# Large-scale lossy data compression based on an a priori error estimator in a Spectral Element Code

Oana Marin[a], Michel Schanen[a], Paul Fischer[b]

[a] *Mathematics and Computer Science Division, Argonne National Laboratory*
[b] *University of Illinois at Urbana Champaign*

**Abstract**

An increasing bottleneck for high performance computing and large-scale simulations is the I/O of supercomputers. We present a highly scalable lossy data compression algorithm with controllable error based on an *a priori* estimator, applied to complex curvilinear meshes. Integral transforms, such as, the the discrete Chebyshev transform (DCT) or discrete Legendre transform (DLT) due to their energy compactness property can compress a signal optimally. In these approaches the data field is mapped to spectral space and subsequently truncated within a user-specified tolerance which ensures that the recovered signal is below a desired error threshold. The truncated data field is compressed for high I/O speed and low storage using bitwise encoding. The method presented here is endowed with an a priori error estimator to allow for a dynamic compression ratio which can be as low as 3% compression for data visualization even in cases as complex as fully developed turbulent flow. For checkpointing problems however, the compression ratio is problem dependent. We also derived an orthogonal transform compatible with the spectral discretization on Gauss-Legendre-Lobatto curvilinear grids, which displays superior performance to the traditional DCT. The algorithm is implemented in the spectral-element code Nek5000 and tested on highly turbulent large-scale simulation data of up to 3.2 billion degrees of freedom. The implementation via tensor products is highly efficient, reducing the flops in matrix-vector multiplications from $\mathcal{O}(N^{2d})$ down to $\mathcal{O}(N^{d+1})$ in a d-dimensional space.

*Keywords:* Large-scale, I/O, High performance computing, discrete Legendre transform, discrete Chebyshev transform, bitwise encoding, lossy compression, computational fluid dynamics, error estimation

*Corresponding author
*Email address:* oanam@anl.gov (Oana Marin)

## 1. Introduction

Handling and storing the large data sets generated by massively parallel simulations forms a significant part of computation difficulties. Performance at the various levels of the memory hierarchy is steadily diverging, rendering computations I/O bound and memory bound. This work focuses on reducing the amount of data that has to be written to disk without effecting the usefulnnes of the data by applying lossy data compression. The compression ratio is selected optimally through the implementation of an a priori error estimator. We explore two integral transforms, the discrete Legendre transform and the discrete Chebyshev transform, both with a high energy compactness property.

Compression algorithms can be classified as lossy, where data loss is allowed within given bounds, and lossless, where no error is incurred but the space savings are lower. Transforming the data in spectral space where the high energy modes are compacted in a few terms allows for truncation at low frequencies. A notorious transform in spectral space, which has been widely used for compression in image processing is the discrete Chebyshev transform (DCT). This approach has been the basic algorithm for JPEG image compression since the early 1970s [1]. The orthogonal transforms based strategy are lossy compression algorithms since low frequency modes are truncated in spectral space. More recently, the same ideas were introduced in computational fluid dynamics simulations [21] , but initially were limited to Cartesian grids, extending later to nonconformal meshes. However these approaches are not equipped with an error estimator due to the increased difficulty of curvilinear meshes. Error estimation is crucial for scientific data while less important in image processing. The resolution and high frame rate are the main factors that have an immediate impact in the field of image processing, whereas the audience of scientific data compression is focused on trunaction errors and I/O speed. Additionally, the data layout in digital imaging is given by orthogonal equidistant tensor product grids, while scientific computing involves complex geometries.

Data compression can be achieved by employing other strategies such as the discrete wavelet transform (DWT) [5, 22] or floating point compression [11, 19]. Another approach for data compression of time varying sequences relies on building reduced order models, via proper orthogonal decomposition (POD) [18], [25]. This approach exploits the dynamics of the simulation. However, it may require a time series and the dynamics of the system may not always be properly captured as noted in [18]. These lossy compression strategies can be complemented by a fast lossless compression algorithm [3] that, instead of truncating, maps the data in such a way that the representation of probable sequences are shorter than the one of improbable sequences.

The current work is focused on integral transforms that map the signal to spectral space when higher order spectral element methods are used for the numerical computations. This choice is mandated mainly by the flexibility and scalability of integral transforms. A transform to spectral space can be performed locally on disjoint blocks of data (elements), ensuring a highly paral-

lelizable algorithm. The implementation of an integral transform such as DCT or DLT is equivalent to a matrix premultiplication that is invariant all throughout the domain for any smooth signal. This feature renders the amenable to large scale computations. Following initial developments of the DCT transform we extend the data compression to complex, large scale problems, and derive an a priori error estimator that controls the truncation in spectral space. We also identified that when spectral methods are used for the discretization of the partial differential equation it is more suitable to derive and employ an othogonal transform compatible with that discretization. To this end we derived the DLT transform for Gauss-Legendre-Lobatto grids, which differs from the known finite Legendre transform only by a factor accounting for the end points.

Most works focus on the error in $L_\infty$ or root mean square norm, as in [12], however, we focus on the $L_2$ norm which leads to an easy to handle expression for the truncation tolerance on curvilinear grids.

Gains in data representation due to compression become significant at large scales and high dimensions and affect not only disk storage but also I/O speed. When used solely for visualization purposes we can achieve a as low as 3% in compression; however, we intend to use this technique for a wider range of applications, such as adjoint-based optimization [20], and to this end we have developed an a priori error estimator that gives a clear account of how much data can be truncated in order to restore the solution with a given accuracy. Once the data is truncated, the compression is performed on the actual bits of data. This is achieved through bitwise encoding that specifically targets the truncation symbol, here 0, as the one with highest probability. This demands that data is sufficiently truncated such that the implementation increases I/O speeds, in addition to decreasing file sizes. We note that many more coding strategies are available, such as embedded zerotree coding [23], recursive bottom up [14] we chose the bitwise implementation for its simplicity and the control its lends to the user over the essential compression induced numerical errors.

This paper starts with a description of the data representation (Section 2), followed by a description of orthogonal transforms with the focus on discrete Chebyshev transform and discrete Legendre transform in (Section 3). Based on the properties of these transforms we derive a priori error estimators in Section 4. Concluding the theoretical aspects of this work we outline the bitwise encoding strategy (Section 5). The results, on cases of most difficulty are presented in Section 5.2, together with studies of the increase in I/O performance. In Section 7 we briefly summarize our conclusions.

