of the data to make the retrieval of lower precision data much faster. This is clearly applicable to non-uniform mesh data, but MGARD's primary contribution is the speed of its compression and decompression of data [14].

Manifold Harmonic Basis Aside from data compression, there is also related work involving the generation of the Manifold Harmonic Basis (MHB). The MHB is used in mesh filtering, mesh compression, parametrization, and segmentation [51, 106, 99, 29]. This task is complicated by a no free lunch theorem [113] for the discretization of Laplace operators. The no free lunch theorem states that for 2D non-uniform meshes, Laplace discretizations cannot be symmetric, strictly positive, locally defined, and vanish on linear functions. Each of these properties may be important for different applications of the discretized operator. Initial attempts to generate these basis functions used symmetrized discrete Laplacian op-

erators [72], defined over a graph ($G = (V, E)$), and Galerkin projection [60]. Later a more refined approach used the Discrete Exterior Calculus to more correctly consider the symmetry and discretization of the continuous Laplace operator [110]. Yet other approaches have been applied to point clouds using Voronoi cells [69] or Delaunay triangulation [7]. In the context of compression, these approaches can be slow or may not generalize directly beyond 2 dimensions. Our method is instead based in radial basis function (RBF) interpolation and avoids the discretization of the Laplace operator, which allows for a faster construction of the MHB that works in all dimensions.

4.1.3 Contributions

Our contributions primarily fall into two categories. First, those contributions we make to the generation of the MHB, and second, the contributions we make to the compression of data discretized on a mesh. Aside from these two groups of contributions, this work is also the first to use the MHB to compress data that lies on a mesh. While Karni et al. [51] used the MHB in a compression setting, it was to compress the nodes of the mesh.

MHB Generation The collocation method we use to generate the MHB is most closely related to Kansa's original RBF collocation method [49]. The equation on which we are using this collocation method bears resemblance to the Helmholtz equation ($\Delta f = -\kappa^2 f$), but we are not concerned with any particular solution for a given κ, and furthermore we seek real solutions since complex solutions make the resulting basis unusable for data compression.[1] Thus the considerations we make to guarantee that the Generalized Eigenvalue Problem produces a basis of real eigenvectors is unique to our work. This is also the first MHB generation method which is not dependent on the particular dimension the points are embedded in, a consequence of the collocation technique.

[1] Projecting real data onto a complex basis results in a doubling of the number of bits required to represent the spectrum.

Compression The DCT has convenient properties that make the resulting compression chain slightly easier than when using the MHB. We have overcome these challenges with small contributions to the block coder compression chain. The MHB, unlike the DCT, is not guaranteed to contain the all ones vector, $\mathbf{1} \in \mathbb{R}^n$. To fix this we simply subtract the mean from the data, and compress this scalar along with the spectra of the mean centered data. More significantly, the DCT on a grid produces a simple and convenient ordering for the spectra, via spatial frequencies of the cosines. For an element of the DCT, $\cos 2\pi\lambda_i x$, with frequency λ_i, we can order the spectrum according to the sorting of λ_i. However the basis elements of the MHB are not spatially uniform, and eigenvalues of the GEVP may have geometric multiplicity. This makes the compression of the data spectrum more complicated, as we cannot use the method ZFP [64] does to compress trailing zeros of each bit plane. Instead we apply a range encoder, whose compression performance is only reliant on the entropy of the data, and not the ordering of the spectrum.

4.2 Methods

The focus of this work is the usage of the MHB as a DCT replacement for entropy exploitation, and we demonstrate this via a implementation of a data compression scheme. In order to highlight the impact of the MHB, we implement a similar compression chain to ZFP. First we will explain our contribution to the derivation of the MHB which is via an RBF collocation method [31], then we will explain the compression chain, before demonstrating our results on test data.

4.2.1 Manifold Harmonic Basis via Interpolation

While many methods for deriving the MHB involve a discretization of the Laplace operator via Finite Elements, we instead consider a function $f : \mathbb{R}^d \to \mathbb{R}$, whose values we consider on the set of points $\{\boldsymbol{x}_i\}_{i=1,\dots,n} = \mathcal{X}$ for $\boldsymbol{x}_i \in \mathbb{R}^d$. We can then form an interpolant $\tilde{f}$ via an RBF approximation to f

$$\tilde{f}(\boldsymbol{x}) = \sum_i \alpha_i \phi(\|\boldsymbol{x}_i - \boldsymbol{x}\|_2), \tag{4.6}$$

where the function $\phi(\cdot)$ is a RBF. There are many options for $\phi(\cdot)$, but we will require that the resulting interpolation matrix is positive definite and symmetric. In this work we use the kernel

$$\phi(\|\boldsymbol{x}_i - \boldsymbol{x}\|_2) = e^{-(\epsilon\|\boldsymbol{x}_i - \boldsymbol{x}\|_2)^2} \text{ for } \epsilon \geq 0,$$

since it is a common choice and also yields the required properties [73]. There are some other RBFs that fit this criteria, but we do not explore them here. If we were to fit an interpolant, we would use the interpolation system with $r_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$,

$$\begin{bmatrix} \Delta\phi(r_{11}) & \Delta\phi(r_{21}) & \dots & \Delta\phi(r_{1n}) \\ \Delta\phi(r_{12}) & \Delta\phi(r_{22}) & \dots & \Delta\phi(r_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ \Delta\phi(r_{1n}) & \Delta\phi(r_{2n}) & \dots & \Delta\phi(r_{nn}) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{x}_1) \\ f(\boldsymbol{x}_2) \\ \vdots \\ f(\boldsymbol{x}_n) \end{bmatrix}. \tag{4.7}$$

In matrix form we write this as,

$$\Phi\boldsymbol{\alpha} = f(\mathcal{X}). \tag{4.8}$$

Consider the eigenproblem for the Manifold Harmonic Basis applied to the continuous interpolant,

$$\Delta\tilde{f} = \lambda\tilde{f} \ . \tag{4.9}$$

We can then apply the Laplace operator directly to the continuous function,

$$\Delta\tilde{f}(\boldsymbol{x}_j) = \sum_i \alpha_i \Delta\phi(\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2) = \lambda\tilde{f}(\boldsymbol{x}_j) \ . \tag{4.10}$$

Rearranging and letting $r_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$ we can turn this into a discrete eigenvalue problem over the vector of interpolation coefficients $\boldsymbol{\alpha}$,

$$\begin{bmatrix} \Delta\phi(r_{11}) & \Delta\phi(r_{21}) & \dots & \Delta\phi(r_{1n}) \\ \Delta\phi(r_{12}) & \Delta\phi(r_{22}) & \dots & \Delta\phi(r_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ \Delta\phi(r_{1n}) & \Delta\phi(r_{2n}) & \dots & \Delta\phi(r_{nn}) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \lambda \begin{bmatrix} \phi(r_{11}) & \phi(r_{21}) & \dots & \phi(r_{1n}) \\ \phi(r_{12}) & \phi(r_{22}) & \dots & \phi(r_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(r_{1n}) & \phi(r_{2n}) & \dots & \phi(r_{nn}) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} . \tag{4.11}$$

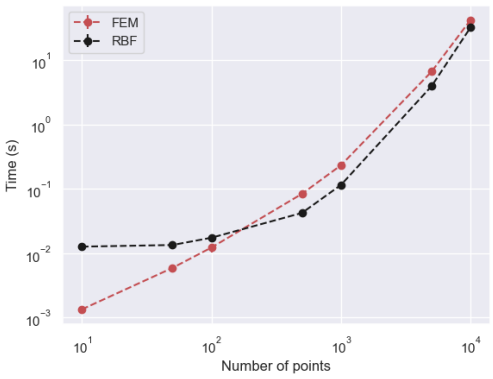
We can write this generalized eigenvalue problem (GEVP) in matrix form as,

$$\Delta\Phi\boldsymbol{\alpha} = \lambda\Phi\boldsymbol{\alpha} \ . \tag{4.12}$$

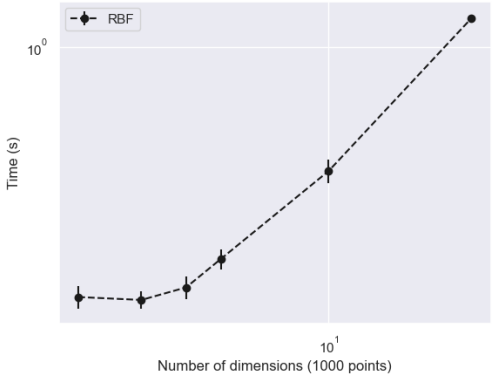
Effectively we are using the interpolated form as an ansatz for the solution of Eq. (4.12), which allows us to generate a MHB with less computational burden than a FE method. We are guaranteed that this generalized eigenproblem has a set of n linearly independent solutions $\{\boldsymbol{v}_i\}_{i=1\dots n}$ for $\boldsymbol{v}_i \in \mathbb{R}^n$ with real eigenvalues $\{\lambda_i\}_{i=1\dots n}$ when $\Delta\Phi$ and Φ are symmetric, and Φ is positive definite [35]. These are important features for the usefulness of the solution, since without a basis of vectors we risk having data that inhabits the null space of $\{\boldsymbol{v}_i\}_{i=1\dots n}$. This would lead to uncontrolled loss of data into this null space, and increase compression errors. If those vectors are not also real, then projection onto the vectors leads to a doubling of the data to store the spectrum.

RBF interpolation using Gaussian RBFs involves the selection of the parameter $\epsilon \geq 0$ and there is a well known trade off, called the uncertainty principle [22], between the numerical stability of the large ϵ regime and the high interpolation accuracy of the small ϵ regime. Here we use $\epsilon = 1$ for all our experiments, and 32 bit floating point numbers in the GEVP.

Now that we have established how we form the MHB using our RBF collocation method, it is important to distinguish it from the traditional FE method found in the literature. The FE method has been run at the same resolution as the discretized domain



(a) Runtime vs size of a 2D mesh.



