Learning to differentiate

Oskar Ålund, Gianluca Iaccarino and Jan Nordström

Ålund, O., Iaccarino, G., Nordström, J., (2021), Learning to differentiate, *Journal of Computational Physics*, 424, 109873. https://doi.org/10.1016/j.jcp.2020.109873

Learning to Differentiate

Oskar Ålund[a], Gianluca Iaccarino[b], Jan Nordström[a,c]

[a] *Department of Mathematics, Computational Mathematics, Linköping University, SE-581 83 Linköping, Sweden*
[b] *Department of Mechanical Engineering and Institute for Computational Mathematical Engineering, Stanford University, Stanford, California 94305*
[c] *Department of Mathematics and Applied Mathematics, University of Johannesburg, P.O. Box 524, Auckland Park 2006, South Africa*

Abstract

Artificial neural networks together with associated computational libraries provide a powerful framework for constructing both classification and regression algorithms. In this paper we use neural networks to design linear and non-linear discrete differential operators. We show that neural network based operators can be used to construct stable discretizations of initial boundary-value problems by ensuring that the operators satisfy a discrete analogue of integration-by-parts known as summation-by-parts. Our neural network approach with linear activation functions is compared and contrasted with a more traditional linear algebra approach. An application to overlapping grids is explored. The strategy developed in this work opens the door for constructing stable differential operators on general meshes.

1. Introduction

Discrete differential operators serve as the basic building blocks of numerical schemes for solving partial differential equations (PDEs). A variety of different discretization techniques exist, e.g. Finite Element (FEM), Finite Volume (FVM), and Finite Difference (FDM) methods, each with their own benefits and drawbacks. A central property for any kind of discretization is *stability*. That is the existence of an estimate of the solution to the discretized problem in terms of boundary and/or initial data. For a well-posed Initial Boundary Value Problem (IBVP), stability ensures that errors in the discrete solution do not grow uncontrollably as time progresses. For consistent linear problems, stability also guarantees convergence of the numerical solution to the exact solution of the original IBVP.

Summation-By-Parts (SBP) is a method-agnostic property that has been applied extensively in the literature in order to show stability of a wide variety of numerical schemes, including FDM, FVM, and Discontinuous Galerkin (DG) methods [1, 2, 3, 4, 5, 6]. As the name suggests, SBP is a discrete analogue of integration by parts and, together with weak boundary conditions imposed using the so-called Simultaneous Approximation Terms (SATs), forms a cohesive

and systematic approach to stable discretization design known as the SBP-SAT technique.

Artificial neural networks (hereafter referred to as neural networks) are rudimentary models of cerebral subsystems. The networks are typically built as successive layers of computational neurons, with the first layer corresponding to the inputs and the last layer yielding the outputs. Mathematically, the outputs are obtained as nonlinear combinations of the inputs and the values stored in the intermediate layers. It is not uncommon to build networks with dozens of layers and tens of neurons per layer. The determination of the structure of the layers (number of layers, neurons, and type of connections) is typically based on experience while the values stored in the neurons are obtained by solving an optimization problem aimed at minimizing the error incurred when reproducing known input/output relations (supervised learning). Neural networks have seen an explosive growth in popularity in the recent decades, both in terms of efficacy and accessibility. The applications span a wide range of fields, such as computer vision, natural language processing, autonomous driving, games, etc. Furthermore, state of the art, open source computer libraries such as Tensorflow [7] and Keras [8] provide powerful high level abstractions for implementing and training neural networks. The title "Learning to Differentiate" is a play on words, using the fact that "learning" in a machine learning setting means to derive weights through optimization—something that will be leveraged throughout the paper.

In this paper we lay the groundwork for constructing stable discretizations of IBVPs using SBP operators based on neural networks. Some previous attempts at incorporating machine learning techniques for PDEs have emerged in the literature recently. For example, a neural network can be used as a PDE solution ansatz by leveraging the universal approximation theorem [9]. A similar idea have been used to solve fractional PDEs [10]. Conversely, certain classes of neural networks can be interpreted as discretizations of ordinary differential equations (ODEs), allowing one to use existing ODE solvers to train them [11]. Another application is to train a neural network to discover governing equations by processing experimental data (i.e. parameter estimation) [12]. However, to the best of our knowledge, it is not known how to systematically construct stable schemes based on neural networks for IBVPs.

Our approach focuses on the discretization of the differential operator. That is, given an IBVP $u_t + \mathcal{L}(u) = 0$, we want to construct a discrete version of $\mathcal{L}$ using a neural network, such that the resulting discrete scheme is stable. We demonstrate the present method by solving an IBVP in one physical dimension and incorporating the SBP property using a neural network with linear activation functions. More precisely, for a given grid we train a quadrature followed by an operator which satisfies the SBP property with respect to this quadrature. Even though for a given quadrature an SBP operator can be found by solving a linear system [1], finding quadratures (with positive weights) on arbitrary grids is a nontrivial task, especially in higher dimensions. However, it has been shown that certain optimization problems related to quadrature construction (similar to those used in this paper) can be readily extended to multiple dimensions [13]. The possibility of generating quadratures on arbitrary meshes could po-

tentially pave the way for a systematic stable treatment of overlapping meshes [14, 15, 16], or general unstructured grids [2, 17].

To illustrate the potential of using neural networks in operator construction, we describe an application to overlapping grids in one dimension. Overlapping grids are notoriously difficult to work with, but a number of successful approaches based on interpolation has been shown to be practical [16, 18, 19, 20, 21]. Proofs of stability for low order interpolation based methods in one dimension can be found in [22, 23, 24], in higher dimensions stability is not known. In this paper we explore a different approach, based on a hybrid FDM/FEM formulation. A one-dimensional overlap, formed by two uniform grids, is split into elements, and the available nodes are used to solve for element-specific SBP operators. The non-overlapping parts are treated using traditional finite difference based SBP operators. Due to the general nature of SBP-SAT couplings, this approach leads to a class of high order, provably stable schemes for overlapping meshes in one dimension.

We also discuss and compare two methods of computing the element-specific operators—one based on traditional linear solvers, and another one based on Tensorflow.

The paper is structured as follows. In Section 2 we give a brief introduction to SBP operators. In Section 3 we show how to represent SBP operators as neural networks, and how to train them on irregular grids. We demonstrate how the networks can be used to construct a stable discretization of the linear advection equation. Section 3 also includes a conventional linear algebra formulation of these operators. In Section 4 the custom operators are interpreted as solutions to constrained linear systems that can be solved using either a classical linear solver or a Tensorflow approach. The splitting of overlapping grids into FDM parts and FEM parts are discussed in Section 5. Section 6 shows numerical results for two model problems, as well as a comparison between the Matlab and the Tensorflow approaches to solve for the element-specific operators. Conclusions are outlined in Section 7.

