MEMORY-EFFICIENT EMULATION OF PHYSICAL TABULAR DATA USING QUADTREE DECOMPOSITION

By

Jared Carlson

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Compuatational Mathematics, Science, and Engineering – Master of Science

2022

ABSTRACT

MEMORY-EFFICIENT EMULATION OF PHYSICAL TABULAR DATA USING QUADTREE DECOMPOSITION

By

Jared Carlson

Computationally expensive functions are sometimes replaced in simulations with an emulator that approximates the true function (e.g., equations of state, wavelength-dependent opacity, or composition-dependent materials properties). For functions that have a constrained domain of interest, this can be done by discretizing the domain and performing a local interpolation on the tabulated function values of each local domain. For these so-called tabular data methods, the method of discretizing the domain and mapping the input space to each subdomain can drastically influence the memory and computational costs of the emulator. This is especially true for functions that vary drastically in different regions. We present a method for domain discretization and mapping that utilizes quadtrees, which results in significant reductions in the size of the emulator with minimal increases to computational costs or loss of global accuracy. We apply our method to the electron-positron Helmholtz free energy equation of state and show over an order of magnitude reduction in memory costs for reasonable levels of numerical accuracy.

ACKNOWLEDGEMENTS

Jared Carlson acknowledges Matt Piekenbrock for assistance with the compact mapping scheme, and Bronson Messer, Adam Alessio, and Saiprasad Ravishankar for their input. BWO was supported in part by MSU's Office of Research and Innovation through the Institute of Cyber-Enabled Research, by NSF grants no. OAC-1835213 and AST-1908109, and by NASA ATP grants NNX15AP39G and 80NSSC18K1105. SMC is supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics, Early Career Research Program under Award Number DE-SC0015904. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and Office of Nuclear Physics, Scientific Discovery through Advanced Computing (SciDAC) program under Award Number DE- SC0017955. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) that are responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In scientific computing, we often deal with tabular data as an effective way of emulating (i.e., approximating) computationally expensive functions over a constrained domain. This is especially beneficial in cases where function evaluations happen frequently, are relatively close in the input domain, and/or are relatively smooth in the output space. In cases like these, it is often faster to evaluate many values of the function offline and then use them for interpolation during runtime [1]. Although this introduces a degree of error, tabulating more function values can reduce this error to the desired threshold.

The trade-off for the reduction in computational costs that are gained by doing an interpolation instead of evaluating the true function during runtime is the memory cost of storing the tabulated data. The tabular data used by the emulator must be held in memory and portions of it must be loaded to the compute device whenever an evaluation of the function is made. To address these issues, we propose a method of domain decomposition over the emulator's domain that uses a tree structure, as opposed to a uniform domain decomposition [2]. Although using a uniform decomposition has the benefit of $\mathcal{O}(1)$ computational complexity for mapping inputs to the correct cell, the error of the interpolation within each cell can vary drastically. As a consequence, the choice of cell size leads to under-refinement in some areas and/or over-refinement in others, which is especially evident when discontinuities are present or when the function changes rapidly in some regions of parameter space but varies slowly in others. Thus for a uniform table decomposition, the trade-off becomes either a higher error than desired or larger memory cost than needed. By using a tree structure, we show that this issue can be remedied with only a small additional cost of mapping inputs to the correct cell, i.e., traversing the tree.

The use of a tree structure to deal with areas of different optimal resolutions is a problem that has long been addressed in the field of image compression [3, 4]. Images are generally represented as a set of uniformly spaced cells, or pixels. One way to compress an image is to use quadtree

compression, which reduces neighboring groups of pixels that are close to the same color into a single larger pixel. Our method uses a similar technique but combines regions for which the expected interpolation error is low. Our method is also analogous to the adaptive mesh refinement technique, which is frequently used in large-scale physics simulations [5], and a variety of techniques in computer science [6].

Another constraint of common tabular methods is the requirement that all cells do the same type of interpolation. This restriction limits the type of interpolation scheme that can be used, as they must behave well throughout the emulator's domain. We show that adaptively selecting the type of interpolation scheme used on a cell-by-cell basis can decrease the resulting interpolation error, further allowing for smaller tables.