(b) Runtime vs dimension of the points.

Figure 4.2: Runtime of MHB Generation.

in each case. While in 2D the FE method produces a basis of real solutions, in Fig. 4.2a we can see it is marginally slower. In the small N regime the speed of the RBF method is dominated by the auto-differentiation framework with which we implemented the GEVP system. This difference in speed is insignificant to the compression task, since the MHB can be generated once and used for compression over and over again. Much more importantly, the FE method has been developed specially for 2D spatial data, and so higher dimensions of non-uniformity are not compatible with this method. In Fig. 4.2b we can see that the RBF collocation method works in any dimension.

In order to illustrate more clearly how these first two stages work, let's take the example mesh in Fig. 4.3a and partition it into the blocks in Fig. 4.3b.

(a) An example simulation mesh.

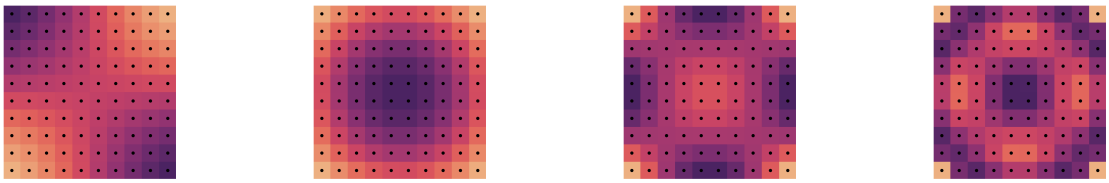
(b) The example mesh after using METIS to create blocks.

Figure 4.3: An example mesh and its color-coded partition using METIS.

We can now take a particular block and form the MHB over that block using the RBF interpolation we described above Fig. 4.4. This allows us to form the spectral representation of the data in the block that we will later compress.

Figure 4.4: An example block and some elements of its MHB.

When the domain of the MHB changes, the functions also change and when we apply the RBF collocation method to a grid, these basis elements more closely resemble the familiar DCT/DST.



$\boldsymbol{v}_1$ $\boldsymbol{v}_2$ $\boldsymbol{v}_{11}$ $\boldsymbol{v}_{23}$

Figure 4.5: Elements of a grid's MHB.

4.2.2 Compression of the MHB Spectrum

After partitioning and projecting the data blocks onto the Manifold Harmonic Basis, we still must perform compression on the resulting data. In order to accomplish this we use a pipeline very similar to ZFP, which will facilitate comparison between the two methods and allow the effect of the basis on the compression to be more obvious.

Let $\boldsymbol{f}^k \in \mathbb{R}^n$ be the k-th block of the partition, and let $\{\boldsymbol{v}_i^k\}_{i=1,\dots,n}$ be the basis for the k-th block of the partition. We occasionally add an i superscript to elements of the bit representations of floating point numbers to denote that they belong to the i-th element of the data, but in general omit them when we are only working with a single representative number.

Mean Centering and Projection First we mean center the data: for $\mu^k = \frac{1}{n}\sum_j f_j^k$, the centered data $\tilde{\boldsymbol{f}}^k$ is then

$$\tilde{\boldsymbol{f}}^k = \boldsymbol{f}^k - \mu^k. \tag{4.13}$$

The spectrum of the data block is given by

$$\boldsymbol{g}^k = V_k^T \tilde{\boldsymbol{f}}^k. \tag{4.14}$$

for $V_k = [\boldsymbol{v}_1^k, \dots, \boldsymbol{v}_n^k]$.

Block Floating Point First the data block is converted to a block floating point representation, which uses a single exponent for the entire block. Consider the case where

our data are IEEE single precision floating point numbers [48], which is a 32 bit floating point representation. This representation consists of 1 sign bit, 8 exponent bits and 23 bits to represent the mantissa, taking the form

$$g_i^k = (-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times \left(1 + \sum_{j=1}^{23} b_{23-j} 2^{-j}\right). \tag{4.15}$$

Here the notation $(0010101)_2$ denotes an integer base two, which in this case $(0010101)_2 = (21)_{10}$ [2] . We replace the exponents in the block with the maximum exponent found in the block,

$$(c_7 c_6 \ldots c_0)_2 = \max_i (b_{30}b_{29}\ldots b_{23})_2^i. \tag{4.16}$$

By using a single exponent for the entire block, we save $8 \times (n-1)$ total bits per block. The new form of this floating point number is then

$$g_i^k = (-1)^{b_{31}} \times 2^{(c_7 c_6 \ldots c_0)_2 - 127} \times \left(\sum_{j=0}^{22} b_j 2^{j-22}\right). \tag{4.17}$$

Here $(c_7 c_6 \ldots c_0)_2$ is the common exponent for the block floating point representation. If the block has a wide range of values in it, this can create compression error unless extra bits are added to the floating point representation. In practice, the size of the blocks aims to make data of a homogeneous scale inside each of the blocks.

Thresholding At runtime, the user specifies an error threshold, in our case in terms of absolute pointwise error. This is then used to set bits below the threshold to zero. This is accomplished by first converting the threshold into our block floating point representation and then by comparing the resulting threshold with the least significant bits of the data in the block. Setting these bits to zero lowers the entropy of the bits in the block, and makes them more compressible.

Truncation After thresholding, many of the bits in the least significant bit planes, see Fig. 4.6, are now zeros. We can encode these bits in a single integer which encodes how

[2] The -127 shift allows floating point comparisons between negative and postive numbers to examine fewer bits.

many zeros to add to the end of our representation. This is similar to a run length encoding of the trailing zeros in each of the n numbers in the block, but we use the minimum run length in order to encode them simultaneously in one integer. This saves a total of $(t-1)\times(n-1)$ bits per block. Note the change in the summation index for $t-1$ truncated bits,

$$g_i^k = (-1)^{b_{31}} \times 2^{(c_7 c_6 \ldots c_0)_2 - 127} \times \left(\sum_{j=t}^{22} b_j 2^{j-22} \right). \tag{4.18}$$

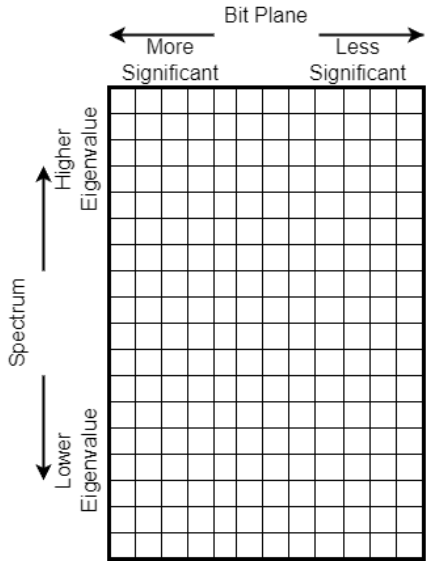


Figure 4.6: A sketch of the bit array after processing into block floating point format.

Nega-Binary We can view our current representation in the following form,

$$g_i^k = 2^{(c_7 c_6 \ldots c_0)_2 - 127} \times 2^{-22} \times \left((-1)^{b_{31}} \times (b_{22} b_{21} \ldots b_t)_2 \right) \tag{4.19}$$

where the term on the right of the equation can be interpreted as a (signed) integer. To facilitate more efficient bit plane encoding later, we convert this term to negative binary which encodes both the magnitude and the sign of the number in the first nonzero bit of the representation. This gives us the following,

$$g_i^k = 2^{(c_7 c_6 \ldots c_0)_2 - 127} \times 2^{-22} \times (q_{23} q_{22} \ldots q_t)_{-2}. \tag{4.20}$$

where $\{q_i\}_{i=23,\ldots t}$ are the bits of the negative binary representation up until the truncated bit plane $t-1$.

The chain of steps described above, beginning with the block floating point representation and ending with the negative binary conversion are all elements found inside of the ZFP compression chain. This gives us a compact representation of the spectra of our block and allows us to continue to a lossless encoding step that further compresses each block.

Arithmetic Range Encoding We apply a classical technique from the late 1970s known as arithmetic range encoding to each bit plane in order to take further advantage of the reduced entropy of the spectral representation of the data [71]. This approach encodes a stream of bits as a single number by using a model of the probabilities that each symbol will occur. Our case is the most simple, since our alphabet consists of $\{0, 1\}$. Given a pair of probabilities $\{p_0, p_1\}$ the fraction of the data that is 0 and 1 respectively, and a range within which to encode our symbols, we apply the algorithm symbol by symbol until the entire message is encoded.

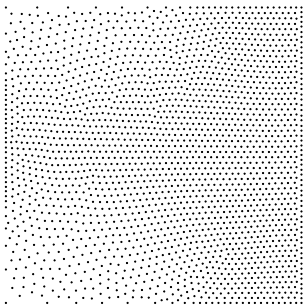
This encoding method relies on accurate probabilities and messages of sufficient length to avoid the overhead cost from expanding the length of the message [57]. In the case where $p_0 = 0.5 = p_1$ no compression is possible. We utilize a 32-bit integer for the encoding range via the python package "constriction" [6].

We expect the MHB transform to produce a few large coefficients and many smaller coefficients, on average. Let the bit planes be indexed by $k \in \{23, 22, \ldots, t\}$. This means that as we move from the most significant bit plane (23), towards the least significant bit plane (t), we expect the first few bit planes to contain a larger proportion of zeros, relative to the least significant bit planes. We find the bit plane in which we first see $p_0^k = 0.45$, which acts as a threshold for when we no longer expect to compress the following bit planes,

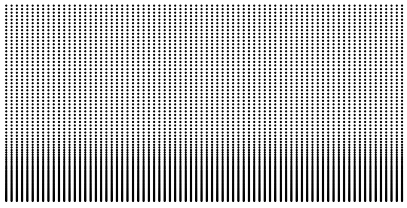
and then find the true proportion of zeros in the preceding bit planes. We then transmit two quantized numbers with the range encoded data: one number encodes p_0, as a fraction in $(0.0, 0.5)$ (in 4 bits) and the other encodes the bit plane at which we no longer apply arithmetic range encoding (in 5 bits). We call this method Mesh-Float-Zip, or MFZ for short.