2. SBP Operators

SBP operators are discrete differential operators that satisfy a discrete version of integration by parts. Given a grid $x_0 < x_1 < \cdots < x_N$, an SBP operator $\mathbf{D} = \mathbf{P}^{-1}\mathbf{Q}$ is built from a (typically diagonal) quadrature matrix $\mathbf{P}$ and a matrix $\mathbf{Q}$ such that

$$\mathbf{Q} + \mathbf{Q}^\top = \mathrm{diag}(-1, 0, 0, ..., 1)\,. \tag{1}$$

We assume for simplicity that the endpoints of the discretized interval are included in the grid points $\{x_i\}$ (although this is not necessary [1]). The quadrature matrix $\mathbf{P}$ defines an inner product $(\mathbf{u}, \mathbf{v}) := \mathbf{u}^\top P \mathbf{v}$ on $\mathbb{R}^{N+1}$. If $\mathbf{u}$ and $\mathbf{v}$ are evaluations on the grid of some smooth functions u and v, then $(\mathbf{u}, \mathbf{v}) \approx \int_{x_0}^{x_N} uv dx$. Furthermore, it follows from (1) that

$$(\mathbf{u}, \mathbf{D}\mathbf{v}) = u_N v_N - u_0 v_0 - (\mathbf{D}\mathbf{u}, \mathbf{v})\,. \tag{2}$$

Note that (2) is a discrete analogue of the integration by parts relation

$$\int_{x_0}^{x_N} uv'dx = u(x_N)v(x_N) - u(x_0)v(x_0) - \int_{x_0}^{x_N} u'vdx\,. \tag{3}$$

SBP operators are widely used to construct provably energy stable discretizations of partial differential equations. They were originally conceived as finite difference based operators with a modified boundary stencil that provided the SBP property. Diagonal quadrature matrices lead to a class of operators known as SBP$(2q, q)$, where q denotes the order of accuracy at boundary nodes, and $2q$ the order of accuracy at interior nodes. It can be shown that this is the optimal relationship between the boundary and interior accuracy. SBP operators also span element based methods, and we can think of them as polynomial differentiators that operate on quadrature nodes inside the elements. The $(2q, q)$ relation resurfaces in this setting as the optimal relation between the order of accuracy of the quadrature and the order of accuracy of the SBP operator (more precisely, a quadrature of order $2q$ is a necessary and sufficient condition for the existence of an SBP operator of order q) [1].

3. Machine Learning and Differentiation

Consider the task of differentiating quadratic polynomials on a three point grid. This is a well defined linear operation—three point values uniquely determine a quadratic polynomial and its derivative. Let us fix the grid points $\Omega_h = \{-1, 0, 1\}$ in the interval $\Omega = [-1, 1]$. It is straightforward to compute the matrix $\mathbf{D}$ that differentiates quadratic polynomials on Ω_h. Let

$$\mathbf{1} = \begin{bmatrix}1 & 1 & 1\end{bmatrix}^\top, \quad \mathbf{x} = \begin{bmatrix}-1 & 0 & 1\end{bmatrix}^\top, \quad \mathbf{x}^2 = \begin{bmatrix}1 & 0 & 1\end{bmatrix}^\top.$$

Then $\mathbf{D}$ is uniquely determined by the relations $\mathbf{D1} = \mathbf{0}$, $\mathbf{Dx} = \mathbf{1}$, and $\mathbf{Dx}^2 = 2\mathbf{x}$. Or, in matrix form,

$$\mathbf{D}\begin{bmatrix}1 & -1 & 1\\ 1 & 0 & 0\\ 1 & 1 & 1\end{bmatrix} = \begin{bmatrix}0 & 1 & -2\\ 0 & 1 & 0\\ 0 & 1 & 2\end{bmatrix} \implies \mathbf{D} = \begin{bmatrix}-1.5 & 2 & -0.5\\ -0.5 & 0 & 0.5\\ 0.5 & -2 & 1.5\end{bmatrix}. \tag{4}$$

What does it mean for a neural network to learn this operation? A rudimentary approach is to consider an input layer with three neurons, a single fully connected layer, linear activation functions and zero bias. This means that the neural network is a linear map described by a 3×3 weight matrix $\mathbf{W} = (w_{ij})$. The network can then be trained by minimizing a loss function depending on known input and output vectors. For example, if we use the mean square error loss function, training is equivalent to minimizing the quantity $\sum_{i=0}^{2} \|\mathbf{p}_i' - \mathbf{W}\mathbf{p}_i\|_2^2$, where $\mathbf{p}_i$ and $\mathbf{p}_i'$ are the basis polynomials and its derivatives in $\mathbb{P}_2$, evaluated in the grid nodes. The solution to this minimization problem is the matrix $\mathbf{D}$ above.

Remark 1. *The choice of the network architecture (connections, layers) is independent to the transfer of information between neurons which is governed by the activation function. We have focused on simple linear laws and single-layer structures to contrast the present strategy to more classical approaches.*

Remark 2. *The matrix* $\mathbf{D}$ *can be used to differentiate quadratic polynomials on an arbitrary interval* $[a, b]$ *by a transformation. The matrix* $\frac{2}{b-a}\mathbf{D}$ *differentiates quadratic polynomials evaluated in* $x_0 = a$, $x_1 = (a + b)/2$, $x_2 = b$.

Remark 2 implies that we can formulate refinable semi-discretizations of PDEs using $\mathbf{D}$. Consider the linear advection equation

$$u_t + u_x = 0, \quad x \in (0, 1), \quad t > 0, \tag{5}$$

with $u(0, x) = f(x)$ and $u(t, 0) = g(t)$. We subdivide the unit interval into N elements, $I_n = ((n - 1)/N, n/N)$, $n = 1, 2, \ldots, N$. Each element is discretized into three equidistant points associated with a solution vector $\mathbf{u}^{(n)} = (u_0^{(n)}, u_1^{(n)}, u_2^{(n)})^\top$ and a discrete differential operator $\mathbf{D}^{(n)}$ based on $\mathbf{D}$ and Remark 2. Let $\mathbf{e}_0 = (1, 0, 0)^\top$ and consider the system

$$\begin{aligned}
\mathbf{u}_t^{(1)} + \mathbf{D}^{(1)}\mathbf{u}^{(1)} &= \sigma_1(u_0^{(1)} - g)\mathbf{e}_0 \\
\mathbf{u}_t^{(2)} + \mathbf{D}^{(2)}\mathbf{u}^{(2)} &= \sigma_2(u_0^{(2)} - u_2^{(1)})\mathbf{e}_0 \\
&\vdots \\
\mathbf{u}_t^{(N)} + \mathbf{D}^{(N)}\mathbf{u}^{(N)} &= \sigma_N(u_0^{(N)} - u_2^{(N-1)})\mathbf{e}_0\,.
\end{aligned} \tag{6}$$

The elements are coupled weakly by the penalty terms on the right-hand sides. The relation (6) is essentially a nodal discontinuous Galerkin scheme [25], but it is not clear how to choose the penalty coefficients σ_n such that the scheme is stable (i.e. such that $\mathbf{u}$ is bounded in terms of g and f).

We might by an intuitive argument guess the sign of σ_n: If $u_0^{(n)} - u_2^{(n-1)}$ is positive, then we should add a negative contribution to the time-derivative of $u_0^{(n)}$ to push the solution toward continuity—hence $\sigma_n < 0$. However, the required magnitude of σ_n is difficult to guess at this stage—if it is too small, the scheme will not be stable, and if it is too large, the scheme will be stiff. Effects of penalty coefficients with insufficient magnitude can be seen to the left in Figure 1; the boundary and interface conditions are not imposed at sufficiently high rate, in this case causing the solution to lag behind inflow data. In general, the behavior of an unstable scheme is unpredictable. Depending on initial and boundary data the solution may grow uncontrollably or lose energy, or it may be accurate for an unknown period of time. In any case, if we cannot prove stability of the scheme, we cannot trust it to produce accurate results.