## 2. Data Representation of the Spectral-Element Method

The spectral-element method has a two-layered data representation, at the element level and inside an element, which is populated with a set of points which are the roots of orthogonal polynomials. Spectral-element methods may differ in their choice of element representation: hexahedral, tetrahedral, and even hexagonal, or hybrids thereof. In terms of the spectral representation

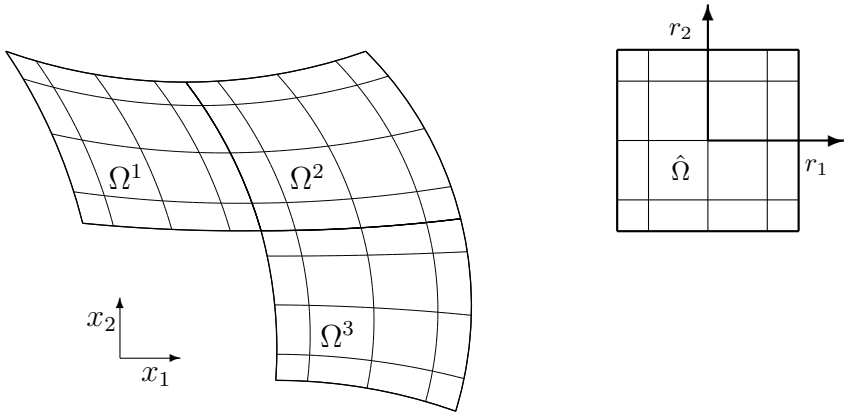



Figure 1: Spectral element mesh, 3 elements with 5 Gauss-Legendre-Lobatto points in each coordinate direction per element. Reference element on the left.

inside each element, the most popular choices are Gauss-Legendre-Lobatto points (GLL) [4] or Chebyshev points [2]. The code Nek5000 [7], used here to showcase the compression strategy at large scale, is based on GLL spectral grids and hexahedral elements; however, for completeness we also tested the method on Chebyshev grids.

The entire domain $\Omega$ is split into elements $\Omega_e$. The solution is required to be continuous across each element boundary, class $C^0$, and admits continuous derivatives within an element, class $C^1$; for details see Section 4.1 and for a lengthier discussion [4].

Consider an element $\Omega_e$ with inner data points $\mathbf{x} = (x_1, x_2, x_3)$. The solution within the element can be represented as a linear combination of orthogonal polynomials

$$\mathbf{u}(\mathbf{x})|_{\Omega_e} = \sum_{i,j,k} u^e_{ijk} \phi_i(x_1)\phi_j(x_2)\phi_k(x_3) \ , \tag{1}$$

where $\phi$ are Lagrange polynomials and the data points $\mathbf{x}$ are GLL points. The polynomials $\phi$ are orthogonal and evaluate to unity at each grid point.

Furthermore the solution over the entire mesh is represented as

$$\mathbf{u}(\mathbf{x})|_{\Omega} = \sum_{j=1}^{M} \mathbf{u}^j \delta_{je} \ , \text{with} \quad \delta_{je} = \begin{cases} 1, & j = e \\ 0, & j \neq e \end{cases} \ .$$

where $M$ is the number of elements building up the mesh, and $\delta_{je}$ is the Kronecker delta function.

In Figure 1 we can distinguish between these two levels of representation, element-wise and GLL inner points. The parallelism of Nek5000 is always performed at the element level; in other words, GLL points interior to an element can never be split between two processors, which ensures a very high parallel scalability.

Without detailing the numerical methods based on a variational formulation, we note that the spectral-element method, just like the finite-element method, does not seek a point-wise discrete solution that approximates the analytical solution, but instead a solution that minimizes the projection on a polynomial space embedded in a suitably chosen Hilbert space, see [4]. With this in mind, a curvilinear element in the spectral element code Nek5000 (be it a part of either a structured or unstructured mesh) is mapped to a reference element where the spectral integration quadrature is defined. If we assume that there exists a mapping from an element $\Omega_e$ to a reference element $\hat{\Omega}_e$, given by $\mathbf{x}(\mathbf{r}) : \Omega_e \to \hat{\Omega}_e$ and that any isoparametric deformation $\mathbf{x}$ is

$$\mathbf{x}(\mathbf{r}) = \sum_{ijk} x_{ijk}\phi_i(r_1)\phi_j(r_2)\phi_k(r_3) \ ,$$

then a data field over the curvilinear mesh of discrete points $\mathbf{x}$ takes the form $\mathbf{u}(\mathbf{x}) = \sum_{ijk} u_{ijk}\phi_i(r_1)\phi_j(r_2)\phi_k(r_3)$ with $r_1, r_2, r_3$ coordinates of the reference element, as illustrated in Figure 1.

In this work we are mainly interested in the $L_2$ norm on curvilinear meshes, namely,

$$\int_{\Omega_e} \mathbf{u}^2(\mathbf{x}) \ \mathrm{d}\Omega_e = \int_{\hat{\Omega}_e} \mathbf{u}^2(\mathbf{r})\mathcal{J}(\mathbf{r}) \ \mathrm{d}\hat{\Omega}_e \ , \tag{2}$$

with $\mathcal{J}$ being the Jacobian of the mapping $\mathbf{x}(\mathbf{r})$:

$$\mathcal{J}(\mathbf{r}) = \det(X^r), \quad (X^r)_{ij} = \frac{\partial x_i}{\partial r_j}, \quad i, j = 1, \ldots, d \ .$$

### 3. Data compression via orthogonal integral transforms

Orthogonal transforms share several essential properties such as orthogonality and separability, however for curvilinear meshes the most important feature is that one can easily couple them with a mapping to cartesian grids. In the present context we discuss curvilinear domains which are paved with quadrilaterals/hexahedrals. For a Delaunay tringulation the present discussion still holds given that for efficient computations using tensor products, triangular domains can be mapped to quadrilaterals, see [2].

Orthogonal transforms can be easily constructed from the orthogonal relation of Jacobi polynomials, or further extensions such as [6]. Consider a set of orthogonal polynomials $\phi_i$ such that

$$\int_{\hat{\Omega}} \phi_i(x)\phi_j(x)\mathcal{W}(x)d\hat{\Omega} = f_i\delta_{ij} \ , \tag{3}$$

where $\mathcal{W}$ is a weight under which the polynomials are orthogonal, and $f$ is a scaling of the orthogonal projection. This orthogonality holds on reference elements, i.e. in $d$ dimensions domains $\hat{\Omega} = [-1, \ 1]^d$. Within the formalism

of Equation 3 one can represent any Jacobi polynomial, as well as other orthogonal transforms. The discrete version of Equation 3 can be stated for a $T_{ij} = \phi_i(x_j)$ as

$$T^T W T = f \ . \tag{4}$$

However we prefer to scale the right side $f$ which leads to $T_{ij} = \sqrt{f_j}\phi_i(x_j)$, providing the discrete equivalent of Equation 3 to be

$$T^T W T = I \ , \tag{5}$$

where $W$ is the mass matrix, a diagonal matrix of the integration weights also including $\mathcal{W}$, and $T$ is an $N \times N$ matrix.