4.3 Results

We test on a pair of 2D datasets and one 3D dataset which are all discretized over non-uniform meshes. The first 2D dataset is a jet flame ignition process that is seen in Section 2.4.2, and the second is a turbulent flow field over a non-uniform grid Section 4.3. Each of the 2D datasets has 4 channels of data, and is represented as single precision floating point numbers. The 3D dataset is a uniform mesh over a non-Cartesian domain, and has 1 single precision floating point channel of data. Each channel is processed separately in these examples to highlight how performance can vary over different data.



(a) The ignition mesh.



(b) The flat plate turbulent flow mesh.

We compress the data exclusively in the spatial dimensions, to emphasize the impact of the spatial compression that each method is performing. Thus, to measure the performance of each method at a given compression ratio, we perform time averaging over our metrics

to arrive at a final metric for the performance of the spatio-temporal dataset. Let $\boldsymbol{F}^t$ be a spatial slice of the data at time t and $\tilde{\boldsymbol{F}}^t$ its decompressed approximation. We will use a pair of different metrics to evaluate performance, the first is a time-averaged relative Frobenius-norm also seen in Chapter 1 Eq. (2.12),

$$\frac{1}{T}\sum_{t=1}^{T}\frac{\|\tilde{\boldsymbol{F}}^t-\boldsymbol{F}^t\|_{\mathrm{HS}}}{\|\boldsymbol{F}^t\|_{\mathrm{HS}}} \tag{4.21}$$

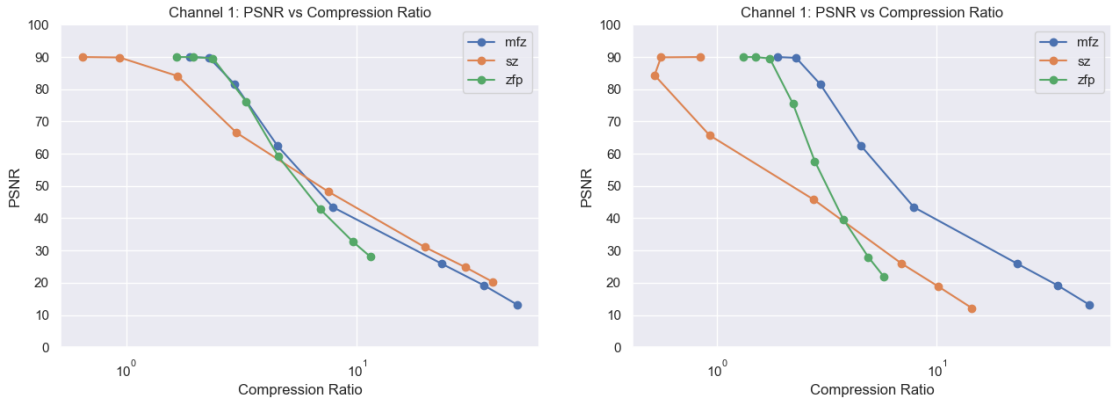
where $t=1,\ldots,T$ indexes the time dimension of the (time-dependent) PDE simulation and $\|\boldsymbol{F}^t\|_{\mathrm{HS}}=\left(\sum_{ij}|\boldsymbol{F}^t_{ij}|^2\right)^{1/2}$ is the Hilbert-Schmidt (aka Frobenius) norm. The second is the time-averaged Peak-Signal-to-Noise-Ratio or PSNR,

$$\frac{1}{T}\sum_{t=1}^{T}10\log_{10}\left(\frac{I^2}{\mathrm{MSE}(\boldsymbol{F}^t,\tilde{\boldsymbol{F}}^t)}\right) \tag{4.22}$$

where $\mathrm{MSE}(\boldsymbol{F}^t,\tilde{\boldsymbol{F}}^t)=\frac{1}{N}\sum_{ij}(\boldsymbol{F}^t_{ij}-\tilde{\boldsymbol{F}}^t_{ij})^2$ is the mean-squared error and $I=\max\boldsymbol{F}^t-\min\boldsymbol{F}^t$.

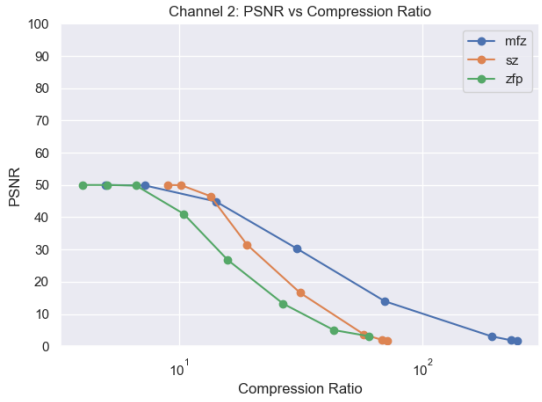
4.3.1 Ignition Dataset

This dataset consists of 450 uniformly sampled time steps of a jet ignition simulation that lie on a non-uniform spatial mesh of size 2189. The wavefront is fully resolved in time. This dataset is transport dominated for the initial portion of the simulation until the jet flame reaches steady state. The dataset is approximately lexicographically ordered, so even in the mesh the points that are adjacent in memory are spatially correlated. In order to show the difference this makes in the results we run ZFP, SZ and Mesh-Float-Zip (MFZ) on each of the channels in the dataset both with and without shuffling the data. PDE data that is adjacent in memory is not always spatially correlated, as it may be rearranged for more efficient memory access patterns or as in pivoting to reduce matrix condition numbers.

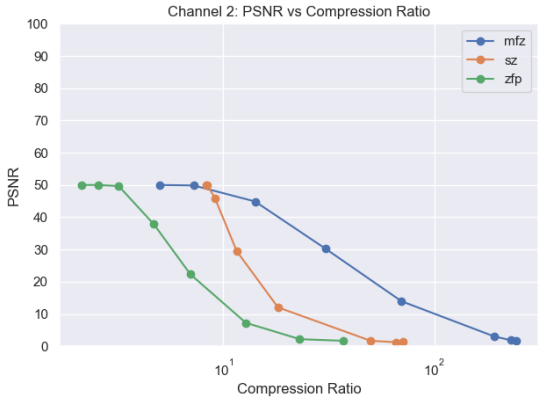


(a) Oxygen Concentration

(b) **Shuffled** Oxygen Concentration



(c) CO_2 Concentration



(d) **Shuffled** CO_2 Concentration

Figure 4.8: Demonstrating the effect of shuffling the data order, for combustion data. In the plots, higher PSNR is better.

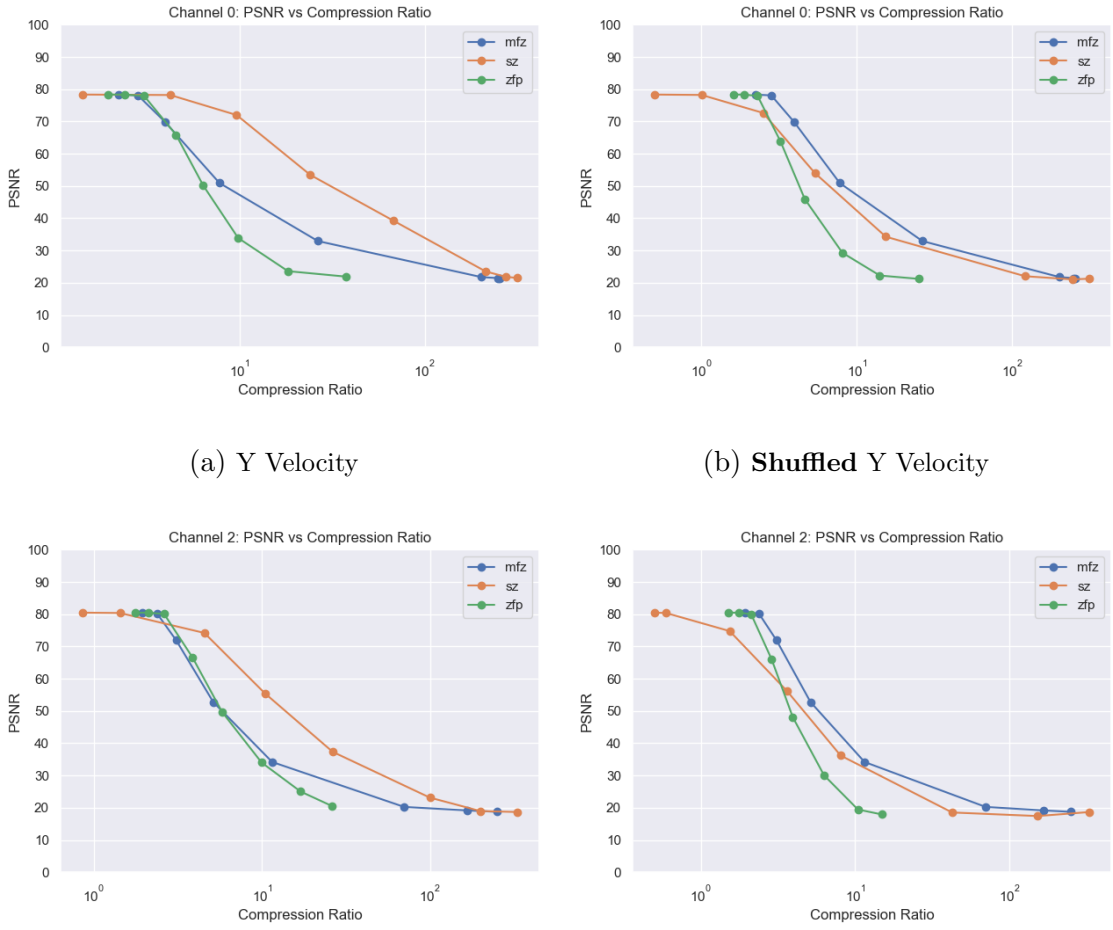
We can see in Fig. 4.8 that MFZ performs well across all of the data seen here, but there is wide variation in the overall performance. The oxygen concentration channels, seen in Fig. 4.8b and Fig. 4.8a have the most complex dynamics since the oxygen jet is being injected into the simulation domain. This makes these channels much less spatially compressible, and so the margins between each of the methods are very small. Even still, the shuffled data Fig. 4.8b exhibits a sharp decrease in performance for ZFP and SZ. This shows that these methods are heavily relying on the memory layout of the data to match

the results of MFZ.