As a proxy problem, we will use interpolation of the electron-positron Helmholtz free energy [7, 8], which is used in modeling the equation of state of stellar plasmas. Although this free energy and resulting equation of state are relatively simple, they have characteristics features that are representative of (and thus can be generalized to) other, more complex, equations of state and opacities, examples of which are shown in [9, 10, 11, 12, 13, 14]. In these cases, the tabular data is supplied in a uniform grid and often without any interpolation scheme included [8, 15, 16, 17, 12, 18]. This leaves the interpolation of table values up to the application using that data, which typically resorts to an N-dimensional linear interpolation scheme [19, 13], or less commonly a more sophisticated higher-order polynomial-based interpolation [8]. As a consequence, the method described here can therefore be readily applied to existing tabular data sets. We will show that by using our method on the electron-positron Helmholtz free energy we can reduce the memory consumption of the table by at least an order of magnitude, while still maintaining comparable levels of accuracy to standard interpolating schemes.

This paper is organized as follows. In Section 2, we present our new method for the construction of an emulator using quadtree decomposition. We then present the results obtained from an implementation of this method in Section 3. Finally, we discuss our findings and present our conclusions in Sections 4 and 5, respectively.

The code for the quadtree emulator is available at https://doi.org/10.5281/zenodo.4770200. The data sets used for training and testing the quadtree emulator are available at https://doi.org/10.5281/zenodo.4739173.

An N-dimensional version of the code is publicly available at https://github.com/Carlson-J/ND-tree_tabular_data_emulator. This code utilizes additional performance optimizations, which is documented in the repository.

CHAPTER 2

METHODS

2.1 Emulating the Electron-Positron Helmholtz Free Energy

The electron-positron Helmholtz free energy $f = f(\rho, T)$ is a function of density ρ and temperature T and is used in astrophysical simulations of stellar phenomena to calculate the contribution of electrons and positrons of arbitrary degeneracy and relativity to the total fluid equation of state [20], and as a consequence its value is needed multiple times per volume element per time step. A visualization of f is shown in Figure 2.1(a). Because the inputs and outputs of f vary over orders of magnitude, Figure 2.1(a) shows $\log_{10}(|f|)$, with the ρ, T domain also transformed by $\log_{10}(\cdot)$.

Due to the high computational cost of evaluating this function, values of f are computed prior to simulation execution and tabulated. At runtime, the tabulated values of f are interpolated between to estimate f at a given density and temperature. The domain over which the values of f are tabulated depends on the simulation being run. Figure 2.1(a) shows a typical range in density and temperature.

Once the domain of interest is determined it is decomposed into a complete disjoint set of cells, i.e., there is no overlap between the domains of the cells and the union of the cells' domains covers the entire domain of interest. The standard method, developed by Timmes & Swesty 2000 [8], referred to hereafter as TS00, does the decomposition by creating a grid of cells over the domain that are evenly spaced in log-space. This allows a relatively small number of cells to cover densities and temperatures that vary over many orders of magnitude.

Each cell has a corresponding model that approximates f within the cell's domain. Here we define a model as a fully defined mapping (no free parameters) from the input domain $\mathscr{R}^2$ to the output space $\mathscr{R}^6$, where the outputs are f and its first-order and second-order derivatives.

The TS00 method uses a model class of biquintic polynomials to perform the interpolation, which we will refer to as the linear-space model class because we are interpolating in the linear-space, i.e., we do not do any log transforms on the domain or function values. This is opposed to the