The transform $T$ applied to an array $u$ maps it to spectral space into $v = Tu$. To retrieve the solution vector in real space the inverse $T^{-1} = T^T W$ is applied to obtain $u = T^T W v$.

### *3.1. Discrete Chebyshev Transform*

The discrete Chebyshev transform is a form of a Cosine transform and the terms are, at times, used interchangably The main advantage of the discrete Cosine transform is that it represents the real part of the Fourier transform. In the current context, however, we do not make use of the connection between DCT and the fast Fourier transform, but rather the correspondence of the DCT to the Karhunen-Loève transform (KLT) . In [17], it was shown that DCT is the optimal KL transform, having the best energy compaction efficiency for strongly correlated Markov processes. We also prefer the use of Chebyshev transform terminology since it indicates clearly the spectral grid we operate on.
The DCT transform has been derived from Chebyshev polynomials and a change of coordinates to the Chebyshev points given as $x = \cos \pi k/N$. Since this particular transform has been extensively studied we merely state its formula

$$T_{ij} = f_j \cos\left[\frac{\pi}{2N}(2i-1)(j-1)\right] \ , \tag{6}$$

with

$$f_j = \begin{cases} \sqrt{1/N}, & j = 1, \\ \sqrt{2/N}, & 2 \leq j \leq N \ . \end{cases}$$

The mass matrix has already been scaled out in this case so it amounts to a simple identity matrix. Further references for the cosine transform can be found in [17] and for the Chebyhev polynomials in [4].

### *3.2. Discrete Legendre Transform*

The discrete Legendre transform or finite Legendre transform arises from the orthogonality of Legendre polynomials. The transform $T_{ij} = \sqrt{f_j}\phi_i(x_j)$ is constructed in this case from polynomials $\phi_i$, which can be computed from the recurrence relationship

$$(i+1)\phi_{i+1}(x) = (2i+1)x\phi_i(x) - i\phi_{i-1}(x), \quad i = 1, \ldots N \tag{7}$$

where $\phi_1(x) = 1$, $\phi_2(x) = x$ while the integration weights are

$$w_i = \frac{2}{N(N+1)} \frac{1}{\phi_N^2(x_i)}, \; i = 0, \dots N \tag{8}$$

To derive the DLT on a Gauss-Legendre-Lobatto grid we insert the Legendre polynomials Equation 7 into Equation 3, and together with their associated mass maxtrix $W = diag(w_i)$ we obtain the scaling $f$. Since in the current work we operate on Gauss-Legendre-Lobatto grids the expression of the transform will have a slightly modified expression to account for the end points. As a result $f_j$ will take the values

$$f_j = \begin{cases} \sqrt{(2j-1)/2}, & 1 \leq j \leq N-1, \\ \sqrt{(N-1)/2}, & j = N. \end{cases} \tag{9}$$

### *3.3. Tensor Products*

The separability property of the orthogonal transforms makes them an efficient tool from a computational viewpoint. Due to separability they can be applied on a higher-dimensional field as a tensor product of one-dimensional transforms, thus lowering the computational costs of evaluation.

Consider three matrices $A = (a)_{ij}, \; i,j = 1, \dots N$, $B = (b)_{ij}, \; i,j = 1, \dots N$, $C = (c)_{ij}, \; i,j = 1, \dots N$ then the Kronecker product in a three dimensional space reads

$$\mathbf{v} = (A \otimes B \otimes C)\mathbf{u} = (A \otimes B)\mathbf{u}C^T = A\mathbf{u}C^T B^T \; .$$

In index notation this yields

$$v_{ijk} = \sum_{l=1}^{N} \sum_{m=1}^{N} \sum_{p=1}^{N} a_{il} u_{lmp} c_{pk} b_{mj} \; . \tag{10}$$

In Equation 10 each matrix-matrix product necessitates $N^3$ flops. We have two such products, and the last one has to be performed $N$ times, leading to a total count of $N^3 + N^3 + N^4$. This reduces the evaluation costs from $\mathcal{O}(N^6)$ operations (if the entire matrix is explicitly constructed) to $\mathcal{O}(N^4)$ operations. This aspect of separability, which allows for tensor product representations, becomes important with increasing dimension since the evaluation times go from $\mathcal{O}(N^{2d})$ to $\mathcal{O}(N^{d+1})$, which is clearly advantageous for higher dimensions $d$. Now we apply the orthogonal transform $T$ in all three directions since $N_x = N_y = N_z = N$ resulting in

$$\mathbf{v} = (T \otimes T \otimes T)\mathbf{u} = T\mathbf{u}T^T T^T \tag{11}$$

and upon return to real space for the inverse $T^{-1} = T^T W$

$$\mathbf{u} = (T^{-1} \otimes T^{-1} \otimes T^{-1})\mathbf{v} = TW \; \mathbf{v} \; WT^T \; WT^T \; .$$

This outlines the simplicity of the implementation which only requires the storage of a one-dimensional transform, to be applied for any element.

## 4. A priori error estimator

The truncation of the signal occurs in spectral space after the signal has been transformed. However, in order to maintain control over the error incurred through the truncation of the flow field, it is necessary to devise an a priori error estimator that guarantees that the data can be retrieved with a desired accuracy. As will be shown shortly the $L_2$ norm in spectral space is equivalent to the $L_2$ norm in real space; although the norm is computed locally, it is also satisfied globally.

*Truncation in spectral space.* Assuming a spectral discretization with weights gathered in the mass matrix $W$, the $L_2$ norm in one dimension over a reference element $\hat{\Omega} = [-1,\ 1]$ becomes

$$\|u\|_{L_2} = \sqrt{\int_{\hat{\Omega}} u^2 \, d\hat{\Omega}} \approx \sqrt{\underline{u}^T W \underline{u}} \, .$$

where we consider $u$ to be a one dimensional signal, having $\underline{u}$ as its discrete correspondent. Now if we transfer the field $\underline{u}$ into spectral space via either orthogonal transform (DCT or DLT), we have $\underline{v} = T\underline{u}$:

$$\|v\|_{L_2} = \sqrt{\int_{\hat{\Omega}} v^2 \, d\hat{\Omega}} \approx \sqrt{\underline{v}^T W \underline{v}} = \sqrt{\underline{u}^T T^T W T \underline{u}} \, . \tag{12}$$

From the properties of both the DLT and DCT we have that $T^T W T = I$.