The CO_2 channel seen in Fig. 4.8c and Fig. 4.8d has much more simple dynamics and a large number of zeros are found in the data, making this a much easier channel to compress. As there is more spatial correlation and structure, MFZ is able to take advantage of this structure and performs better than ZFP and SZ in both the shuffled Fig. 4.8d and unshuffled Fig. 4.8c cases.

4.3.2 Flat Plate Turbulent Flow

This dataset consists of 603 time steps with 9956 spatial points, and simulates a turbulent fluid flow over a flat plate. Unstable turbulence structures present near the plate on the bottom of the domain near the no-slip boundary condition and so the mesh is resolved more finely in that region. This dataset is discretized over a non-uniform grid, making it the most similar to the expected spatial layout that ZFP and SZ were designed for.



(a) Y Velocity

(b) **Shuffled** Y Velocity

(c) Streamwise (X) Velocity

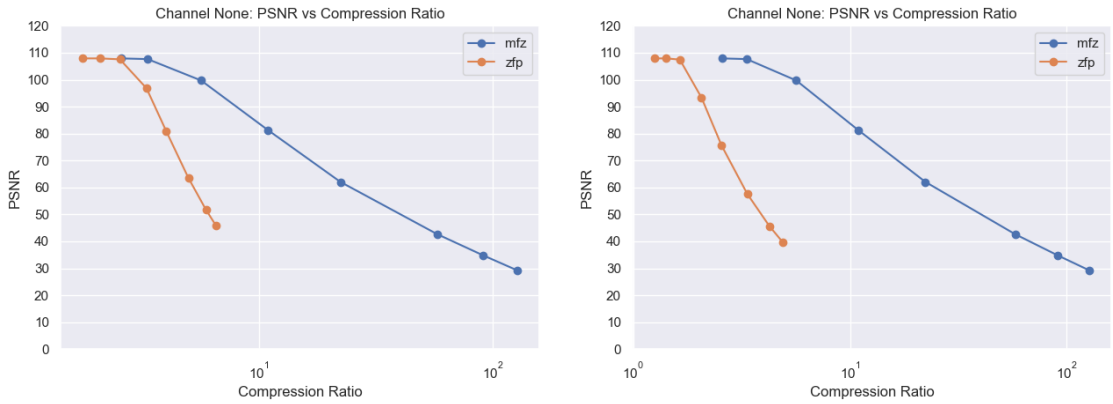
(d) **Shuffled** Streamwise (X) Velocity

Figure 4.9: Demonstrating the effect of shuffling the data order, for turbulent flow data. In the plots, higher PSNR is better.

We can see that the SZ outperforms ZFP and MFZ in both unshuffled cases, Fig. 4.9a and Fig. 4.9c. Given SZ3's complex internals, it is difficult to understand exactly why this is the case. At a high level, the models SZ is applying describe the data very well. ZFP and MFZ have very similar performance in this dataset, with MFZ having a slight edge in the Y velocity. We see again that shuffling the data, as in Fig. 4.9b and Fig. 4.9d, makes MFZ moderately better than SZ and ZFP, especially in the low compression / high PSNR regime.

4.3.3 Neuron Transport

The neuron transport dataset is a simulation of the action potential of a membrane channel which starts as a point mass of action potential in a 3D membrane channel. This mesh is non-Cartesian, although spatially uniform in the meshed regions. It consists of 600 timesteps and 116,543 points, and is much larger than the other datasets seen in this chapter. MFZ applies the same collocation method to generate a MHB in 3D, and extends its block size from roughly 8×8 to roughly $8 \times 8 \times 8$.[3]



(a) Action Potential

(b) **Shuffled** Action Potential

Figure 4.10: Demonstrating the effect of shuffling the data order, for neuron transport data. In the plots, higher PSNR is better.

The layout of the neuron transport dataset in memory naturally has much less spatial correlation in adjacent memory components. This means that the unshuffled data is much less compressible for ZFP, as seen in Fig. 4.10a, but clearly there is still some spatial correlation since the performance of ZFP is degraded with shuffling Fig. 4.10b. This illustrates a case where the memory layout of PDE data loses its spatial correlation and becomes much less compressible for standard compressors but not for MFZ.

[3] We are not able to present SZ's performance on this dataset, due to a bug in the SZ results that produced abnormally large errors.

4.3.4 Block Size

The block size is frequently a fixed parameter in block coders, since the implementation of the block coder does not always allow for the block size to be changed easily. In MFZ, the block size is a runtime parameter that affects the speed of the compression and the compression factor. For MFZ we have chosen the block size to be approximately $8 \times \cdots \times 8$ and be uniformly sized across the data. Algorithms such as JPEG 2000 [70] use blocks of different sizes, depending on how well it compresses in each size. ZFP uses slightly smaller blocks (4 elements per dimension), whereas SPERR uses much larger blocks (256 elements per dimension). To illustrate how block sizes can affect compression, consider the ignition data seen in Section 4.3.1 which has a large discontinuity at the boundary of the oxygen jet, and resulting jet flame. At large block sizes this discontinuity would be present in every block, whereas small block sizes create many highly compressible blocks and some blocks with a discontinuity. Ultimately, the optimal block size is a function of the data. A more advanced variant of MFZ may vary the block size or fit it to the data and store the block size with the file, but our goal here was to demonstrate the advantages of a block coder that uses the MHB and so it has not been a focus of this research.

4.3.5 Drawbacks

There are a number of drawbacks to MFZ as presented here, which make it less effective than it could otherwise be. Firstly, we do not compress across blocks, time nor channels. This leaves global patterns, temporal correlation and inter-channel correlation uncompressed. Second, the 32-bit integer range that we compress into (during range encoding) is large for the message sizes since we may not be using the entire information capacity of all of the 32-bit integers, but the convenience of the "constriction" package has made implementation much easier. This leaves some overhead in our lossless compression scheme. Third, our representation for all zero blocks is inefficient compared to something like ZFP. This means

that for data which contains a larger number of zero blocks, our performance suffers slightly. Finally, the code has been implemented in Python, which is slow compared to state of the art compression software written in C++. For example the MFZ compressor is 50× slower than ZFP, owing mostly to inefficiencies in array creation, type casting and bit string manipulation in Python. A better implementation of MFZ could approach the performance of ZFP, aside from the MHB creation and a single floating point matrix-vector multiplication (ZFP uses a special DCT, implemented only with bit shifts). The MHB creation consists of many small generalized eigenvector problems, which can be parallelized and are only done once at the beginning of a compression task. We believe that a better implementation would make this a useful compression tool for scientists working with data over meshes.

4.4 Conclusion

We have presented a block coder data compression tool that has been designed specifically for non-uniform meshes. We accomplish this by partitioning the mesh with METIS, generating the Manifold Harmonic Basis for each block, and using a set of data compression steps that are used in ZFP [64]. This compressor shows improvement over ZFP on non-uniform meshes in 2D, and in 3D the improvement is even larger. We leverage a collocation method to help to generate the MHB faster, and in arbitrary dimensions. We also use classical data compression tools to adapt to some of the changes the MHB necessitates in the compression chain. This results in an advancement of the block coder style compressors and an improvement in the compression of scientific data on non-uniform domains.

Chapter 5

Conclusion

This thesis has covered numerical methods for the building linear and non-linear operators for non-uniform data sources, with the goal of exploiting spatio-temporal correlation to create compressed representations of data for storage.

In Chapter 1, we covered a neural network layer called QuadConv [21] that allows for the construction of convolutional neural network architectures over non-uniform data sources. We tested this layer on a data compression problem and showed that it performs better than existing neural network layers.

In Chapter 2, we extended max pooling to non-uniform data sources using mesh downsampling techniques [58] and improved upon QuadConv. We made QuadConv converge faster, compute faster and provide more flexibility when considering parameter and accuracy budgets for practical problems. We also made QuadConv easier to work with and demonstrated how to create typical convolutional neural network architectures using this improved interface (Appendix B).

Finally in Chapter 3, we used a collocation technique on a generalized eigenvalue problem in order to derive a Manifold Harmonic Basis. This allowed us to generate the MHB in 2D, 3D and beyond with faster runtimes than the popular FEM technique. Using this method we created Mesh-Float-Zip, an error-bounded compressor for non-uniform data sources that leverages the MHB in the same way that JPEG [111] or ZFP [64] uses the discrete cosine transform. This technique surpassed state of the art compressors on data

sets whose memory layout did not match their spatial correlation.

We believe there are a number of interesting directions for future research, specifically in the compression regime that exists between neural networks and classical compressors. Compressors in this regime have the opportunity to leverage some runtime to learn about the data, before making high accuracy compressed representations. We believe leveraging the lessons from both the neural methods (which has learned non-linear maps to exploit data correlation) and the classical compressors (which use spectral structure and floating point compression tricks) will lead to interesting and effective solutions.

Bibliography

[1] Martín Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: **arXiv preprint arXiv:1603.04467** (2016).

[2] Nasir Ahmed, T Natarajan, and Kamisetty R Rao. "Discrete cosine transform". In: **IEEE transactions on Computers** 100.1 (1974), pp. 90–93.