In order to produce a differentiation operator which can be used to construct provably stable discretizations one may define a quadrature matrix $\mathbf{P} \in \mathbb{R}^{3\times3}$. More precisely, let $\mathbf{P}$ be a diagonal matrix with a quadrature along the diagonal such that $\mathbf{1}^\top\mathbf{P}\mathbf{x}^k = \int_{-1}^{1} x^k dx$ for k as large as possible. We can then train the

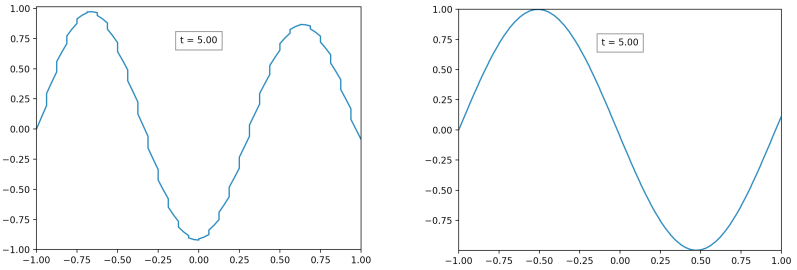

Figure 1: Left: The solution to system (6) with too small penalty coefficients and $g(t) = \sin(\pi t)$. Right: The solution to system (6) with appropriate penalty coefficients and $g(t) = \sin(\pi t)$.

network on the inputs $\mathbf{p}_i$ and the outputs $\mathbf{P}\mathbf{p}_i'$. This will result in a weight matrix $\mathbf{Q}$ such that $\mathbf{D} = \mathbf{P}^{-1}\mathbf{Q}$ is a differential operator. If, in addition, we impose the property that $\mathbf{Q} + \mathbf{Q}^\top = \operatorname{diag}(-1, 0, 1)$, then $\mathbf{D}$ will satisfy the SBP property with respect to $\mathbf{P}$. This means that it can be used to construct provably stable schemes using the standard SBP-SAT framework [3, 1]. The penalty coefficients in (6) can be determined by using a discrete version of the energy method, see [3] for details. A plot of a stable solution to (6) with appropriate penalty coefficients is shown to the right in Figure 1. Note that it is smooth and conserves the energy.

If our grid points are arbitrarily placed and we do not have an a priori known quadrature, we can train a quadrature matrix $\mathbf{P}$ just like we trained $\mathbf{D}$ before, by using basis polynomials as inputs and their integrals as outputs. This procedure of training a quadrature matrix $\mathbf{P}$ followed by determining a matrix $\mathbf{Q}$, such that $\mathbf{D} = \mathbf{P}^{-1}\mathbf{Q}$, is a straightforward way to compute SBP operators on arbitrary grids using standard machine learning software. A word of warning: during our training sessions we found out that we must be careful in choosing the training data. Recall that there is a unique operator that differentiates all polynomials in $\mathbb{P}_2$ exactly on any 3-point grid (it is derived as in the beginning of this section). Therefore, if we impose additional requirements such as an SBP property, we should expect our trained operator to lose at least one order of accuracy, unless we happen to find a quadrature for which the unique uniformly second order operator *is* an SBP operator.

In our specific example, it turns out that the matrix $\mathbf{D}$ in (4) is in fact an SBP operator with respect to a particular quadrature—the Gauss–Lobatto quadrature [4]. Note that both the Gauss–Lobatto quadrature and the derivative matrix $\mathbf{D}$ in (4) is recovered by training $\mathbf{P}$ on the basis in $\mathbb{P}_2$ and by training $\mathbf{Q}$ on the basis in $\mathbb{P}_1$. That is, even though the neural network represented by $\mathbf{Q}$ was only trained on affine polynomials, it still converged to the second order accurate operator. In general we cannot expect to be this lucky. The order of an SBP operator can be no larger than half the order of its associated quadrature (rounded up) [1], so if our trained quadrature ends up being second order (and

not third order as in the Gauss–Lobatto case), our SBP operator must necessarily be first order. Indeed, if the interior grid node x_1 is randomly placed inside the interval $(-1, 1)$, then training a quadrature $\mathbf{P}$ followed by a matrix $\mathbf{Q}$ results in a *first* order operator $\mathbf{D} = \mathbf{P}^{-1}\mathbf{Q}$.

3.1. Neural Network Based Differential Operators

Next, we demonstrate how to use single-layer linear neural networks for solving IBVPs in a stable manner. Suppose we are given a grid $\Omega_h = \{-1, -0.3, 0, 0.7, 1\}$ (there is nothing special about this grid, the interior points can be chosen arbitrarily). Our first step is to look for a diagonal quadrature matrix $\mathbf{P}$ on this grid. Or, equivalently, to train a neural network with five input neurons, a single output neuron (where the five neural weights represent the quadrature weights), a linear activation function, and zero bias. We will do this by training on polynomials as described above. As a side condition, our quadrature weights must be positive since negative quadrature weights will lead to instabilities in SBP-SAT schemes. For a five point grid, a fourth order quadrature is guaranteed to exist, but not necessarily with positive weights. For this specific Ω_h, it turns out that we need to drop to third order to find a quadrature with positive weights.

We compute such a quadrature using Tensorflow [7] with positive weight constraints and polynomials in $\mathbb{P}_3$ as training data (see Section 4.1 for details). More precisely, the input training data consists of the polynomials 1, x, x^2, x^3, evaluated in the grid nodes, i.e. the vectors

$$\begin{bmatrix} 1.000 \\ 1.000 \\ 1.000 \\ 1.000 \\ 1.000 \end{bmatrix}, \begin{bmatrix} -1.000 \\ -0.300 \\ 0.000 \\ 0.700 \\ 1.000 \end{bmatrix}, \begin{bmatrix} 1.000 \\ 0.090 \\ 0.000 \\ 0.490 \\ 1.000 \end{bmatrix}, \begin{bmatrix} -1.000 \\ -0.027 \\ 0.000 \\ 0.343 \\ 1.000 \end{bmatrix}.$$

The output training data set consists of the corresponding integrals of the same polynomials over the interval $[-1, 1]$. We exit the training loop when the loss function stops decreasing. When the training halts, the mean squared error on the training set is $\approx 5 \times 10^{-31}$, and the resulting quadrature, truncated to 7 decimals, is

$$\mathbf{P} = \mathrm{diag}(0.2758077, 0.5015049, 0.6813768, 0.3835038, 0.1578065)\,.$$

Next we train the matrix $\mathbf{Q}$. Recall that we want to enforce the identity $\mathbf{Q} + \mathbf{Q}^\top = \mathrm{diag}(-1, 0, 0, 0, 1)$ (i.e. the SBP property). A classical machine learning approach is to add a term to the loss function known as a regularizer. In this case we add the squared Frobenius norm

$$\|\mathbf{Q} + \mathbf{Q}^\top - \mathrm{diag}(-1, 0, 0, 0, 1)\|_F^2$$

to the loss function, penalizing deviation from the SBP property.

As stated above, the order of our resulting SBP operator can be no larger than 2. Hence, our input training data will be the polynomials 1, x, and x^2 evaluated in the grid nodes, and our output training data will be their respective derivatives 0, 1, and $2x$ evaluated in the grid nodes. Note that the matrices $\mathbf{P}$ and $\mathbf{Q}$ are typically not unique—the solutions we find will depend on the optimizer and on the initial weights.