For reference elements the error inccured in spectral space equals that in real space, However on curvilinear elements we need to take into account the Jacobian of the coordinate transformation $x(r) : \hat{\Omega} \rightarrow \Omega$, which destroys the orthogonality in Equation 3. To account for it we need to rewrite the inner product using $\underline{u}_D = diag(\underline{u})$ as

$$\int_{\Omega} u^2 \mathcal{J} d\Omega = Tr(\underline{u}_D W \underline{u}_D J) = Tr(\underline{u}_D^2 W J)$$

where the Jacobian matrix $J$ is also diagonal as is the mass matrix $W$. This representation allows now for commutativity facilitating the reuse of Equation 3 by shifting the Jacobian outside of the inner product. There are of course other ways to represent the inner product, such as a summation of a Haddamard product however this notation best suits our present purposes.

To tie again the spectral space inner product to the real space we compute

$$\underline{v}^T W \underline{v} = (T\underline{u}_D)^T W (T\underline{u}_D) = \underline{u}_D^T T^T W T \underline{u}_D = \underline{u}_D^2$$

And the $L_2$ norm in real space becomes

$$\int_{\Omega} u^2 \mathcal{J} d\Omega = Tr(\underline{u}_D^2 W J) = Tr(\underline{v}^2 W J) \tag{13}$$

The expression in Equation 13 connects the real space norm $\int_\Omega u^2(r)J(r)d\Omega$ to the one in spectral space over a curvilinear element $\int_\Omega v^2(r)J(r)d\Omega$.

The reasoning has been stated so far in one dimension, however since any higher dimensional implementation is to be performed via tensor products according to Section 3.3, it caries through easily by taking into account the identity $(A \otimes B)(C \otimes D) = (AC \otimes BD)$. To prove this we rewrite the right hand side of Equation 12 considering $\mathbf{W}$ to be the three dimensional mass matrix

$$\begin{aligned} \underline{\mathbf{v}}^T \mathbf{W} \underline{\mathbf{v}} &= \mathbf{u}^T (T^T \otimes T^T \otimes T^T)(W \otimes W \otimes W)(T \otimes T \otimes T)\mathbf{u} \qquad (14) \\ &= \mathbf{u}^T (T^T\ W\ T \otimes T^T\ W\ T \otimes T^T\ W\ T)\mathbf{u}. \qquad (15) \end{aligned}$$

It is therefore clear that all operations can be performed in three dimensional space.

*Local to global error.* The entire domain $\Omega$ consists of elements $\Omega_e$, such that $\Omega = \cup_e \Omega_e$. The global velocity field is therefore split into its local restrictions $\mathbf{u}_e = \mathbf{u}|_{\Omega_e}$. Because of the nature of our computational domains, which are curvilinear with nonuniform elements, we need to introduce the weighted norm $\|\cdot\|_w$ scaled by $V = \int_\Omega d\Omega$, the volume of the domain, or element, namely,

$$\|\mathbf{u}\|_w = \sqrt{\frac{\int_\Omega \mathbf{u}^2\ d\Omega}{\int_\Omega d\Omega}} = \sqrt{\frac{\|\mathbf{u}\|^2_{L_2}}{V}}\ . \qquad (16)$$

By virtue of the spectral element domain decomposition and using $\mathbf{u}|_{\Omega_e} = \mathbf{u}_e$ we can write the global $L_2$ norm as a sum of local norms:

$$\|\mathbf{u}\|^2_{L_2} = \sum_e \|\mathbf{u}_e\|^2_{L_2}\ .$$

This translates in terms of the norm weighted by the volume as

$$\frac{\|\mathbf{u}\|^2_{L_2}}{V} = \frac{1}{V}\sum_e \frac{\|\mathbf{u}_e\|^2_{L_2}}{V_e} V_e\ ,$$

which once again with the current definition of the norm, Equation 16, is

$$\|\mathbf{u}\|^2_w = \frac{1}{V}\sum_e \|\mathbf{u}_e\|^2_{w_e} V_e\ . \qquad (17)$$

Regarding the truncation error incurred by $\tilde{\mathbf{u}}$, if we impose a threshold on the global solution $\|\mathbf{u} - \tilde{\mathbf{u}}\|_w < \epsilon$, then from Equation 17 it is sufficient to impose a similar local norm $\|\mathbf{u}_e - \tilde{\mathbf{u}}_e\|_{w_e} < \epsilon$.

### *4.1. Continuity at boundaries*

Elementwise truncation in spectral space may lead to discontinuities at the boundaries if performed agressively. In signal processing this problem is tackled by replacing the DCT by the modified discrete cosine transform [15] which

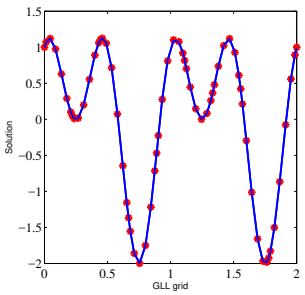
oversamples the signal outside the block where the signal is compressed. However for large scale computations this aproach would alter the scalability since a wide range of points need to be communicated from one block to another.

Fortunately the spectral element method is well suited to blockwise truncation since boundary values are already duplicated from one element to its neighbours and continuity is ensured via direct summation. The spectral element method is an *h*-type finite element method, with the property that integration over each element is performed with spectral accuracy. Across elements continuity can be ensured by averaging the boundary values. Note this is equivalent to direct summation used in spectral element methods [4], [10].

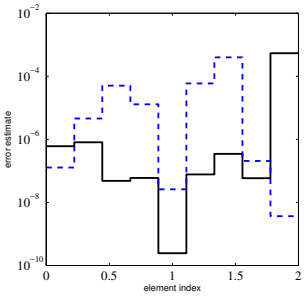
To illustrate that discontinuities across boundaries are properly removed via direct summation we take a closer look at an one dimensional example of spectrally representing the signal consisting of two superposed modes: $u = \cos(4\pi x) + \sin(2\pi x)$. The signal is represented on a spectral element grid consisting of $E = 9$ elements, each element having a Gauss-Legendre-Lobatto grid of $N$ points. A typical operating polynomial order used in production runs [16] is $N = 8$ and we make the same choice here.

Following [10, 13] we note that per each element the solution is represented with a certain sectral accuracy, and we illustrate for the given case Figure 2b what is the inccured error per each element. Spectral accuracy is more difficult to measure than it is for algebraically accurate schemes since the error for a polynomial order $N$ discretization decays as $e_N = c\, e^{-\sigma N}$ with $\sigma$ and $c$ to be determined. In a posteriori fashion one can measure the error and find its slope in a logscale which gives the values for $\sigma$ and $c$.

Subsequently we truncate the signal in spectral space (within each element representation) down to what corresponds in real space $L_2$ norm to an error of $10^{-4}$ which would be an appropiate threshold for many computations. In Figure 2a we note that the signal retrieved after a truncation in spectral space is not qualitatively different despite the truncation.




(a) Signal over two periods: (blue) original signal, (red) truncated signal for $L_2$ norm $10^{-4}$.




(b) A posteriori error per element of original signal (black) and truncated signal (dashed blue).