[3] Kendall Atkinson. **An introduction to numerical analysis**. John wiley & sons, 1991.

[4] Grey Ballard, Alicia Klinvex, and Tamara G Kolda. "TuckerMPI: A parallel C++/MPI software package for large-scale data compression via the Tucker tensor decomposition". In: **ACM Transactions on Mathematical Software (TOMS)** 46.2 (2020), pp. 1–31.

[5] Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. "TTHRESH: Tensor compression for multidimensional visual data". In: **IEEE transactions on visualization and computer graphics** 26.9 (2019), pp. 2891–2903.

[6] Robert Bamler. "Understanding Entropy Coding With Asymmetric Numeral Systems (ANS): a Statistician's Perspective". In: **arXiv preprint arXiv:2201.01741** (2022).

[7] Mikhail Belkin, Jian Sun, and Yusu Wang. "Constructing Laplace operator from point clouds in $\mathbb{R}$ d". In: **Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms**. SIAM. 2009, pp. 1031–1040.

[8] G Berkooz, P Holmes, and J L Lumley. "The Proper Orthogonal Decomposition in the Analysis of Turbulent Flows". In: **Annual Review of Fluid Mechanics** 25.1 (1993), pp. 539–575. DOI: 10.1146/annurev.fl.25.010193.002543.

[9] Elwyn R Berlekamp. "Block coding with noiseless feedback". PhD thesis. Massachusetts Institute of Technology, 1964.

[10] Alexandre Boulch. "ConvPoint: Continuous convolutions for point cloud processing". In: **Computers & Graphics** 88 (2020), pp. 24–34. ISSN: 0097-8493. DOI: 10.1016/j.cag.2020.02.005.

[11] John Burkardt. **Airplane**. Generated from 3D model. 2012. (Visited on 11/08/2022).

[12] C Sidney Burrus, Ramesh A Gopinath, and Haitao Guo. "Wavelets and wavelet transforms". In: **rice university, houston edition** 98 (1998).

[13] Tony F Chan, Jinchao Xu, and Ludmil Zikatanov. "An agglomeration multigrid method for unstructured grids". In: **Contemporary Mathematics** 218 (1998), pp. 67–81.

[14] Jieyang Chen et al. "Understanding performance-quality trade-offs in scientific visualization workflows with lossy compression". In: **2019 IEEE/ACM 5th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-5)**. IEEE. 2019, pp. 1–7.

[15] Dario Coscia et al. **A Continuous Convolutional Trainable Filter for Modelling Unstructured Data**. 2022. DOI: 10.48550/ARXIV.2210.13416.

[16] Germund Dahlquist and Åke Björck. **Numerical methods in scientific computing, volume I**. SIAM, 2008.

[17] Sheng Di and Franck Cappello. "Fast error-bounded lossy HPC data compression with SZ". In: **2016 ieee international parallel and distributed processing symposium (ipdps)**. IEEE. 2016, pp. 730–739.

[18] James Diffenderfer et al. “Error analysis of zfp compression for floating-point data”. In: **SIAM Journal on Scientific Computing** 41.3 (2019), A1867–A1898.

[19] Kevin Doherty and Cooper Simpson. **PyTorch-QuadConv**. `github.com/AlgorithmicDataReduct` `PyTorch-QuadConv`. Quadrature-based convolutions for deep learning. 2023.

[20] Kevin Doherty and Cooper Simpson. **QuadConv**. `github.com/kvndhrty/QuadConv`. 2022.

[21] Kevin Doherty et al. “QuadConv: Quadrature-based convolutions with applications to non-uniform PDE data compression”. In: **Journal of Computational Physics** 498 (2024), p. 112636.

[22] Wilna Du Toit. “Radial basis function interpolation”. PhD thesis. Stellenbosch: Stellenbosch University, 2008.

[23] Jarek Duda. “Asymmetric numeral systems”. In: **arXiv preprint arXiv:0902.0271** (2009).

[24] Alec M Dunton et al. “Pass-efficient methods for compression of high-dimensional turbulent flow data”. In: **Journal of Computational Physics** 423 (2020), p. 109704.

[25] William Falcon and The PyTorch Lightning team. **PyTorch Lightning**. Version 1.7. 2019. DOI: `10.5281/zenodo.3828935`.

[26] Robert M Fano. **The transmission of information**. Vol. 65. Massachusetts Institute of Technology, Research Laboratory of Electronics ..., 1949.

[27] Matthias Fey and Jan E. Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: **ICLR Workshop on Representation Learning on Graphs and Manifolds**. 2019.

[28] Matthias Fey et al. “Splinecnn: Fast geometric deep learning with continuous b-spline kernels”. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. 2018, pp. 869–877.

[29] Michael S Floater and Kai Hormann. "Surface parameterization: a tutorial and survey". In: **Advances in multiresolution for geometric modelling** (2005), pp. 157–186.

[30] Alyson Fox et al. "Stability analysis of inline ZFP compression for floating-point data in iterative methods". In: **SIAM Journal on Scientific Computing** 42.5 (2020), A2701–A2730.

[31] Carsten Franke and Robert Schaback. "Solving partial differential equations by collocation using radial basis functions". In: **Applied Mathematics and Computation** 93.1 (1998), pp. 73–82.

[32] Hongyang Gao and Shuiwang Ji. "Graph u-nets". In: **international conference on machine learning**. PMLR. 2019, pp. 2083–2092.

[33] Andrew Glaws, Ryan King, and Michael Sprague. "Deep learning for in situ data compression of large turbulent flow simulations". In: **Phys. Rev. Fluids** 5 (11 Nov. 2020), p. 114602. DOI: 10.1103/PhysRevFluids.5.114602.

[34] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: **Proceedings of the thirteenth international conference on artificial intelligence and statistics**. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[35] Gene H Golub and Charles F Van Loan. **Matrix computations**. JHU press, 2013.

[36] Qian Gong et al. "MGARD: A multigrid framework for high-performance, error-controlled data compression and refactoring". In: **SoftwareX** 24 (2023), p. 101590.

[37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. **Deep Learning**. http://www.deeplearningbook.org. MIT Press, 2016.

[38] D. S. Grebenkov and B.-T. Nguyen. "Geometrical Structure of Laplacian Eigenfunctions". In: **SIAM Review** 55.4 (2013), pp. 601–667. DOI: 10.1137/120880173. eprint: https://doi.org/10.1137/120880173. URL: https://doi.org/10.1137/120880173.

[39] Yulan Guo et al. "Deep Learning for 3D Point Clouds: A Survey". In: **IEEE Transactions on Pattern Analysis and Machine Intelligence** 43.12 (2021), pp. 4338–4364. DOI: 10.1109/TPAMI.2020.3005434.

[40] Mahmoud Habboush et al. "DE-ZFP: An FPGA implementation of a modified ZFP compression/decompression algorithm". In: **Microprocessors and Microsystems** 90 (2022), p. 104453.

[41] Rana Hanocka et al. "Meshcnn: a network with an edge". In: **ACM Transactions on Graphics (TOG)** 38.4 (2019), pp. 1–12.

[42] Bingsheng He et al. "Efficient gather and scatter operations on graphics processors". In: **Proceedings of the 2007 ACM/IEEE Conference on Supercomputing**. 2007, pp. 1–12.

[43] Kaiming He et al. "Deep residual learning for image recognition". In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. 2016, pp. 770–778.

[44] MA Hernández. "Chebyshev's approximation algorithms and applications". In: **Computers & Mathematics with Applications** 41.3-4 (2001), pp. 433–445.

[45] David A Huffman. "A method for the construction of minimum-redundancy codes". In: **Proceedings of the IRE** 40.9 (1952), pp. 1098–1101.

[46] Lawrence Ibarria et al. "Out-of-core compression and decompression of large n-dimensional scalar fields". In: **Computer Graphics Forum**. Vol. 22. 3. Wiley Online Library. 2003, pp. 343–348.

[47] "IEEE Standard for Binary Floating-Point Arithmetic". In: **ANSI/IEEE Std 754-1985** (1985), pp. 1–20. DOI: 10.1109/IEEESTD.1985.82928.

[48] William Kahan. "IEEE standard 754 for binary floating-point arithmetic". In: **Lecture Notes on the Status of IEEE** 754.94720-1776 (1996), p. 11.

[49] Edward J Kansa. "Multiquadrics—A scattered data approximation scheme with applications to computational fluid-dynamics—II solutions to parabolic, hyperbolic and elliptic partial differential equations". In: **Computers & mathematics with applications** 19.8-9 (1990), pp. 147–161.

[50] Azam Karami, Mehran Yazdi, and Grégoire Mercier. "Compression of hyperspectral images using discerete wavelet transform and tucker decomposition". In: **IEEE journal of selected topics in applied earth observations and remote sensing** 5.2 (2012), pp. 444–450.

[51] Zachi Karni and Craig Gotsman. "Spectral compression of mesh geometry". In: **Proceedings of the 27th annual conference on Computer graphics and interactive techniques**. 2000, pp. 279–286.

[52] George Karypis and Vipin Kumar. "METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices". In: (1997).

[53] Sanghyeon Kim and Eunbyung Park. "Smpconv: Self-moving point representations for continuous convolution". In: **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**. 2023, pp. 10289–10299.

[54] Diederik P. Kingma and Jimmy Ba. **Adam: A Method for Stochastic Optimization**. 2014. DOI: 10.48550/ARXIV.1412.6980.

[55] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: **arXiv preprint arXiv:1609.02907** (2016).

[56] Artem Komarichev, Zichun Zhong, and Jing Hua. **A-CNN: Annularly Convolutional Neural Networks on Point Clouds**. 2019. DOI: 10.48550/ARXIV.1904.08017.

[57] Glen G Langdon. "An introduction to arithmetic coding". In: **IBM Journal of Research and Development** 28.2 (1984), pp. 135–149.