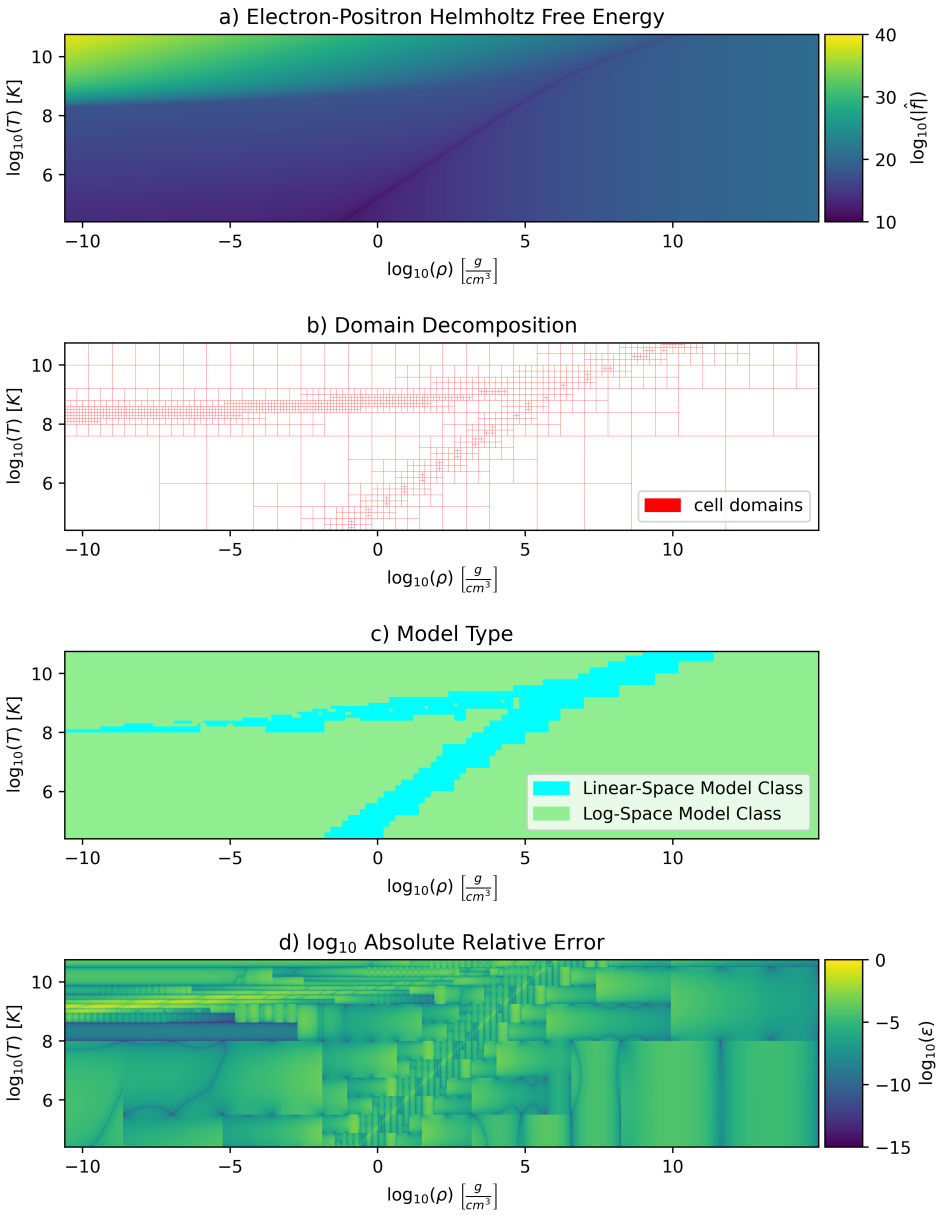


Figure 2.1: An example of various aspects of the quadtree decomposition with a max depth ε_{th}=6 and a relative error threshold $D_{\text{max}} = 10^{-3}$. Four quadtrees, placed side-by-side along the x-axis of the table, were trained to cover this domain. a) The $\log_{10}$ of the absolute value of the electron-positron Helmholtz free energy f. b) The domain decomposition. c) The mapping of the linear-space and log-space model classes to the decomposed domain. d) The relative error of the resulting emulator.

log-space model class, which will be introduced later, where both the domain and function values are transformed using the log function.