Figure 2: Spectral representation and error for a signal $u = \cos(4\pi x) + \sin(2\pi x)$ on a grid of $E = 9$ elements, and $N = 8$ gridpoints per element.

One essential aspect is that continuity accross elements is naturally satisfied in the limit of $N \to \infty$, in which case the signal is represented error free on each element. For very high error on an element the continuity may be violated even for a simple signal which has not been truncated in spectral space. Assessing the difference in error between truncated and original signal we conclude that direct summation has to handle the same jump in accuracy from one element to the next which is consistent with the observations we made in practice regarding the smoothness of the signal.

## 5. Compression Algorithm and Implementation

The number of entries within one spectral element is $M = P^d$, where $d$ is the spatial dimension and $P$ the polynomial order. Based on the orthogonal transform and the a priori error estimator, we are able to truncate $K$ entries of $\mathbf{u}$ by setting each one to zero. This leaves us with $N = M - K$ nonzero entries. We assume that each entry takes $m$ bits in its binary encoding $e : \mathbb{R} \to \mathbb{B}^m$, where

$$x \mapsto \underbrace{[b_0, b_1, \ldots, b_{m-2}, b_{m-1}]_2}_{m \text{ bits}} ,$$

with $\mathbb{B}$ being the space of binary numbers. One example of such an encoding is the binary numbers representation where $x = \sum_{i=0}^{m-1} b_i \cdot 2^i$. With double precision floating-point number encoding as defined by IEEE $m$ is equal to $64$ bits. Our on the fly compression algorithm has to be efficient in terms of speed and compression ratio and, flexible with regard to the encoding $e$. Without loss of generality we assume that $e(0) = [0, \ldots, 0]_2$. We propose the compression encoding $h$:

$$h([b_0, b_1, \ldots, b_{m-2}, b_{m-1}]_2) = \begin{cases} \underbrace{[0]_2}_{1 \text{ bit}}, \text{ if } b_0 = \cdots = b_{m-1} = [0]_2 , \\ \underbrace{[1, b_0, b_1, \ldots, b_{m-2}, b_{m-1}]_2}_{m+1 \text{ bits}}, \text{ else } . \end{cases}$$

The entries of value zero are all compressed by using the single bit 0 for their encoding. Every other value is being prepended with the single bit 1. It is a simplified Huffman encoding where we distinguish only two types of symbols: the zeros that are being compressed and the nonzeros. To make this encoding achieve high compression ratios, we assume that zero is by far the most prevalent number.

A lower bound of the compression ratio is for example $\frac{K}{M}$, where all the zeros have zero size. Using our encoding, we end up with $K + (M - K) \cdot (m + 1)$ bits and thus a compression ratio of

$$\frac{K + (M - K) \cdot (m + 1)}{M \cdot m} .$$

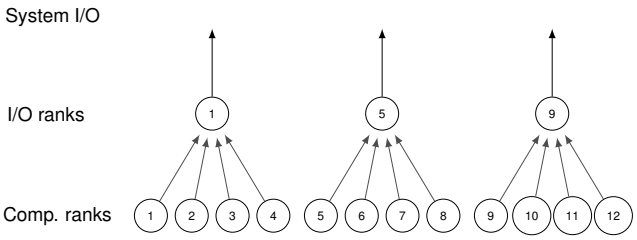



Figure 3: I/O write operation using 12 processes with 3 I/O ranks in Nek5000. Notice that 1, 5, and 9 are both computing and writing.

For $K = 0$, we end up with an increase of $M$ bits, whereas with $K = M$ our highest compression ratio is $\frac{1}{m}$. It follows that this encoding is useful only if $K \gg M/m$. The complexity of this encoding is linear in the number of total grid points $n$ totalling to $\mathcal{O}(n \cdot d)$ with $d$ being the dimension.

### *5.1. Compressed I/O in Nek5000*

I/O has a wide range of use cases, including visualization, resilience, and adjoint checkpointing. It represents the lowest layer of the memory hierarchy where the data is written on permanent storage devices (e.g., disks). Because it is the last memory layer, it generally has the slowest bandwidth and the highest latency of all the memories involved in the computation. This bottleneck poses great challenges for large-scale computer systems, and thus the amount of output data should be selected wisely.

One way of addressing this bottleneck is to adapt the hardware to the access patterns. Burst buffers, for example, assume short peaks of high disk access. Another alternative is to compress and reduce the amount of data. In Nek5000, the stored data is the velocity vector field and the pressure scalar field at each timestep. The geometry can be neglected as it only has to be stored once per simulation. Because of the continuity we assume that these two fields are an excellent candidate for the spectral truncation described in Section 3, while ignoring the effects of shocks.

The I/O implementation of Nek5000 is shown in Figure 3. It distinguishes I/O *ranks* as a subset of *compute ranks*. For brevity reasons we focus only on the write operation, which is fundamentally similar to reads and more frequent. After a write operation is initiated, every compute rank sends its data to the I/O ranks, which then dispatch the data to the system I/O API (i.e., `fwrite`).

Our compression algorithm is made up of two stages. One is the truncation relying on the orthogonal transforms (see Section 4), and the other is the actual compression of the zeros created by the truncation (see Section 5). To achieve maximum scalability, we apply the truncation on the compute ranks. Consequently, it would also be desirable to apply the compression on the compute ranks. However, this design raises flexibility concerns. One data segment $i$ is compressed from full-length $n_i$ to the compressed-length $c_i$ and written out

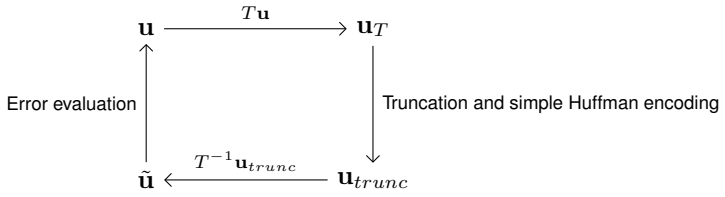



Figure 4: Workflow: (1) Apply the operator $T$ to $\mathbf{u}$ yielding $\mathbf{u}_{dct}$, (2) truncating and compressing $\mathbf{u}_{dct}$ to $\mathbf{u}_{trunc}$, (3) and apply the inverse operator to $T^{-1}$ to $\mathbf{u}_{trunc}$ yielding the restored velocity field $\tilde{\mathbf{u}}$

serially segment after segment. To know the length $c_i$ of the compressed segments, we also store the length $c_i$ at the beginning of each compressed data segment. If the data is compressed on the compute ranks, the I/O ranks collect the various compressed data segments together with the prepended $c_i$'s and write them to disk.