Remark 3. *It may be tempting to throw more training data at the problem, e.g. higher order polynomials. This is not a good idea and will result in a useless operator, since we would be searching for an operator that does not exist. This issue generalizes to any kind of neural network which is not rich enough to capture the training data. In our case there is a clear theoretical basis for the choice of training data (we know that an SBP operator of half the order of the quadrature exists and any operator of higher order than that cannot be SBP), but that is almost never the case in general. Determining appropriate training data for a neural network usually involves guesswork and trial and error attempts.*

Remark 4. *The choice of polynomials as training data has a straightforward justification due to Taylor's formula—it will make our operator accurate for any smooth function on a small interval. It is conceivable that training on other kinds of functions could be effective, but so far we have seen no such indication.*

We compute an operator based on the regularizer described above. The weights in $\mathbf{Q}$ are initialized to 1. When the training halts, the mean squared error on the training set is 1.8220×10^{-16} and the resulting matrix $\mathbf{Q}$, truncated to 7 decimals, is

$$\mathbf{Q} = \begin{bmatrix} -0.5000000 & 0.3744424 & 0.2753372 & -0.1264006 & -0.0233790 \\ -0.3744424 & 0.0000000 & 0.1709185 & 0.2548713 & -0.0513474 \\ -0.2753372 & -0.1709185 & 0.0000000 & 0.3049724 & 0.1412834 \\ 0.1264006 & -0.2548713 & -0.3049724 & 0.0000000 & 0.4334430 \\ 0.0233790 & 0.0513474 & -0.1412834 & -0.4334430 & 0.5000000 \end{bmatrix}.$$

Note that this matrix does not satisfy the SBP property exactly, even though it does not show in the truncated version above since the maximum error is $\approx 10^{-16}$. Note also that it is essential that the training set is not too large. If the polynomial x^3 is added to the training set, the resulting operator:

$$\mathbf{Q} = \begin{bmatrix} -0.5053358 & 0.3481000 & 0.1073104 & -0.0461702 & 0.0619085 \\ -0.3449816 & -0.0017785 & 0.4340986 & -0.1159564 & 0.0554273 \\ -0.1056009 & -0.4354396 & 0.0002329 & 0.7239845 & -0.1924932 \\ 0.0444963 & 0.1182323 & -0.7223384 & -0.0001103 & 0.5268234 \\ -0.0595074 & -0.0566643 & 0.1899691 & -0.5263598 & 0.5030312 \end{bmatrix}$$

is very inaccurate with a mean squared error of $\approx 10^{-2}$.

Remark 5. *It is also possible to solve for* $\mathbf{Q}$ *using Keras (a high level API built on top of Tensorflow) and hard coded weight constraints. Keras is easier*

in terms of implementation, but we found that the accuracy of the resulting operators were much lower, and was increased by several orders of magnitude by directly using Tensorflow. High floating point precision is not a major concern in deep learning, hence some routines in Keras may be suboptimal in terms of accuracy.

3.2. Linear algebra based differential operators

The problem being solved in Section 3.1 can be phrased more succinctly in terms of two constrained linear systems—one to solve for the quadrature matrix $\mathbf{P}$, and one to solve for $\mathbf{Q}$. Suppose we would like to find positive quadrature weights based on the points $\{x_i\}$ such that the resulting quadrature is exact for polynomials up to order p. Finding such a quadrature is equivalent to the problem of finding $\mathbf{p}$ such that

$$\mathbf{V}_p^\top \mathbf{p} = \mathbf{b}, \quad \mathbf{p} > 0\,, \tag{7}$$

where $\mathbf{V}_p$ is the p:th order Vandermonde matrix:

$$\mathbf{V}_p = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^p \\ 1 & x_1 & x_1^2 & \cdots & x_1^p \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^p \end{bmatrix}, \tag{8}$$

$\mathbf{p}$ is a vector of quadrature weights (i.e. the diagonal elements of $\mathbf{P}$), and $\mathbf{b}$ is a vector of integrals of the monomials over the reference element. As stated earlier, the existence of solutions to (7) is not guaranteed in general; in practice we may have to either increase the number of points in the element, or lower the order of accuracy in order to find a solution (see [26, 27] for a more thorough discussion).

Suppose further that, given a quadrature matrix $\mathbf{P}$, we would like to find a matrix $\mathbf{Q}$ such that the discrete differential operator $\mathbf{D} = \mathbf{P}^{-1}\mathbf{Q}$ is exact for polynomials up to order q, and such that $\mathbf{Q} + \mathbf{Q}^\top = \mathrm{diag}(-1, 0, 0, \ldots, 1)$. The problem of finding $\mathbf{Q}$ can be stated more explicitly in the form of a constrained matrix equation:

$$\mathbf{Q}\mathbf{V}_q = \mathbf{P}\mathbf{V}_q', \quad \mathbf{Q} + \mathbf{Q}^\top = \mathrm{diag}(-1, 0, 0, \ldots, 1)\,, \tag{9}$$

where $\mathbf{V}_q$ is the q:th order Vandermonde matrix and $\mathbf{V}_q'$ is the derivative of the q:th order Vandermonde matrix:

$$\mathbf{V}_q' = \begin{bmatrix} 0 & 1 & 2x_0 & \cdots & qx_0^{q-1} \\ 0 & 1 & 2x_1 & \cdots & qx_1^{q-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 1 & 2x_N & \cdots & qx_N^{q-1} \end{bmatrix}. \tag{10}$$

4. Solution Procedures

The resulting systems of equations by the two procedures in Sections 3.1 and 3.2 can be solved in different ways. In this section we discuss the pros and cons of traditional methods and a machine learning approach.

4.1. Solving for $\mathbf{P}$ *and* $\mathbf{Q}$ *with Tensorflow*

Tensorflow is an open source machine learning software library that can be used to implement and solve a wide range of optimization problems. As the name suggests, Tensorflow is a suitable tool for problems formulated in terms of tensors. In particular, both equation (7) and equation (9) can be straightforwardly implemented as the minimization problems

$$\min_{\mathbf{p}} \quad \|\mathbf{V}_p^\top \mathbf{p} - \mathbf{b}\|_2^2 \tag{11}$$

$$\text{subject to} \quad \mathbf{p} > 0 \tag{12}$$

and

$$\min_{\mathbf{Q}} \quad \|\mathbf{Q}\mathbf{V}_q - \mathbf{P}\mathbf{V}_q'\|_F^2 \tag{13}$$

$$\text{subject to} \quad \mathbf{Q} + \mathbf{Q}^\top = \text{diag}(-1, 0, 0, \ldots, 1) \tag{14}$$

respectively. In Tensorflow parlance we may define the loss functions

$$L_p = \|\mathbf{V}_p^\top \mathbf{p} - \mathbf{b}\|_2^2 + \lambda_p \min\{\min(\mathbf{p}), 0\} \tag{15}$$

and

$$L_q = \|\mathbf{Q}\mathbf{V}_q - \mathbf{P}\mathbf{V}_q'\|_F^2 + \lambda_q \|\mathbf{Q} + \mathbf{Q}^\top - \text{diag}(-1, 0, 0, \ldots, 1)\|_F^2 \,, \tag{16}$$

where λ_p and λ_q are so-called regularization parameters (note that if $\lambda_p < 0$ and $\lambda_q > 0$, then the penalty terms $\lambda_p \min\{\min(\mathbf{p}), 0\}$ and $\lambda_q \|\mathbf{Q} + \mathbf{Q}^\top - \text{diag}(-1, 0, 0, \ldots, 1)\|_F^2$ add positive contributions to the loss functions as long as the constraints are not satisfied). The loss functions L_p and L_q are then minimized using Tensorflow's automatic differentiation. More precisely, Tensorflow computes the exact gradient of L_p with respect to $\mathbf{p}$; then $\mathbf{p}$ is updated as $\mathbf{p}^{(n+1)} = \mathbf{p}^{(n)} - \delta \nabla L_p$ (i.e. by gradient descent), where δ is chosen by one of Tensorflow's optimization algorithms (in our case the Adam method [28]). Once $\mathbf{p}$ is found (i.e. $L_p = 0$), it is used in L_q, which is also minimized using automatic differentiation and gradient descent to find $\mathbf{Q}$.