We define a model class as an under-defined mapping between the input domain and output space. A model class is trained for each cell, during which its parameters are fixed such that the resulting model best emulates f within the cell's domain. For the linear-space model class, these free parameters consist of the value f and several derivatives at four points, one at each corner of the cell,

$$\left[f, \frac{\partial f}{\partial \rho}, \frac{\partial f}{\partial T}, \frac{\partial^2 f}{\partial \rho^2}, \frac{\partial^2 f}{\partial T^2}, \frac{\partial^2 f}{\partial \rho \partial T}, \frac{\partial^3 f}{\partial \rho^2 \partial T}, \frac{\partial^3 f}{\partial \rho \partial T^2}, \frac{\partial^4 f}{\partial \rho^2 \partial T^2}\right],$$

where the partial derivatives of f are in terms of the density ρ and temperature T. All values except for the third and fourth-order derivatives can be computed exactly from values obtained through the equations of state,

$$f = E - TS \tag{2.1}$$

$$\frac{\partial f}{\partial \rho} = \frac{P}{\rho^2} \tag{2.2}$$

$$\frac{\partial f}{\partial T} = -S \tag{2.3}$$

$$\frac{\partial^2 f}{\partial \rho^2} = -2\left(\frac{\partial f}{\partial \rho}\frac{1}{\rho} + \frac{\partial P}{\partial \rho}\frac{1}{\rho^2}\right) \tag{2.4}$$

$$\frac{\partial^2 f}{\partial T^2} = -\frac{\partial S}{\partial T} \tag{2.5}$$

$$\frac{\partial^2 f}{\partial \rho \partial T} = -\frac{\partial S}{\partial \rho}, \tag{2.6}$$

where E, S, and P are the electron-positron energy, entropy, and pressure respectively, and their explicit dependence on ρ and T have been removed for clarity. To compute the higher-order terms, $\frac{\partial^3 f}{\partial \rho^2 \partial T}$, $\frac{\partial^3 f}{\partial \rho \partial T^2}$, and $\frac{\partial^4 f}{\partial \rho^2 \partial T^2}$, we use a sixth order-accurate finite difference scheme [21], using $\frac{\partial^2 f}{\partial \rho \partial T}$ as an input.

We now present a new model class, a biquintic interpolation in the log-space, which we will refer to as the log-space model class. This requires taking the $\log_{10}(\cdot)$ of the inputs, $\log_{10}(|f|)$, and

transforming the derivatives terms as follows,

$$s = \text{sign}(f) \tag{2.7}$$

$$f^* = \log_{10}(|f|) \tag{2.8}$$

$$\rho^* = \log_{10}(\rho) \tag{2.9}$$

$$T^* = \log_{10}(T) \tag{2.10}$$

$$\frac{\partial f^*}{\partial \rho^*} = \frac{\rho s}{f}\frac{\partial f}{\partial \rho} \tag{2.11}$$

$$\frac{\partial f^*}{\partial T^*} = \frac{T s}{f}\frac{\partial f}{\partial T} \tag{2.12}$$

$$\frac{\partial^2 f^*}{\partial \rho^{*2}} = \frac{\ln(10)\rho}{f^2}(fs(\frac{\partial f}{\partial \rho} + \frac{\partial^2 f}{\partial \rho^2}\rho) - (\frac{\partial f}{\partial \rho})^2\rho) \tag{2.13}$$

$$\frac{\partial^2 f^*}{\partial T^{*2}} = \frac{\ln(10)T}{f^2}(fs(\frac{\partial f}{\partial T} + \frac{\partial^2 f}{\partial T^2}T) - (\frac{\partial f}{\partial T})^2T) \tag{2.14}$$

$$\frac{\partial^2 f^*}{\partial \rho^* \partial T^*} = \frac{\ln(10)\rho T}{f^2}(fs\frac{\partial f^2}{\partial T \partial \rho} - \frac{\partial f}{\partial T}\frac{\partial f}{\partial \rho}). \tag{2.15}$$

The higher-order derivatives are computed the same way as in the linear-space model class, but on the transformed values, i.e., using $\frac{\partial^2 f^*}{\partial \rho^* \partial T^*}$, T^*, and ρ^*. This model class requires an additional parameter, s, that holds the sign of f before it is transformed.

2.2 Defining Error

For the electron-positron Helmholtz free energy, the error of interest is the relative error (also called fraction error), defined as

$$\varepsilon = \left|\frac{f - \hat{f}}{\hat{f}}\right|, \tag{2.16}$$

where $\hat{f}$ is the true value and f is the predicted value. We choose this error because the magnitude of f changes by many orders of magnitude throughout the domain, so it is the relative error that is of primary interest when using this quantity in simulations.