In case of read, the data is read from disk by the I/O ranks from separate files and then distributed to compute ranks. If compression would take place on the compute ranks, this distribution would have to rely on meta data for the location of the elements in the data stream and additional hand shakes for the variable length of the data stream. To avoid this we chose the former option of doing the zero compression only on the I/O ranks. Thus our compression only scales with the number of I/O ranks.

### *5.2. Implementation*

The algorithms discussed here are independent of the nature of the data fields, even though the compression is applied here to flow fields from turbulent simulations. This choice is justified by our numerical experiments which showed that fully turbulent flow fields are the hardest to compress since the signal usually has high frequencies and is fully resolved. On the other hand, signals that are overresolved can be highly compressed, and this implies fields stemming from Reynolds-averaged Navier-Stokes, laminar flows, and so forth compress very well using these techniques.

Both cases selected here are results of simulations using the incompressible Navier-Stokes, which in nondimensional formulation are given by

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} &= -\nabla p + \frac{1}{Re}\nabla^2\mathbf{u} + \mathbf{f} \\ \nabla\cdot\mathbf{u} &= 0\,, \end{aligned} \tag{18}$$

where $\mathbf{u}$ is the flow velocity, $p$ the pressure, and $Re$ the Reynolds number given as $Re = UD/\nu$ ($\nu$ being the viscosity).

### *5.3. Algorithm*

Combining the truncation in spectral space (see Section 4), the simple Huffman-based compression (see Section 5), the error estimator (see Sec-

**Algorithm 1** Parallel compression on an already partitioned mesh with `nel` elements each.

```
    procedure SETUP
        Create the DCT/DLT operator T
    end procedure
    procedure TRUNCATE
 5:     for 0 ≤ k < nel do
            u_{T,k} = T · u_k · T^T
            sort(u_{T,k})
            u_{trunc,k} = u_{T,k} > ε
        end for
10: end procedure
    procedure COMPRESS
        if IOnode then
            for q ∈ IOchildren do
                Recv(u_trunc,q)
15:             u_compress =huff_encode(u_trunc)
                output(u_compress)
            end for
        else
            Send(u_trunc,IOparent)
20:     end if
    end procedure
```

tion 4), and the I/O layout in Nek5000 (see Section 5.1), we derive the parallel algorithm in Algorithm 1 and its implementation. The parallel algorithm is described on an already partitioned data `u`, each process executing it locally.

The `Setup` procedure is called once at the startup of Nek5000. It creates the DCT/DLT operator `T`, which depends ultimately only on the polynomial order `p=N-1`. The operator `T` is separable and of size `N·N`, thus its implementation via tensor products renders its computation extremely fast.

The `Truncate` and `Compress` procedures are called one after another each time an output is initiated. Every process iterates over its elements `k` and applies the three-dimensional DCT/DLT using the tensor product implementation of `T` (line 6). Subsequently, the resulting DCT/DLT transformed array $\mathtt{u}_{\mathtt{DCT}}$ is sorted and then truncated based on the error estimator (line 8). No communication is involved in this process, making it embarrassingly parallel.

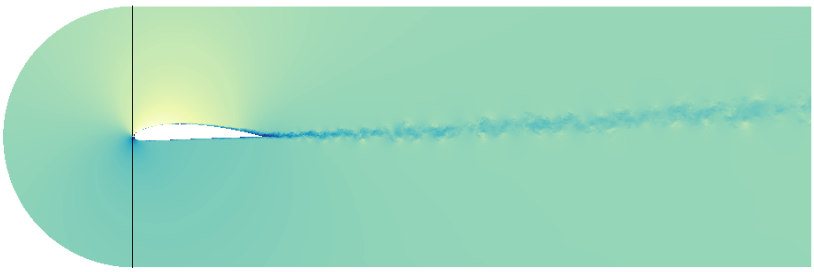
In `Compress` the Huffman encoding is applied at the I/O node level. If the process is an I/O node (see Figure 3), it iterates over all its children and receives the truncated data of every child `q` (line 14). It then applies the Huffman encoding, as seen in Section 5, and writes the output of child `q` to disk. If the process is not an I/O node it just sends the data to its I/O parent `IOparent` (line 19). The number of messages and thus its latency depends on the number of children in `IOchildren` (line 13). Assuming this ratio is constant for a given system and architecture, the compression step is also embarrassingly parallel at the I/O node level.

The uncompressed I/O uses the same communication structure for collection the data as in the routine `COMPRESS`. To achieve a speedup in the I/O using the compression, the `for` loop in lines 13-17 has to be faster than in the uncompressed case. All the operations in that loop, `Recv`, `huff_encode`, and `output` scale linearly with the data size. Thus, if the performance gain of the compression in `output` is high enough to account for the additional time needed for the encoding, we experience a speedup. The operating parameters are highly system dependent. `huff_encode` is a highly memory bandwidth bound routine, whereas `output` is I/O bound. The current development in diverging memory access and I/O speeds gives us confidence that future architectures will increase the benefits of our approach.

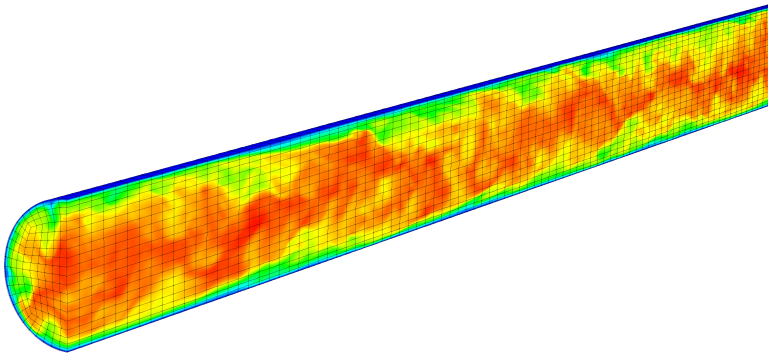
## 6. Results

The first case is a three-dimensional direct numerical simulation of flow past an airplane wing [8] at the high Reynolds number of 400,000 based on freestream velocity and cord length. To fully resolve the turbulent scales requires 3.2 billion grid points (1,847,664 elements), making this a typical large-scale computing application. This flow case, as seen in Figure 5a, is not fully turbulent in the entire domain, and regions with less fluctuations are compressed more than their counterparts in the turbulent region.

The second case is the canonical example of turbulent flow in a straight pipe at a friction Reynolds number of $Re_\tau = 180$, studied in detail in [9]. This is a small case, using 36,480 elements with approximately 18 million gridpoints.



(a) Flow past an airplane wing ($\approx$ 3.2 billion gridpoints).



(b) Flow in a pipe at $Re_\tau = 180$.