Note that $\mathbf{P}$ and $\mathbf{Q}$ can be thought of as single-layer linear neural networks, and minimizing L_p and L_q is equivalent to training the networks. With this perspective, the input training sets consist of polynomial evaluations at the grid nodes (i.e. the columns of $\mathbf{V}_p$ and $\mathbf{V}_q$ respectively). Similarly, the output

training sets are the integrals (i.e. the elements of the vector $\mathbf{b}$ in (7)), and the derivatives (i.e. the columns of $\mathbf{V}_q'$). We will consider another neural network style formulation applied to Burgers' equation in Section 6.2.

Remark 6. *Note that it is possible, but not straightforward, to convert the matrix equation* (9) *to a vector equation (primarily due to the need for the commutation matrix in the constraints conversion). This conversion may be highly nontrivial in certain situations. In higher dimensions we may lose the structure necessary to perform such a conversion in a straightforward manner. Tensorflow is a viable alternative to traditional linear solvers in such contexts, since it makes it simple to implement and solve matrix/tensor-based problems.*

4.2. Solving for $\mathbf{P}$ *and* $\mathbf{Q}$ *with linear solvers*

In order to solve the matrix equation (9) using a linear solver, it is more appropriately formulated as a vector equation. Let $\mathbf{q} = \text{vec}(\mathbf{Q})$ be the columns of $\mathbf{Q}$ stacked on top of each other. Then (9) is equivalent to the vector equation

$$(\mathbf{V}_q^\top \otimes I)\mathbf{q} = \text{vec}(\mathbf{P}\mathbf{V}_q'), \quad (\mathbf{I} + \mathbf{K})\mathbf{q} = (-1, 0, 0, \ldots, 1)^\top, \tag{17}$$

where $\mathbf{K}$ is a commutation matrix such that $\mathbf{K}\mathbf{q} = \text{vec}(\mathbf{Q}^\top)$.

Using a standard linear solver like Matlab's `lsqlin` to solve (17) is *much* faster than the Tensorflow approach (about two orders of magnitude in our tests). However, as mentioned above, to apply a linear solver we must reformulate the problem as a matrix-vector equation, which may be highly nontrivial in more complicated settings.

Solving (7) and (9) (or (17)) will yield an SBP operator $\mathbf{D} = \mathbf{P}^{-1}\mathbf{Q}$ with respect to the points $\{x_i\}$.

Remark 7. *In practice the equations* (7), (9), *and* (17) *are often ill-conditioned. For numerical accuracy it is recommended to always rescale elements to the reference element* $[-1, 1]$ *before solving for* P *and* Q.

5. Overlapping Grids

In higher dimensions it may be infeasible to compute elementwise custom operators for an entire domain. However for overlapping grids we may only need to use this approach in a small (the overlapping) region of the domain. Let us study a one-dimensional proof of concept.

Consider two overlapping uniform grids $\{x_i^l\}$ and $\{x_j^r\}$ as seen at the top of Figure 2. Transitioning from one grid to the other in a simulation requires special treatment of the overlap. The non-overlapping parts are well suited for finite difference based SBP operators due to the uniformity of the grids, whereas the overlap can be treated using the element based operators described in Section 3.2. More precisely we split the grids into two uniform sections and one irregular section at the overlap (as seen in the middle of Figure 2), introducing interfaces where the grid transitions from uniform to non-uniform. The overlap is further subdivided into suitably sized elements as seen at the bottom of Figure 2.

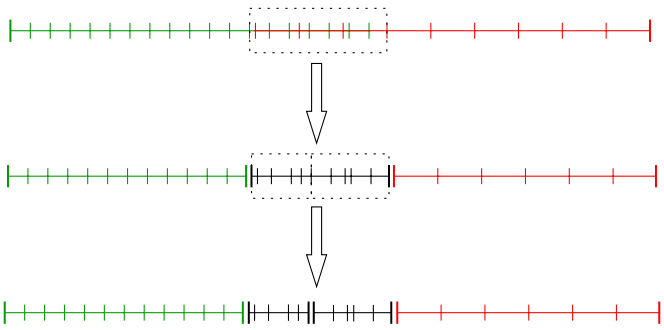

Figure 2: Two overlapping equidistant grids, producing a non-uniform grid on the overlapping part. The grids are split into two uniform parts and one irregular part at the overlap. The overlap is then further subdivided into elements.

Choosing the number of points per element in the overlap is a balancing act. More points leads to larger elements and higher order of accuracy. Fewer points leads to smaller elements and lower order of accuracy, but possibly better conditioned elements for the solution process. To each element in the overlap we associate an SBP operator by solving for $\mathbf{P}$ and $\mathbf{Q}$ as in Section 3.2, and to the uniform grids we associate standard finite difference based SBP operators. This means that all the sections of the grid can be coupled using stable SAT based interface treatments. For example, the advection–diffusion equation

$$u_t + au_x = \epsilon u_{xx} \tag{18}$$

and Burgers' equation

$$u_t + uu_x = 0 \tag{19}$$

both have known stable interface treatments given in [29, 4], that we can directly copy and get stability.

Remark 8. *It is tempting to stop at the middle part of Figure 2 and compute a single custom operator for the entire overlap. While this strategy is viable for small overlaps with few gridpoints, the operators described in the previous sections are not well suited for elements with a large number of gridpoints. It is more difficult to solve for high order operators, and if the order is too large the solver may fail to converge. Even if a high order operator is found, it typically makes more sense to subdivide the overlap into smaller, low order elements to avoid problems such as the one seen in Figure 3.*

6. Numerical Experiments

In this section we explore the idea from Section 5 by considering two model problems; advection–diffusion and Burgers' equation. The SBP operators can be computed using either of the methods from Section 4 and yield similar results

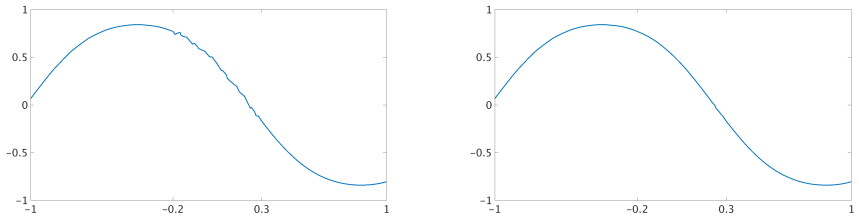

Figure 3: Advection computations on two overlapping domains $\Omega_l = (-1, 0.3)$ and $\Omega_r = (-0.2, 1)$, where Ω_l is discretized with 41 points and Ω_r is discretized with 61 points. In the left panel, a single 10th order operator is used in the overlapping section. This causes inaccuracy because the solution is not well approximated by a 10th order polynomial on the whole overlap. In the right panel, the overlap is subdivided into eight elements using 2nd order operators. Since the solution is well approximated locally in the elements, the disturbances seen in the left panel disappear.

in terms of convergence. We also discuss the difference in numerical accuracy between Matlab and Tensorflow, where SBP operators computed with Matlab performs slightly better. Finally we explore an accuracy relation between the quadrature P and the SBP operator D.