To estimate the error in a region, we first compute ε at a set of points within the region. The points at which we compute ε will either be from our training set or testing set, depending on whether we are training the emulator or predicting the error of a trained emulator, respectively (see

Section 2.7 for details). We then compute three different norms, defined as

$$\mathrm{L}_1 = \frac{||\varepsilon||_1}{N} = \frac{1}{N}\sum_i^N |\varepsilon_i| \tag{2.17}$$

$$\mathrm{L}_2 = \frac{||\varepsilon||_2}{\sqrt{N}} = \frac{1}{\sqrt{N}}\left(\sum_i^N |\varepsilon_i|^2\right)^{\frac{1}{2}} \tag{2.18}$$

$$\mathrm{L}_\infty = ||\varepsilon||_\infty = \max_i(|\varepsilon_i|). \tag{2.19}$$

When training the emulator, we use the L_∞ norm to estimate the error in a region.

When the prediction is done in the log-space, e.g., using the log-space model class, we first transform it back into the linear-space before doing the error calculation. This is because the value that is used in simulations is the linear-space value. Doing the transformation before computing the error also takes into account any numerical error that is introduced due to the transformations. The transform of f^* back into linear-space is simply $f = s10^{f^*}$.

2.3 Combining Model Classes into a Single Emulator

Due to the changing characteristics of f throughout the domain covered by the emulator, certain model classes perform better in different regions. Figure 2.2(a & b) show the error of f using the linear-space and log-space model classes respectively. As seen in the figure, the log-space models perform better than the linear-space models in most, but not all, regions.

The simplicity of using a single model class over the entire domain comes at the cost that it must be able to capture all features of the function. This eliminates the possibility of using only the log-space model class as it fails to emulate portions of the domain, specifically when f crosses zero. Although the linear-space model class can capture all of the features, it has a higher error in regions where the log-space model class excels.

At the cost of additional complexity, we use both model classes throughout the domain. We do this by choosing the model class on a cell-by-cell basis, where the model class with the lowest estimated error over the cell's domain is used (see Section 2.2). The error over the domain when using both model classes in the same emulator is shown in Figure 2.2(c).