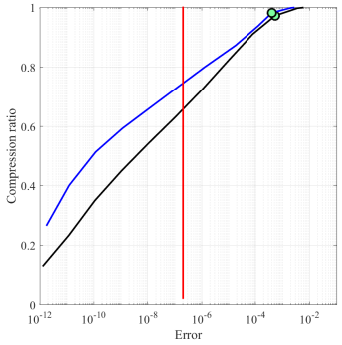
Figure 5: Velocity magnitude of different flow configurations.

The implementation follows Algorithm 1, or the schematic view Figure 4, by which we apply the DCT/DLT transform, then truncate, and compress. Note the distinction we make here between truncation, which is the mathematical procedure of setting to zero the information that is deemed redundant in the signal (i.e. below a certain tolerance), and compression, which is the actual bitwise encoding that removes these zeros from the data set.

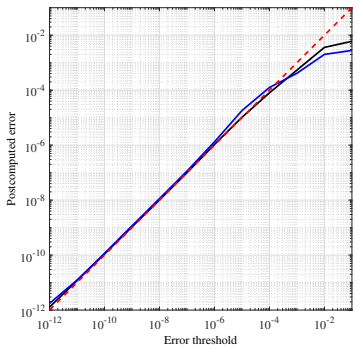
We tested a set of tolerances in the range of $10^{-i}$ with $i = 1, \dots 13$ and compressed optimally based on the a priori estimator in Section 4. Assessing the space savings versus loss of accuracy as in Figure 6a and Figure 7a we note that even in the most challenging case of fully developed turbulence, one can compress more at higher or equal accuracy than by storing data in single precision.

The use of compression is uselful for both file restarts at a minimally necessary accuracy and visualizations. Nonetheless, the requirements for visualization seem to be low. An accuracy of $10^{-2}$, corresponding to 3% compression, is

consistently sufficient for satisfactory visualizations, as illustrated in Figure 8b, and Figure 8c.

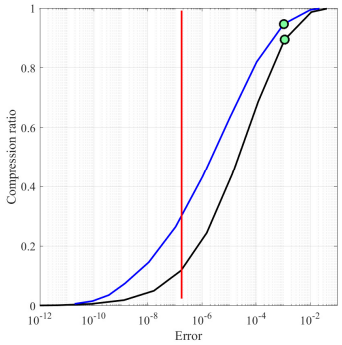



(a) Space savings vs error: (blue) GLL grid, (black) Chebyshev grid, (green marker) corresponds to visualization in Figure 8c.

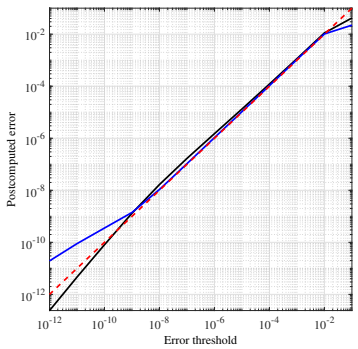



(b) A priori vs a posteriori error: (blue) GLL grid, (black) Chebyshev grid.

Figure 6: Flow past an airplane wing




(a) Space savings $C_r$ vs error: (blue) GLL grid, (black) Chebyshev grid, (green marker) corresponds to visualization threshold.

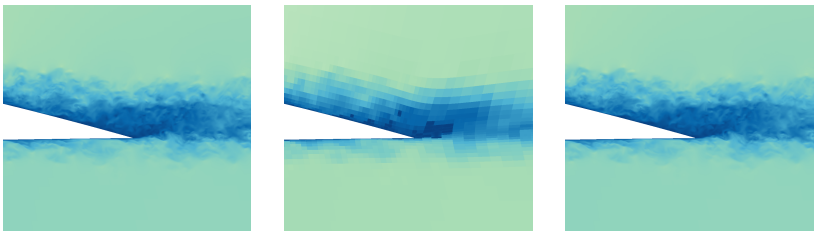



(b) A posteriori error vs imposed tolerance on a priori error estimate: (blue) GLL grid, (black) Chebyshev grid, (red dotted) imposed tolerance.

Figure 7: Flow in a pipe

### *6.1. I/O Performance*

The implementation of the I/O has been described in Section 5.1. We evaluate both the on-node and intranode performance of our implementation. The latter represents a realistic test run using Nek5000 on the established Mira BG/Q system. The on node benchmarks ignore the impact of the network and solely focus on the maximum throughput for a single node. Here, we compare

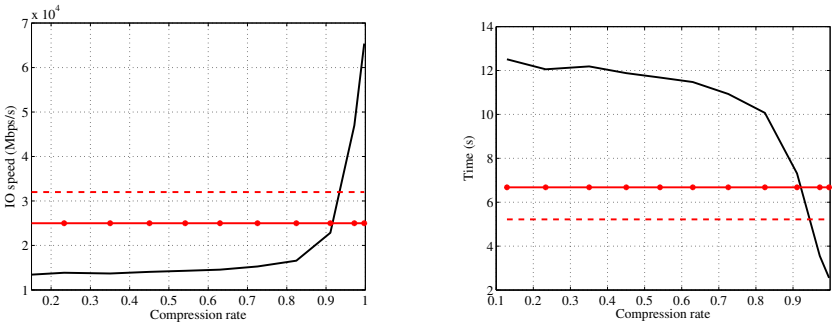


(a) Original uncompressed field

(b) Compression 99%, at an error of $10^{-1}$

(c) Compression 97%, at an error of $10^{-2}$

Figure 8: Visualization retrieved at different compression ratios for flow around a wing.




(a) I/O speed in Mbps/s vs compression ratio using 65,536 processes with 1,024 I/O nodes for the uncompressed and 2,048 I/O nodes for the compressed write, (red dotted) theoretical bandwidth limit, (red continuous) speed for uncompressed data.

(b) Time to write in seconds versus compression ratio, (red dotted) theoretical bandwidth limit, (red continuous) speed for uncompressed data.

Figure 9: I/O performance on the flow past an airplane wing case.

the Mira BG/Q node to the Intel Knight Landing (KNL) architecture. The lack of access to a full KNL system restricts us from analyzing the intra node I/O performance on such a system.

First, we benchmarked the I/O speeds on 1,024 Mira BG/Q nodes at Argonne using 65,536 processes. Each Mira node is composed of 16 cores. Nek5000 was run with 32 MPI processes per such node, with two processes per core. Every 128 Mira nodes are physically connected to one I/O node at 4 GB/s. That 4 GB/s connection is attached to a network switch and the backbone, which eventually leads to the disks. System noise due to other running applications may prevent full usage of the 4 GB/s at the I/O node level. Thus, at 1,024 nodes we may expect a maximum of 32 GB/s assuming no interference with other jobs.