6.1. Advection–Diffusion

Let $a = 1$ and $\epsilon = 0.01$ and consider equation (18) posed on the domain $\Omega = (-1, 1)$. The domain is split into two overlapping subdomains $\Omega_l = (-1, 0.3)$ and $\Omega_r = (-0.2, 1)$. For well-posedness and stability, we impose a Robin condition $au - \epsilon u_x = g$ at $x = -1$, and a Neumann condition $\epsilon u_x = h$ at $x = 1$. The domain $\Omega = \Omega_l \cup \Omega_r$ is discretized using two uniform overlapping grids $\mathbf{x}^l = \{x_0^l, x_1^l, \ldots, x_N^l\}$ and $\mathbf{x}^r = \{x_0^r, x_1^r, \ldots, x_M^r\}$, which are subdivided as described in Section 5 (see Figure 2). We start with the grid sizes $N = 21$ and $M = 31$, and compute numerical solutions on increasingly fine resolutions by doubling the number of grid points. Recall that for the elements in the overlap we must choose the parameters p (the quadrature order), q (the differentiation order), and n (the number of points per element). Since a quadrature of order p is necessary and sufficient for the existence of an SBP operator of order $q = \lceil p/2 \rceil$ [1], for a given p we always select $q = \lceil p/2 \rceil$. Convergence rates for a few different configurations can be found in Tables 1, 2, and 3. Numerical solutions are computed using data from the manufactured solution $u = \sin(2\pi(x+t))$. The convergence rates are computed as $(\ln \|e_k\|_{L^2} - \ln \|e_{k+1}\|_{L^2})/\ln 2$, where $\|e_k\|_{L^2}$ and $\|e_{k+1}\|_{L^2}$ are the errors at time $t = 1$ for two successive grid resolutions.

The order of convergence is determined by the lowest order operators used in the scheme. A pure finite difference scheme based on SBP$(2q, q)$ operators will have a theoretical convergence rate of $q + 1$ (using wide operators for the diffusion term). Similarly, a purely element based method such as discontinuous Galerkin typically achieves a convergence rate of $q + 1$ when using q:th order operators. Indeed, we see in Table 1 that the overall convergence rate will not exceed 2 due to the SBP$(2, 1)$ operators used in the uniform parts, even if third

order operators are used in the overlap. Similarly, in Table 2, the convergence rate will not exceed 3 due to the SBP$(4, 2)$ operators used in the uniform parts. Finally, in Table 3, where SBP$(6, 3)$ operators are used in the uniform parts, we see that optimal order of convergence is only achieved with third order operators in the overlap.

6.2. Burgers' equation

Consider Burgers' equation

$$u_t + uu_x = 0, \quad x \in (-1, 1), \quad t > 0\,, \tag{20}$$

with a Dirichlet boundary condition $u = g$ imposed at $x = -1$ if $u(-1) > 0$, and a Dirichlet boundary condition $u = h$ imposed at $x = 1$ if $u(1) < 0$. It can be shown that an SBP-SAT discretization of the equivalent split formulation

$$u_t + \frac{1}{3}(uu_x + (u^2)_x) = 0, \quad x \in (-1, 1), \quad t > 0\,, \tag{21}$$

is stable [30, 31]. More precisely, let $\boldsymbol{\mathcal{Q}}(\mathbf{u}) = \mathbf{UQ} + \mathbf{QU}$, where $\mathbf{U} = \mathrm{diag}(\mathbf{u})$ and $\mathbf{Q}$ is the almost skew-symmetric matrix associated with an SBP operator $\mathbf{D} = \mathbf{P}^{-1}\mathbf{Q}$, and let $\boldsymbol{\mathcal{L}}(\mathbf{u}) = \frac{1}{3}\mathbf{P}^{-1}\boldsymbol{\mathcal{Q}}(\mathbf{u})$. Then the (single element) scheme

$$\mathbf{u}_t + \boldsymbol{\mathcal{L}}(\mathbf{u})\mathbf{u} = \sigma_l P^{-1}(u_0 - g)\mathbf{e}_0 + \sigma_r P^{-1}(u_N - h)\,, \tag{22}$$

where $\sigma_l = \min(0, -u_0)$ and $\sigma_r = \min(0, u_N)$, is stable. The scheme (22) can be extended in a straightforward manner to multiple elements as in Section 3 using the coupling procedure described in [4].

The operation $\boldsymbol{\mathcal{Q}}(\mathbf{u})\mathbf{u}$ can be seen as a computational graph formed by splitting its components into different layers as illustrated in Figure 4. Decomposing discrete operators as computational graphs make them simple to implement in the Tensorflow/Keras framework [7].

We repeat the convergence study from the previous section. The results can be seen in Table 4, showing the same kinds of patterns seen in Tables 1–3

Remark 9. *In the graph in Figure 4, all the components are precomputed. One may be tempted to train this network from scratch. If the underlying equation has*

nodes	error	rate
21/31	3.51e-02	
41/61	8.58e-03	2.1
81/121	2.13e-03	2.05
161/241	5.22e-04	2.05
321/481	1.30e-04	2.01

$n = 6$, $p = 3$, $q = 2$

nodes	error	rate
21/31	3.32e-02	
41/61	8.33e-03	2.07
81/121	2.08e-03	2.04
161/241	5.18e-04	2.02
321/481	1.29e-04	2.01

$n = 11$, $p = 5$, $q = 3$

Table 1: Advection–diffusion convergence rates for overlapping grids using SBP$(2, 1)$ operators on the uniform parts.

nodes	error	rate
21/31	1.43e-02	
41/61	2.98e-03	2.34
81/121	3.99e-04	2.95
161/241	4.72e-05	3.11
321/481	7.03e-06	2.76

$n = 6,\ p = 3,\ q = 2$

nodes	error	rate
21/31	1.09e-02	
41/61	1.23e-03	3.26
81/121	1.70e-04	2.9
161/241	1.79e-05	3.28
321/481	2.21e-06	3.03

$n = 11,\ p = 5,\ q = 3$

Table 2: Advection–diffusion convergence rates for overlapping grids using SBP$(4, 2)$ operators on the uniform parts.

nodes	error	rate
21/31	1.33e-02	
41/61	2.90e-03	2.27
81/121	3.78e-04	2.99
161/241	4.39e-05	3.13
321/481	6.68e-06	2.73

$n = 6,\ p = 3,\ q = 2$

nodes	error	rate
21/31	1.02e-02	
41/61	7.39e-04	3.92
81/121	1.09e-04	2.81
161/241	4.49e-06	4.65
321/481	3.10e-07	3.87

$n = 11,\ p = 5,\ q = 3$

Table 3: Advection–diffusion convergence rates for overlapping grids using SBP$(6, 3)$ operators on the uniform parts.

a known stable discretization, then we strongly advise against such an approach. It would severely reduce accuracy and introduce stability problems (it is not clear how to impose the SBP property for example). However, with no prior knowledge of the operator, and access to experimental data, training a discrete operator from scratch may be a reasonable approach.