[58] Andrew P Lawrence, Morten E Nielsen, and Bengt Fornberg. "Node subsampling for multilevel meshfree elliptic PDE solvers". In: **Computers & Mathematics with Applications** 164 (2024), pp. 79–94.

[59] Kookjin Lee and Kevin T. Carlberg. "Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders". In: **Journal of Computational Physics** 404 (2020), p. 108973. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2019.108973.

[60] Bruno Lévy. "Laplace-beltrami eigenfunctions towards an algorithm that" understands" geometry". In: **IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)**. IEEE. 2006, pp. 13–13.

[61] Shaomeng Li, Peter Lindstrom, and John Clyne. "Lossy Scientific Data Compression With SPERR". In: **2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. 2023, pp. 1007–1017. DOI: 10.1109/IPDPS54959.2023.00104.

[62] Xin Liang et al. "Sz3: A modular framework for composing prediction-based error-bounded lossy compressors". In: **IEEE Transactions on Big Data** 9.2 (2022), pp. 485–498.

[63] Peter Lindstrom. **Error distributions of lossy floating-point compressors**. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[64] Peter Lindstrom. "Fixed-Rate Compressed Floating-Point Arrays". In: **IEEE Transactions on Visualization and Computer Graphics** 20 (Aug. 2014). DOI: 10.1109/TVCG.2014.2346458.

[65] Peter Lindstrom and Martin Isenburg. "Fast and Efficient Compression of Floating-Point Data". In: **IEEE transactions on visualization and computer graphics** 12 (Sept. 2006), pp. 1245–50. DOI: 10.1109/TVCG.2006.143.

[66] Chuang Liu et al. "Graph pooling for graph neural networks: Progress, challenges, and opportunities". In: **arXiv preprint arXiv:2204.07321** (2022).

[67] Jinyang Liu et al. "Improving lossy compression for sz by exploring the best-fit lossless compression techniques". In: **2021 IEEE International Conference on Big Data (Big Data)**. IEEE. 2021, pp. 2986–2991.

[68] Jinyang Liu et al. "SRN-SZ: Deep Leaning-Based Scientific Error-bounded Lossy Compression with Super-resolution Neural Networks". In: **arXiv preprint arXiv:2309.04037** (2023).

[69] Yang Liu, Balakrishnan Prabhakaran, and Xiaohu Guo. "Point-based manifold harmonics". In: **IEEE Transactions on visualization and computer graphics** 18.10 (2012), pp. 1693–1703.

[70] Michael W Marcellin et al. "An overview of JPEG-2000". In: **Proceedings DCC 2000. Data compression conference**. IEEE. 2000, pp. 523–541.

[71] G Nigel N Martin. "Range encoding: an algorithm for removing redundancy from a digitised message". In: **Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording**. Vol. 2. 1979.

[72] Russell Merris. "A survey of graph Laplacians". In: **Linear and Multilinear Algebra** 39.1-2 (1995), pp. 19–31.

[73] Charles A Micchelli. **Interpolation of scattered data: distance matrices and conditionally positive definite functions**. Springer, 1984.

[74] Mohammadreza Momenifar et al. "A Physics-Informed Vector Quantized Autoencoder for Data Compression of Turbulent Flow". In: **2022 Data Compression Conference (DCC)**. IEEE, Mar. 2022, pp. 01–10. DOI: 10.1109/dcc52660.2022.00039.

[75] Mohammadreza Momenifar et al. **Emulating Spatio-Temporal Realizations of Three-Dimensional Isotropic Turbulence via Deep Sequence Learning Models**. 2021. DOI: 10.48550/ARXIV.2112.03469.

[76] Jean-Michel Muller et al. **Handbook of Floating-Point Arithmetic**. 2nd. Birkhäuser Basel, 2018. ISBN: 3319765256.

[77] Jawad Nagi et al. "Max-pooling convolutional neural networks for vision-based hand gesture recognition". In: **2011 IEEE international conference on signal and image processing applications (ICSIPA)**. IEEE. 2011, pp. 342–347.

[78] Eric Nguyen et al. **S4ND: Modeling Images and Videos as Multidimensional Signals Using State Spaces**. 2022. DOI: 10.48550/ARXIV.2210.06583.

[79] Harry Nyquist. "Certain factors affecting telegraph speed". In: **Transactions of the American Institute of Electrical Engineers** 43 (1924), pp. 412–422.

[80] Keiron O'shea and Ryan Nash. "An introduction to convolutional neural networks". In: **arXiv preprint arXiv:1511.08458** (2015).

[81] Heather Pacella et al. "Task-parallel in situ temporal compression of large-scale computational fluid dynamics data". In: **The International Journal of High Performance Computing Applications** 36.3 (2022), pp. 388–418.

[82] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: **Advances in Neural Information Processing Systems 32**. Curran Associates, Inc., 2019, pp. 8024–8035. DOI: 10.48550/ARXIV.1912.01703.

[83] Allan Pinkus. "Approximation theory of the MLP model in neural networks". In: **Acta numerica** 8 (1999), pp. 143–195.

[84] Charles R. Qi et al. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: **2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. July 2017, pp. 77–85.

[85] Rajat Raina, Anand Madhavan, and Andrew Y Ng. "Large-scale deep unsupervised learning using graphics processors". In: **Proceedings of the 26th annual international conference on machine learning**. 2009, pp. 873–880.

[86] Anurag Ranjan et al. "Generating 3D faces using convolutional mesh autoencoders". In: **Proceedings of the European conference on computer vision (ECCV)**. 2018, pp. 704–720.

[87] Kamisetty Ramam Rao and Patrick C Yip. **The transform and data compression handbook**. CRC press, 2018.

[88] Junuthula Narasimha Reddy. "An introduction to the finite element method". In: **New York** 27 (1993), p. 14.

[89] David W Romero et al. **Towards a General Purpose CNN for Long Range Dependencies in** N*D*. 2022. eprint: 2206.03398.

[90] David W. Romero et al. **CKConv: Continuous Kernel Convolution For Sequential Data**. 2022. eprint: 2102.02611.

[91] David W. Romero et al. **FlexConv: Continuous Kernel Convolutions with Differentiable Kernel Sizes**. 2022. eprint: 2110.08059.

[92] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: **nature** 323.6088 (1986), pp. 533–536.

[93] V. Sanchez et al. “Diagonalizing properties of the discrete cosine transforms”. In: **IEEE Transactions on Signal Processing** 43.11 (1995), pp. 2631–2641. DOI: 10.1109/78.482113.

[94] Nico Schlömer and J. Hariharan. **dmsh**. https://github.com/meshpro/dmsh. Inspired by distmesh. 2022.

[95] Claude Elwood Shannon. “A mathematical theory of communication”. In: **The Bell system technical journal** 27.3 (1948), pp. 379–423.

[96] Martin Simonovsky and Nikos Komodakis. “Dynamic edge-conditioned filters in convolutional neural networks on graphs”. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. 2017, pp. 3693–3702.

[97] Vincent Sitzmann et al. “Implicit neural representations with periodic activation functions”. In: **Advances in neural information processing systems** 33 (2020), pp. 7462–7473.

[98] Ryan W Skinner et al. “Reduced-basis multifidelity approach for efficient parametric study of naca airfoils”. In: **AIAA Journal** 57.4 (2019), pp. 1481–1491.

[99] Olga Sorkine. “Laplacian mesh processing”. In: **Eurographics (State of the Art Reports)** 4.4 (2005), p. 1.

[100] Stanford University Computer Graphics Laboratory. **Stanford Bunny**. Generated from 3D model. 1994. (Visited on 11/08/2022).

[101] Karen Stengel et al. “Adversarial super-resolution of climatological wind and solar data”. In: **Proceedings of the National Academy of Sciences** 117.29 (2020), pp. 16805–16815.

[102] Gilbert Strang. “The discrete cosine transform”. In: **SIAM review** 41.1 (1999), pp. 135–147.

[103] Julian Suk et al. “Mesh Neural Networks for SE (3)-Equivariant Hemodynamics Estimation on the Artery Wall”. In: **arXiv preprint arXiv:2212.05023** (2022).

[104] Haotian Tang et al. “Searching Efficient 3D Architectures with Sparse Point-Voxel Convolution”. In: **Computer Vision – ECCV 2020**. Ed. by Andrea Vedaldi et al. Cham: Springer International Publishing, 2020, pp. 685–702. DOI: 10.1007/978-3-030-58604-1_41.

[105] Gabriel Taubin. “A signal processing approach to fair surface design”. In: **Proceedings of the 22nd annual conference on Computer graphics and interactive techniques**. 1995, pp. 351–358.

[106] G TaubinÝ. “Geometric signal processing on polygonal meshes”. In: **Proceedings of EUROGRAPHICS**. 2000.

[107] David Taubman et al. “Embedded block coding in JPEG 2000”. In: **Signal Processing: Image Communication** 17.1 (2002), pp. 49–72.

[108] trends.google.com. **Google Trends**. 2012. URL: http://trends.google.com/trends.

[109] Stephan Trenn. “Multilayer perceptrons: Approximation order and necessary number of hidden units”. In: **IEEE transactions on neural networks** 19.5 (2008), pp. 836–844.

[110] Bruno Vallet and Bruno Lévy. “Spectral geometry processing with manifold harmonics”. In: **Computer Graphics Forum**. Vol. 27. 2. Wiley Online Library. 2008, pp. 251–260.

[111] Gregory K Wallace. “The JPEG still picture compression standard”. In: **IEEE transactions on consumer electronics** 38.1 (1992), pp. xviii–xxxiv.

[112] Shenlong Wang et al. “Deep parametric continuous convolutional neural networks”. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. 2018, pp. 2589–2597.

[113] Max Wardetzky et al. “Discrete Laplace operators: no free lunch”. In: **Symposium on Geometry processing**. Vol. 33. Aire-la-Ville, Switzerland. 2007, p. 37.