The number of I/O ranks in Nek5000 does not necessarily correspond to the number of physical I/O nodes on Mira. To determine the maximum write speed,

Table 1: Overview of runtime parameters

| Data | Mira Nodes | Nek I/O Ranks | Nek Ranks |
|---|---|---|---|
| Uncompressed | 2,048 | 1,024 | 65,536 |
| Compressed | 2,048 | 2,048 | 65,536 |

we wrote data at various numbers of I/O ranks ranging from 128 to 2,048 with 10 measurements each. The peak performance was achieved by using 1,024 I/O ranks for the uncompressed data and 2,048 ranks for the compressed data. The parameters of the runs are summarized in Table 1, and the performance is plotted in Figure 9a.

The cost of writing compressed data has been at most twice the cost of writing uncompressed data, occurring at a compression rate below 20%. At a compression rate above 90% an I/O speed advantage can be observed. This would be the case mainly with visualization data. As the bitwise mapping is still the bottleneck of our implementation, we expect that further optimizations in the implementation should lead to an I/O speed increase at lower compression ratios.

## 7. Conclusions

We have implemented a scalable lossy data compression in a spectral-element code and controlled the data compression via an a priori error estimator. The lossy data algorithm has been inspired by the DCT transform used by the JPEG image compression algorithm. However, we have identified that the compression ratio may achieve higher ratios by deriving an orthongonal transform compatible with the spectral discretization of the code. Therefore we studied both the traditional DCT transform as well as a variant of the discrete Legendre transform tailored to the girds we were operating on, i.e. Gauss-Legendre-Lobatto. The a priori error estimator is based on the $L_2$ norm, which is a far more suitable norm for continuous scientific data than is the *rms* norm used in image compression. The algorithm has an outstanding performance for visualization, where it is possible to retain the same visualization quality using only 3% of the initial data. The compression of the data on disk is achieved via Huffman encoding, which leads not only to reduced sizes in data but also to faster I/O speeds by saturating the bandwidth at a higher speed. The implementation of the orthogonal transforms is not only flexible but also, because of the separability property, leads to an efficient implementation via tensor products, thereby lowering the computational cost for three-dimensional data from $\mathcal{O}(N^6)$ to $\mathcal{O}(N^4)$. The algorithm is also embarassingly parallel which renders the cost of the compression neglectable in a production code such as Nek5000.

The performance of our bitwise mapping heavily depends on the C compiler, because the code is relying only on C bitwise operators. We observed far better performance results with Intel compiler on the Knights Landing and than with the BG/Q system compiler, reaching 10 times faster compression per core on

Knights Landing. If I/O speed is key, a future implementation should shift the compression onto the compute ranks.

**Acknowledgments**

We acknowledge the support of Barry Smith (ANL), Philipp Schlatter (KTH), Ricardo Vinuesa (KTH), and Nicolas Offermans (KTH). This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This material was based upon work funded by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357

## References


[1] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, Jan 1974.

[2] John P Boyd. *Chebyshev and Fourier spectral methods*. Courier Corporation, 2001.

[3] Yann Collet. Lz4: Extremely fast compression algorithm. *code. google. com*, 2013.

[4] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge University Press, 2002. Cambridge Books Online.

[5] Gordon Erlebacher, M Yousuff Hussaini Director, et al. *Wavelets: Theory and Applications: Theory and Applications*. Oxford University Press, 1995.

[6] Bernd Fischer and Gene H. Golub. How to generate unknown orthogonal polynomials out of known orthogonal polynomials. *Journal of Computational and Applied Mathematics*, 43(1):99 – 115, 1992.

[7] Paul Fischer, James Lottes, Stefan Kerkemeier, Oana Marin, Katherine Heisey, Aleks Obabko, Elia Merzari, and Yulia Peet. Nek5000: User's manual. Technical Report ANL/MCS-TM-351, Argonne National Laboratory, 2015.

[8] Seyed Mohammad Hosseini, Ricardo Vinuesa, Philipp Schlatter, Ardeshir Hanifi, and Dan S Henningson. Direct numerical simulation of the flow around a wing section at moderate reynolds number. *International Journal of Heat and Fluid Flow*, 2016.

[9] George K. Khoury, Philipp Schlatter, Azad Noorani, Paul F. Fischer, Geert Brethouwer, and Arne V. Johansson. Direct numerical simulation of turbulent pipe flow at moderately high Reynolds numbers. *Flow, Turbulence and Combustion*, 91(3):475–495, 2013.

[10] David Kopriva. *Implementing spectral methods for partial differential equations: Algorithms for scientists and engineers*. Springer Science & Business Media, 2009.

[11] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[12] Alexander Loddoch and Jörg Schmalzl. Variable quality compression of fluid dynamical data sets using a 3-d dct technique. *Geochemistry, Geophysics, Geosystems*, 7(1), 2006.

[13] Catherine Mavriplis. Adaptive mesh strategies for the spectral element method. *Computer Methods in Applied Mechanics and Engineering*, 116(1):77 – 86, 1994.

[14] Alistair MoffaT and Vo Ngoc Anh. Binary codes for non-uniform sources. In *Data Compression Conference, 2005. Proceedings. DCC 2005*, pages 133–142. IEEE, 2005.

[15] V. Nikolajevic and G. Fettweis. Computation of forward and inverse mdct using clenshaw's recurrence formula. *Trans. Sig. Proc.*, 51(5):1439–1444, May 2003.

[16] Nicolas Offermans, Oana Marin, Michel Schanen, Jing Gong, Paul Fischer, and Philipp Schlatter. On the strong scaling of the spectral element solver nek5000 on petascale systems. In *Proceedings of the Exascale Applications and Software Conference 2016*, page 5. ACM, 2016.

[17] K. R. Rao and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[18] Muruhan Rathinam, Linda, and R. Petzold. A new look at proper orthogonal decomposition. *SIAM J. Numer. Anal*, 41:1893–1925, 2003.

[19] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. Exploration of lossy compression for application-level checkpoint/restart. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 914–922. IEEE, 2015.

[20] Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver Nek5000. *Procedia Computer Science*, 80:1147 – 1158, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[21] Jörg Schmalzl. Using standard image compression algorithms to store data from computational fluid dynamics. *Computers & Geosciences*, 29(8):1021–1031, 2003.

[22] Kai Schneider and Oleg V Vasilyev. Wavelet methods in computational fluid dynamics*. *Annual Review of Fluid Mechanics*, 42:473–503, 2010.

[23] Jerome M Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on signal processing*, 41(12):3445–3462, 1993.

[24] Aaron Trott, Robert Moorhead, and John McGinley. Wavelets applied to lossless compression and progressive transmission of floating point data in 3-d curvilinear grids. In *Visualization'96. Proceedings.*, pages 385–388. IEEE, 1996.

[25] F. van Belzen and S. Weiland. A tensor decomposition approach to data compression and approximation of nd systems. *Multidimensional Systems and Signal Processing*, 23(1):209–236, 2012.