6.3. Matlab versus Tensorflow

We study the efficacy of solving (7) and (17) using Matlab's `lsqlin` function versus a Tensorflow based implementation of (11)–(14). Consider for example the case where $N = 41$, $M = 61$, and $n = 11$, resulting in an overlapping section composed of four elements. Differentiation errors for the elements in the overlap can be seen in Figure 5. Our Tensorflow implementation uses the natural problem formulation (9) together with a gradient descent method. The fact that we can readily implement and solve matrix/tensor-based problems is the main attraction of Tensorflow; as stated earlier, the conversion from a constrained matrix equation to a vector equation is nontrivial, and should be avoided if possible. Things like the regularization parameters and step sizes must be manually tweaked in order to ensure convergence (in the sense that there is no built in system that does this for us—of course, in a more sophisticated implementation such parameter tweaking should be automated).

Typically the Matlab solver outperforms our Tensorflow approach by about an order of magnitude in terms of accuracy. It should also be noted that there is significant overhead in the Tensorflow optimization loop (likely because it was

nodes	error	rate
21/31	6.51e-02	
41/61	2.30e-02	1.55
81/121	6.66e-03	1.82
161/241	2.02e-03	1.74
321/481	5.86e-04	1.79

$n = 6$, $p = 3$, $q = 2$
SBP(2,1) in uniform part

nodes	error	rate
21/31	6.89e-02	
41/61	2.32e-02	1.63
81/121	6.64e-03	1.84
161/241	2.02e-03	1.73
321/481	5.87e-04	1.79

$n = 11$, $p = 5$, $q = 3$
SBP(2,1) in uniform part

nodes	error	rate
21/31	2.35e-02	
41/61	9.92e-03	1.29
81/121	6.29e-04	4.05
161/241	8.95e-05	2.84
321/481	1.33e-05	2.76

$n = 6$, $p = 3$, $q = 2$
SBP(4,2) in uniform part

nodes	error	rate
21/31	3.51e-02	
41/61	3.49e-03	3.45
81/121	4.09e-04	3.15
161/241	5.01e-05	3.06
321/481	5.84e-06	3.11

$n = 11$, $p = 5$, $q = 3$
SBP(4,2) in uniform part

nodes	error	rate
21/31	1.25e-01	
41/61	1.07e-02	3.67
81/121	6.32e-04	4.16
161/241	8.09e-05	2.99
321/481	1.22e-05	2.74

$n = 6$, $p = 3$, $q = 2$
SBP(6,3) in uniform part

nodes	error	rate
21/31	1.25e-01	
41/61	6.10e-03	4.52
81/121	4.09e-04	3.97
161/241	3.23e-05	3.69
321/481	2.10e-06	3.96

$n = 11$, $p = 5$, $q = 3$
SBP(6,3) in uniform part

Table 4: Burgers' convergence rates for overlapping grids.

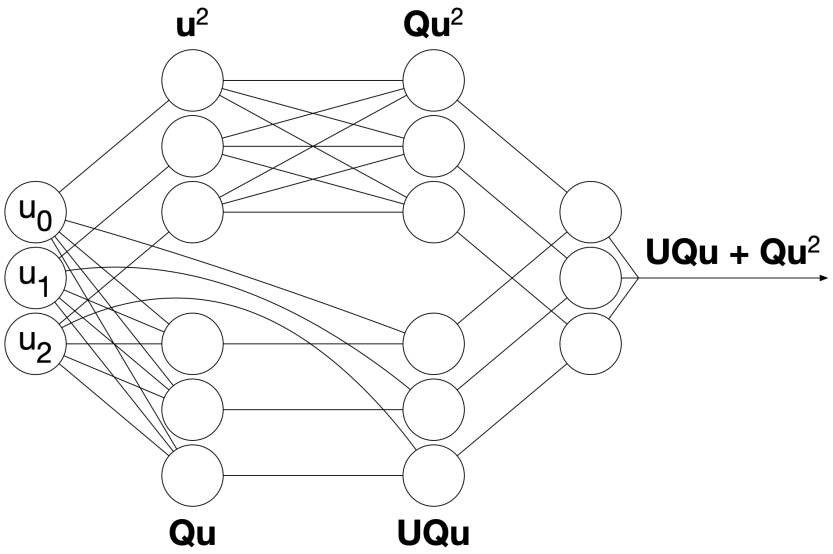

Figure 4: A computational graph representing the Burgers operation $\mathcal{Q}(\mathbf{u})$.

designed to operate on massive problems with millions of weights, where such overhead would be negligible). This results in execution times of around 200 times that of the Matlab solver. In our case the operators are still found in a matter of seconds, but for larger overlaps this will be an issue. The Tensorflow codebase is open source however, so it may be possible to adapt it smaller scale problems and remove the overhead.

6.4. P and D numerical accuracy relation

The numerical accuracy of P and D are closely linked. If the quadrature matrix P does not accurately integrate, then the SBP operator $D = P^{-1}Q$ does not accurately differentiate. To see this effect, consider the family of nodal distributions $\{-1, -0.7, 0.1 + \delta, 0.7, 1\}$, where $\delta \in [0, 0.2]$. In other words, from the starting nodal distribution $\{-1, -0.7, 0.1, 0.7, 1\}$, we shift the node located at 0.1 to the right toward 0.3. Let $p = 3$ be the desired order of the quadrature, and $q = 2$ the desired order of the SBP operator. We know from experiments that for some members of this family of nodal distributions, an accurate 3rd order quadrature with positive weights cannot be found. In Figure 6 we illustrate how the numerical accuracy of D follows the numerical accuracy of P.

7. Conclusions

We have shown that existing IBVP discretization theory can be combined with machine learning techniques to produce stable discretizations. By building the SBP property into a neural network we were able to use the network to construct provably stable schemes for a number of IBVPs.

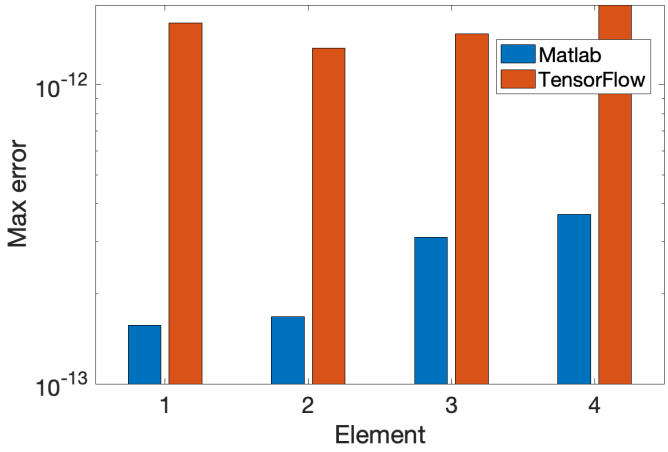

Figure 5: Maximum pointwise differentiation errors (i.e. $\max_{p\in\mathbb{P}_3} \|D\mathbf{p} - \mathbf{p}_x\|_\infty$) produced by computed operators on the elements in an overlapping section consisting of four elements. In terms of accuracy, the Matlab solver works better than the Tensorflow solver in this case.