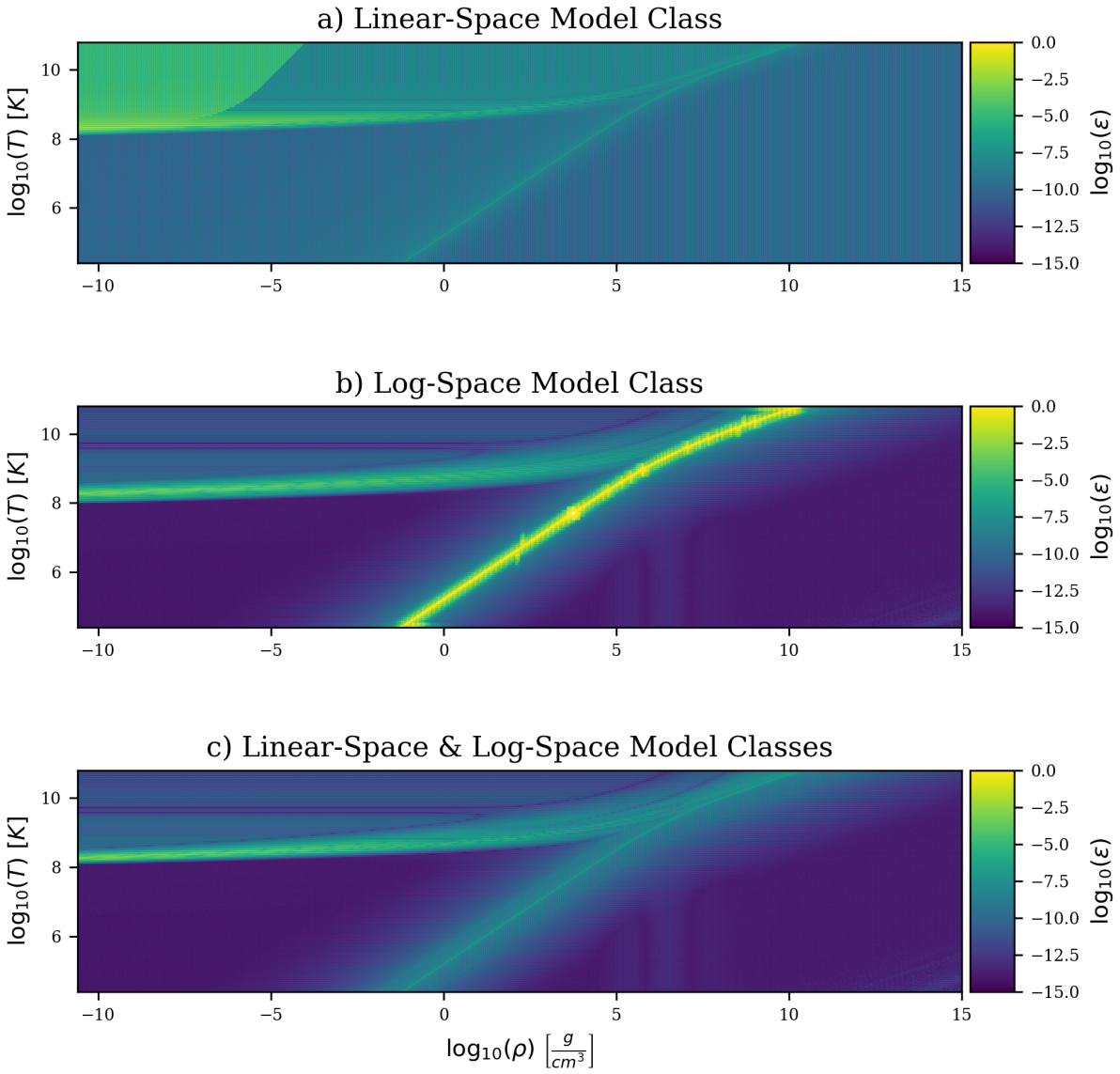


Figure 2.2: The $\log_{10}$ relative error of different model classes at a fixed uniform cell size. (a) linear-space model class, (b) log-space model class (c), and linear-space and log-space model classes. For (c), the best model class is chosen on a cell-by-cell basis such that the predicted error is minimized.

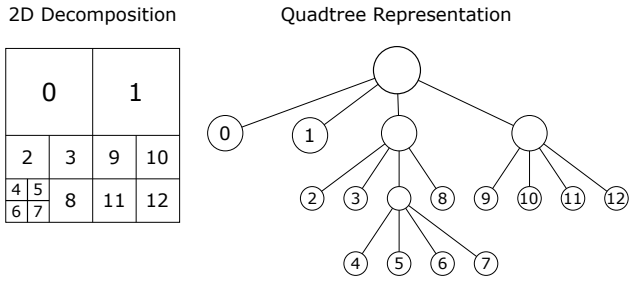


Figure 2.3: An example of a quadtree. Each leaf node is characterized by a number, which corresponds to a region in the 2D decomposition. The ordering of the numbers follows how the leaf nodes are laid out in memory. Cells close together in 2D space tend to be close together in memory as well.

2.4 Domain Decomposition using a Quadtree

Although using multiple model classes decreases the error in many regions, the magnitude of error still changes drastically throughout the domain. This places a constraint on the size of the cells, which has to be chosen such that the region with the highest error is within some threshold. This leads to regions where the cell spacing is much smaller than needed, creating excess memory requirements.

To reduce the memory requirements, we use a quadtree to decompose the domain into a complete disjoint set of cells, an example of which is shown in Figure 2.3. A quadtree is a recursive data structure consisting of nodes. Each node has either four or zero child nodes. Nodes that have no children are called leaf nodes. All nodes have a single parent node with the exception of the root node, which has no parent.

To traverse the tree, an input is first given to the root node. The root node then does a comparison and sorts it to one of its child nodes. This process is repeated recursively until a leaf node is reached, at which point data in the leaf node is returned. In our case, the returned value would be the parameters defining the model for the leaf node's corresponding cell.