[114] Grady Barrett Wright. **Radial basis function interpolation: numerical and analytical developments**. University of Colorado at Boulder, 2003.

[115] Wenxuan Wu, Zhongang Qi, and Li Fuxin. “PointConv: Deep Convolutional Networks on 3D Point Clouds”. In: **2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)**. June 2019, pp. 9613–9622.

[116] Zhirong Wu et al. “3D ShapeNets: A deep representation for volumetric shapes”. In: **2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. Los Alamitos, CA, USA: IEEE Computer Society, June 2015, pp. 1912–1920. DOI: 10.1109/CVPR.2015.7298801.

[117] Zonghan Wu et al. “A Comprehensive Survey on Graph Neural Networks”. In: **IEEE Transactions on Neural Networks and Learning Systems** 32.1 (Jan. 2021), pp. 4–24. DOI: 10.1109/tnnls.2020.2978386.

[118] Yifan Xu et al. “SpiderCNN: Deep Learning on Point Sets with Parameterized Convolutional Filters”. In: **Computer Vision – ECCV 2018**. Ed. by Vittorio Ferrari et al. Cham: Springer International Publishing, 2018, pp. 90–105. ISBN: 978-3-030-01237-3.

[119] Xiaodong Yu et al. “SZX: An ultra-fast error-bounded lossy compressor for scientific datasets”. In: **arXiv preprint arXiv:2201.13020** (2022).

[120] Christoph 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.

Appendix A

Quadrature Convolutions

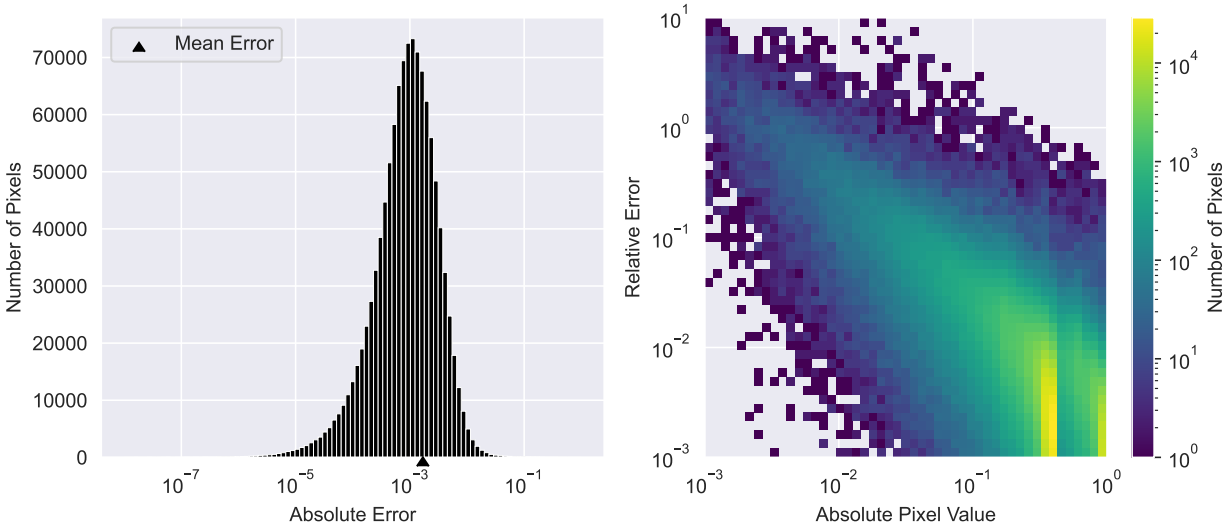


Figure A.1: Error analysis of GCN for the uniform grid ignition data.

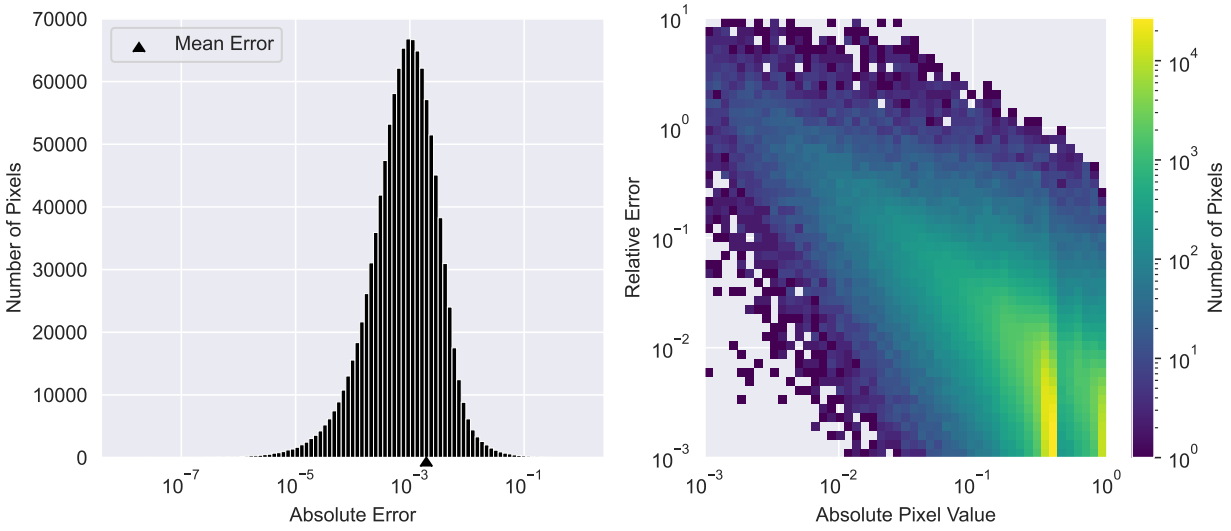


Figure A.2: Error analysis of SplineCNN for the uniform grid ignition data.

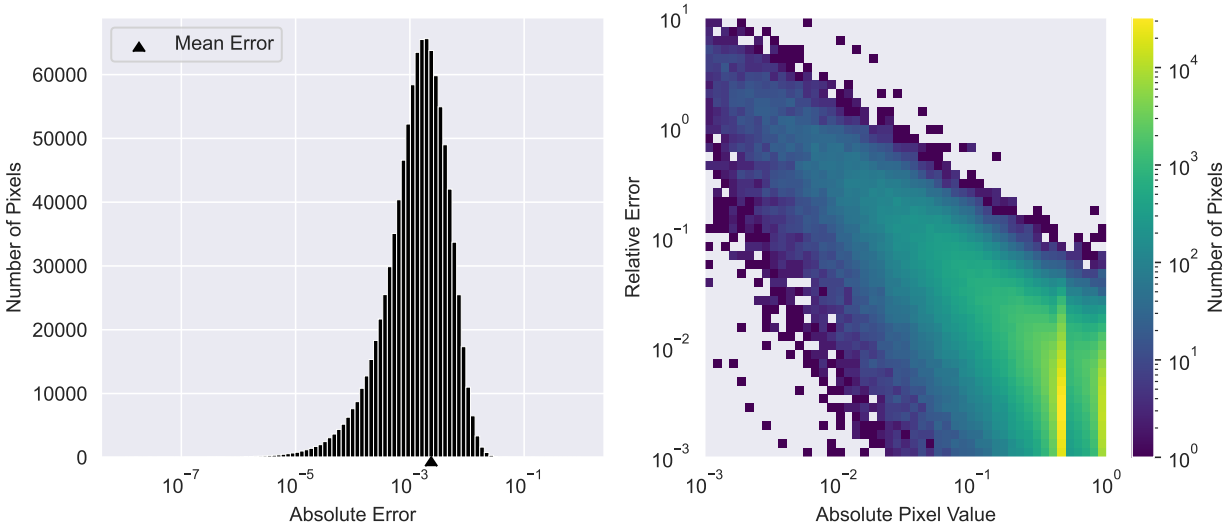


Figure A.3: QCNN error analysis of non-uniform ignition data.

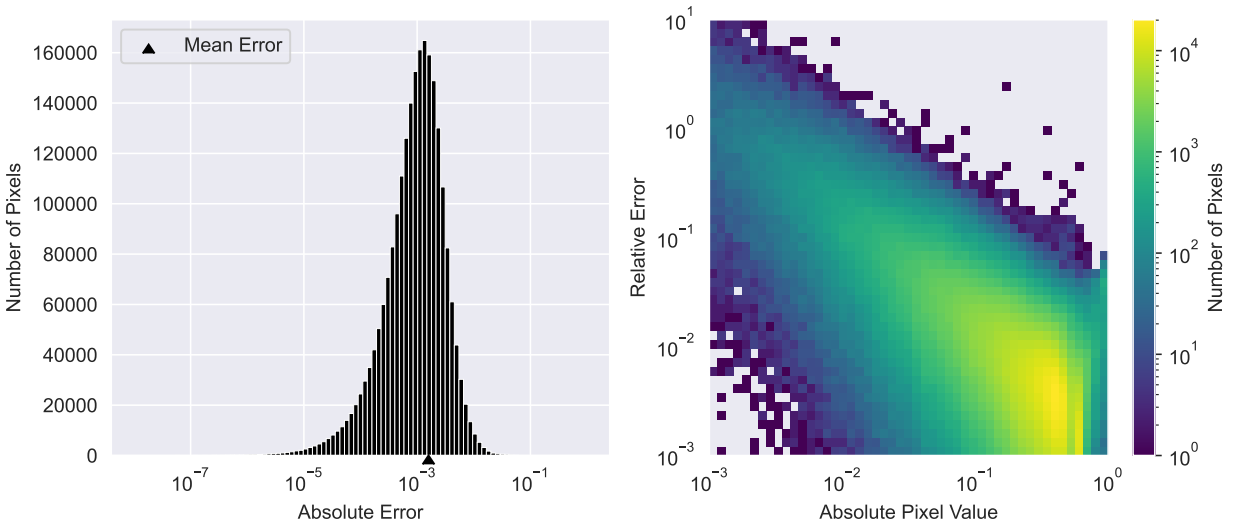


Figure A.4: QCNN error analysis of non-uniform flow data.