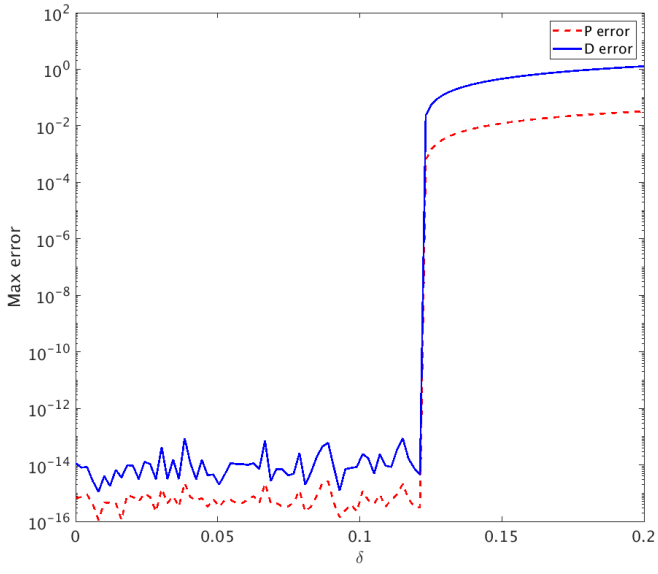

Figure 6: Maximum integration and differentiation errors for P and D on the nodal distribution $\{-1, -0.7, 0.1 + \delta, 0.7, 1\}$.

We have constructed provably stable, high order schemes for overlapping one-dimensional grids based on a hybrid FDM/FEM approach. Two methods of solving for SBP operators on elements with irregular nodal distributions were presented; one based on Matlab and a constrained vector equation, and another based on Tensorflow and a constrained matrix equation.

The Matlab approach outperforms the Tensorflow approach in terms of accuracy and efficiency—however, the convenience of being able to solve the matrix equation directly should not be underestimated, especially for multidimensional problems.

References

[1] David C Del Rey Fernández, Pieter D Boom, and David W Zingg. A generalized framework for nodal first derivative summation-by-parts operators. *Journal of Computational Physics*, 266:214–239, 2014.

[2] Jan Nordström, Karl Forsberg, Carl Adamsson, and Peter Eliasson. Finite volume methods, unstructured meshes and strict stability for hyperbolic problems. *Applied Numerical Mathematics*, 45(4):453–473, 2003.

[3] Magnus Svärd and Jan Nordström. Review of summation-by-parts schemes for initial–boundary-value problems. *Journal of Computational Physics*, 268:17–38, 2014.

[4] Gregor J Gassner. A skew-symmetric discontinuous galerkin spectral element discretization and its relation to sbp-sat finite difference methods. *SIAM Journal on Scientific Computing*, 35(3):A1233–A1253, 2013.

[5] David C Del Rey Fernández, Jason E Hicken, and David W Zingg. Review of summation-by-parts operators with simultaneous approximation terms for the numerical solution of partial differential equations. *Computers & Fluids*, 95:171–196, 2014.

[6] Ken Mattsson, Frank Ham, and Gianluca Iaccarino. Stable and accurate wave-propagation in discontinuous media. *Journal of Computational Physics*, 227(19):8753–8767, 2008.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[8] François Chollet. keras. https://github.com/fchollet/keras, 2015.

[9] Jens Berg and Kaj Nyström. A unified deep artificial neural network approach to partial differential equations in complex geometries. *Neurocomputing*, 317:28–41, 2018.

[10] Guofei Pang, Lu Lu, and George Em Karniadakis. fpinns: Fractional physics-informed neural networks. *SIAM Journal on Scientific Computing*, 41(4):A2603–A2626, 2019.

[11] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.

[12] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[13] Ernest K Ryu and Stephen P Boyd. Extensions of gauss quadrature via linear programming. *Foundations of Computational Mathematics*, 15(4):953–971, 2015.

[14] Fredrik Olsson and N Anders Petersson. Stability of interpolation on overlapping grids. *Computers & fluids*, 25(6):583–605, 1996.

[15] G Chesshire and William D Henshaw. Composite overlapping meshes for the solution of partial differential equations. *Journal of Computational Physics*, 90(1):1–64, 1990.

[16] William D Henshaw. A fourth-order accurate method for the incompressible navier-stokes equations on overlapping grids. *Journal of computational physics*, 113(1):13–25, 1994.

[17] Frank Ham, Ken Mattsson, Gianluca Iaccarino, and Parviz Moin. Towards time-stable and accurate les on unstructured grids. In *Complex Effects in Large Eddy Simulations*, pages 235–249. Springer, 2007.

[18] G Chesshire and William D Henshaw. A scheme for conservative interpolation on overlapping grids. *SIAM Journal on Scientific Computing*, 15(4):819–845, 1994.

[19] William D Henshaw and Donald W Schwendeman. Moving overlapping grids with adaptive mesh refinement for high-speed reactive and non-reactive flow. *Journal of Computational Physics*, 216(2):744–779, 2006.

[20] William D Henshaw and Donald W Schwendeman. Parallel computation of three-dimensional flows using overlapping grids with adaptive mesh refinement. *Journal of Computational Physics*, 227(16):7469–7502, 2008.

[21] Adam Reichert, Michael T Heath, and Daniel J Bodony. Energy stable numerical methods for hyperbolic partial differential equations using overlapping domain decomposition. *Journal of Computational Physics*, 231(16):5243–5265, 2012.

[22] Luis Guillermo Maria Reyna. *I. Stability of Tchebyshev collocation. II. Interpolation for surfaces with 1-D discontinuities. III. On composite meshes.* PhD thesis, California Institute of Technology, 1983.

[23] Fredrik Olsson and N Anders Petersson. Stability of interpolation on overlapping grids. *Computers & fluids*, 25(6):583–605, 1996.

[24] Yulia T Peet and Paul F Fischer. Stability analysis of interface temporal discretization in grid overlapping methods. *SIAM Journal on Numerical Analysis*, 50(6):3375–3401, 2012.

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

[26] M Wayne Wilson. Necessary and sufficient conditions for equidistant quadrature formula. *SIAM Journal on Numerical Analysis*, 7(1):134–141, 1970.

[27] Jan Glaubitz and Philipp Öffner. Stable discretisations of high-order discontinuous Galerkin methods on equidistant and scattered points. *Applied Numerical Mathematics*, 2020.

[28] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[29] M. H. Carpenter, J. Nordström, and D. Gottlieb. A stable and conservative interface treatment of arbitrary spatial accuracy. *Journal of Computational Physics*, 148:341–365, 1999.

[30] Jan Nordström. Conservative finite difference formulations, variable coefficients, energy estimates and artificial dissipation. *Journal of Scientific Computing*, 29(3):375–404, 2006.

[31] Travis C Fisher, Mark H Carpenter, Jan Nordström, Nail K Yamaleev, and Charles Swanson. Discretely conservative finite-difference formulations for nonlinear conservation laws in split form: Theory and boundary conditions. *Journal of Computational Physics*, 234:353–375, 2013.