Appendix B

Refining Quadrature Convolutions

A `ResNetBlock`, implemented on an arbitrary geometry with QuadConv and `Domain` objects.

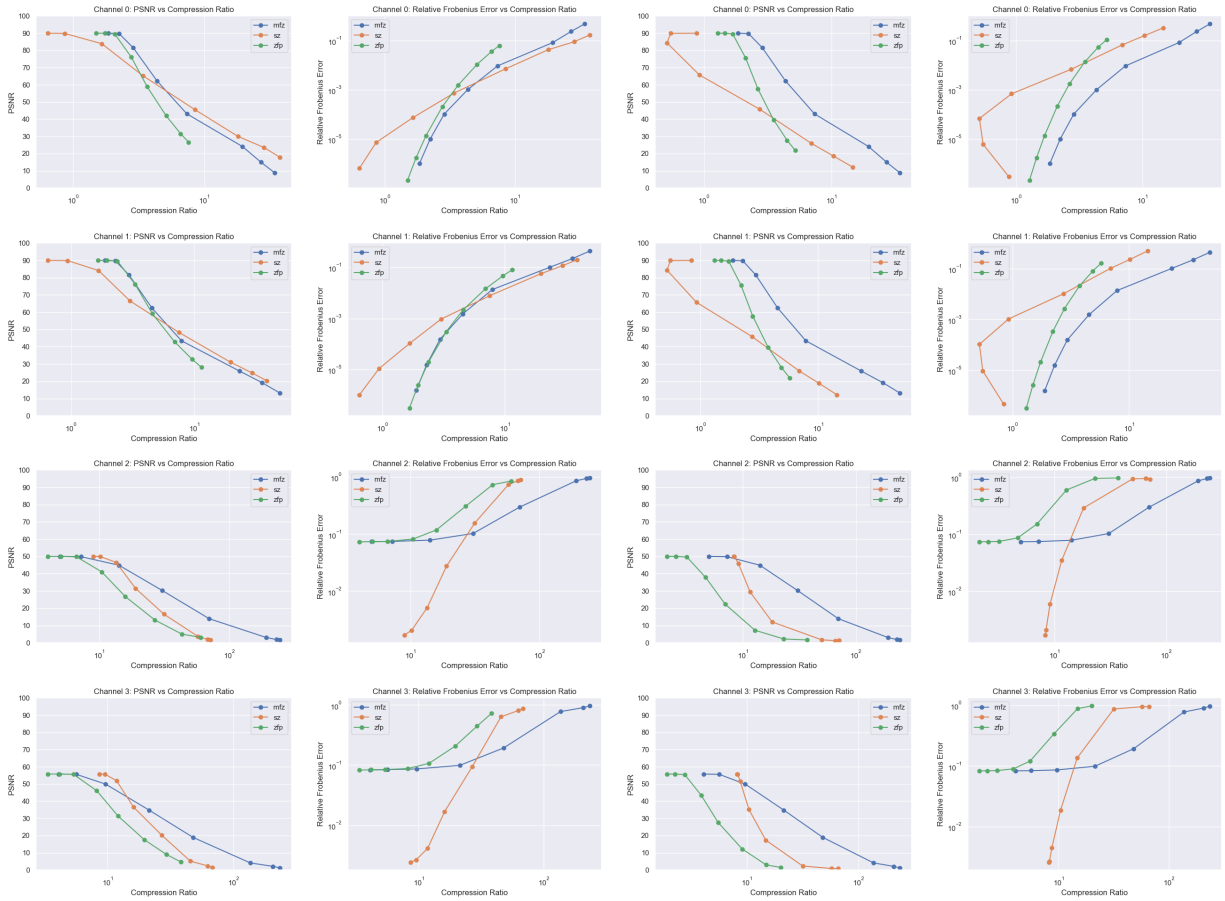
```
import torch
import torch.nn as nn

from torch_quadconv import QuadConv, Mesh_MaxPool, Mesh_AvgPool, Grid

class ResNetBlock(nn.Module):
    def __init__(self, domain, c_in, act_fn=nn.ReLU, subsample=False, c_out=-1):
        """
        Inputs:
            c_in - Number of input features
            act_fn - Activation class constructor (e.g. nn.ReLU)
            subsample - If True, we want to apply a "stride" inside the block
                        and reduce the output shape by 2 in height and width (spatial domain)
            c_out - Number of output features.
                Note that this is only relevant if subsample is True, as otherwise, c_out = c_in
        """
        super().__init__()
        if not subsample:
            c_out = c_in
        else:
            c_out = c_in*4 if c_out == -1 else c_out

        self.block_domain = domain
        self.block_range = domain if not subsample else domain.downsample(factor=2)
        self.conv1 = QuadConv(domain = self.block_domain,
                              range = self.block_range,
                              in_channels = c_in,
                              out_channels = c_out,
                              output_same=True if not subsample else False,
                              bias=False)  # No bias needed as the Batch Norm handles it

        self.bnorm1 = nn.BatchNorm1d(c_out)
        self.conv2 = QuadConv(domain = self.block_range,
                              range = self.block_range,
                              in_channels = c_out,
                              out_channels = c_out,
                              output_same=True,
```

```
                                    bias = False) # No bias needed as the Batch Norm handles it

            self.bnorm2 = nn.BatchNorm1d(c_out)

            #Transform
            self.downsample = Mesh_AvgPool(self.block_domain.pool_map(kernel_size = 2))
                                if subsample else None
            self.expand_downsample = nn.Conv1d(c_in, c_out, kernel_size=1, stride=1)
                                if subsample else None

            self.act_fn = act_fn()

        def forward(self, x):

            z = self.conv1(x)
            z = self.act_fn(self.bnorm1(z))
            z = self.conv2(z)
            z = self.bnorm2(z)

            if self.downsample is not None:
                x = self.downsample(x)
                x = self.expand_downsample(x)

            out = z + x
            out = self.act_fn(out)
            return out

```

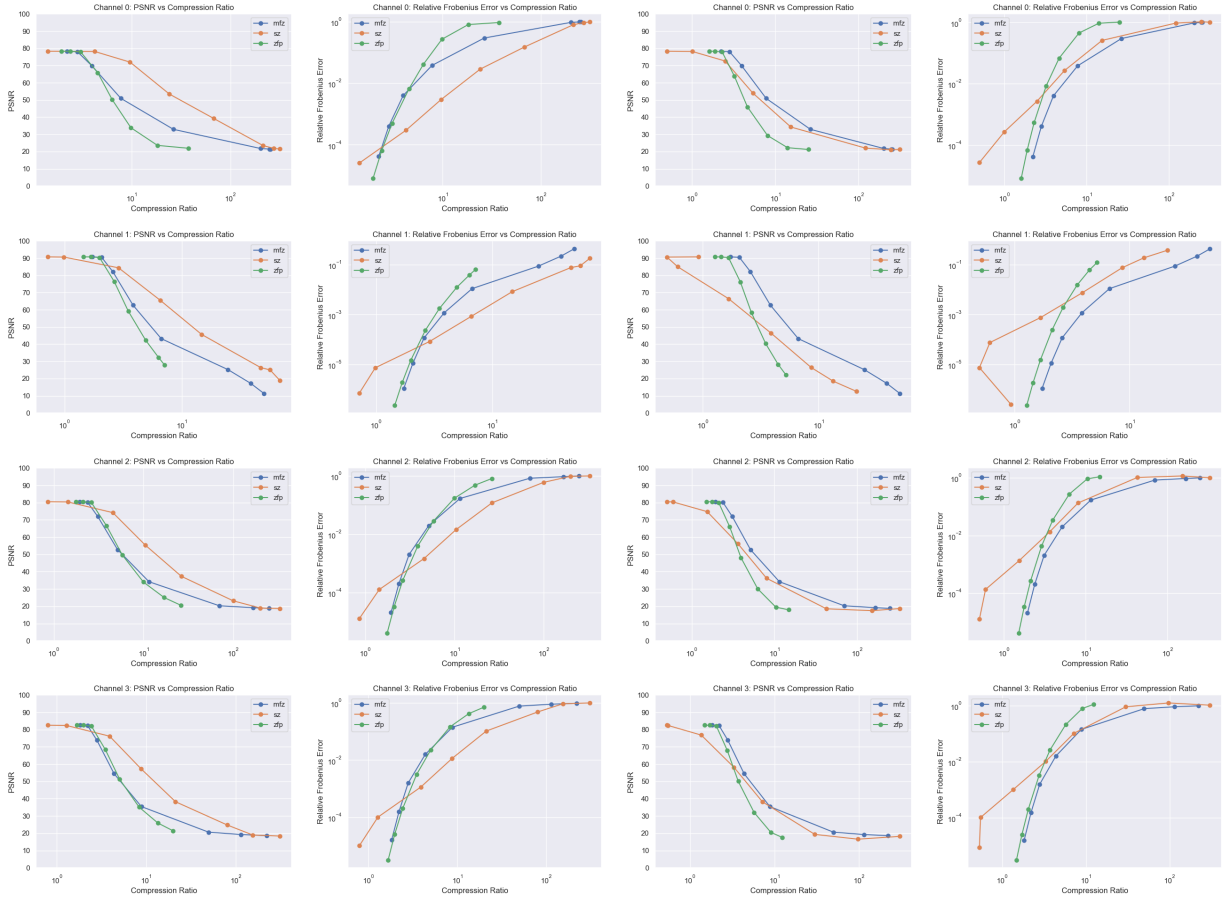
Appendix C

Mesh-Float-Zip

C.0.1 Ignition Mesh: Full Results



C.0.2 Flat Plate: Full Results



C.0.3 Neuron Transport: Full Results