To decompose the 2D input space (density and temperature), each node of the quadtree is defined by four boundaries (top, bottom, left, and right), forming a rectangle in the 2D space. The input for the quadtree is a pair of density and temperature values (T, ρ) that are within the domain of the root node. The root node sorts inputs into one of the four Cartesian quadrants of its domain, with

the origin being the midpoint of all four boundaries. The ordering of the cells follows the Morton coding scheme [22]. We use the Morton ordering to reduce memory lookup costs, as it tends to keep things close in the domain near in index space and therefore near in memory. The input value is then passed to the child node that corresponds to that region. This process is repeated recursively until a leaf node is reached.

Each leaf node corresponds to a rectangular cell in the domain. The quadtree, therefore, defines a subjective function from the root node's domain to the set of leaf nodes. Each cell/leaf node has a corresponding model that is defined over its domain.

2.5 Training the Quadtree Emulator

Training the quadtree emulator entails recursively refining cells until the estimated error is below a given threshold ε_{th} or the maximum depth D_{max} of the tree has been reached. As discussed in Section 2.2, the error within a cell is estimated by computing the error at multiple points within the cell's domain and taking the max of those errors.

To refine a cell, the cell is divided into four Cartesian quadrants with the origin centered at the midpoint of its four boundaries. A new leaf node is then created for each quadrant. An example of how the domain is decomposed after training is shown in Figure 2.1(b), where four quadtrees have been put side-by-side along the x-axis to cover a domain of interest.

The determination of which model class to use for each cell is the same as described in Section 2.3. However, because the cells are no longer uniformly distributed, two density and two temperature values that specify the domain of each cell must be stored with the weights of the model, as it can no longer be determined based solely on its index. An example of how different model classes are mapped to different regions is shown in Figure 2.1(c).

2.6 Creating the Memory Compact Quadtree Emulator

Once the quadtree has been built, it is saved in a way that is memory efficient. Since the tree is static once built, it is converted into a set of arrays holding the information about the models and

how to map the input space to these models.

For each model class, the model parameters are stored in a 2D array. Each row corresponds to a single model and consists of the values needed for the model to perform predictions. For the linear-space model class, this requires 40 64-bit floats[1]. For the log-space model class, 41 64-bit floats are required, the extra float being the sign variable. The order of the rows follows the ordering of the quadtree, shown in Figure 2.3, which tends to keep cells close in the domain close in memory.

Mapping an input to the correct model requires determining which cell the input is in and where the corresponding model is stored in memory. To do this we first map the input to its corresponding Cartesian index space. This is done by creating a grid of cells over the domain that are uniformly spaced in each dimension. The dimension of the grid is $2^{D_{\max}}$ by $2^{D_{\max}}$, i.e., a fully refined quadtree of depth $D_{\max}$. The i_ρ, j_T cell that contains the density and temperature input can then be computed by

$$i_\rho = \text{floor}\left(\frac{\rho - \rho_{\min}}{\Delta_\rho}\right), \text{ where } \Delta\rho = \frac{\rho_{\max} - \rho_{\min}}{2^{D_{\max}}}$$
$$j_T = 2^{D_{\max}} - 1 - \text{floor}\left(\frac{T - T_{\min}}{\Delta_T}\right), \text{ where } \Delta_T = \frac{T_{\max} - T_{\min}}{2^{D_{\max}}}.$$

Note that the difference in j_T compared to i_ρ is because we started our Morton ordering at the $T_{\max}$ instead of the $T_{\min}$.

These indices can then be mapped to the quadtree index space by constructing an index at the bit level. Let $i_\rho = x_1x_2x_3...x_N$ and $j_T = y_1y_2y_3...y_N$ be N-bit unsigned integers corresponding to the density and temperature indices respectively, with each x_n or y_n corresponding to the nth bit. The index in the quadtree index space is computed by $k = x_1y_1x_2y_2x_3y_3...x_Ny_N$, yielding a 2N-bit unsigned integer [23]. Figure 2.4(a) shows an example of the quadtree indices for a uniform grid of cells.

We then create a mapping from the quadtree index space to the correct model, e.g., mapping the cell index in Figure 2.4(a) to the correct model in Figure 2.4(b), such that cell's domain in (a) is within the model's domain in (b). To do this, we first create a mapping array M of size $4^{D_{\max}}$ ($D_{\max}$

[1]This is 4 more variables then used by TS00 since the density and temperature ranges must also be stored

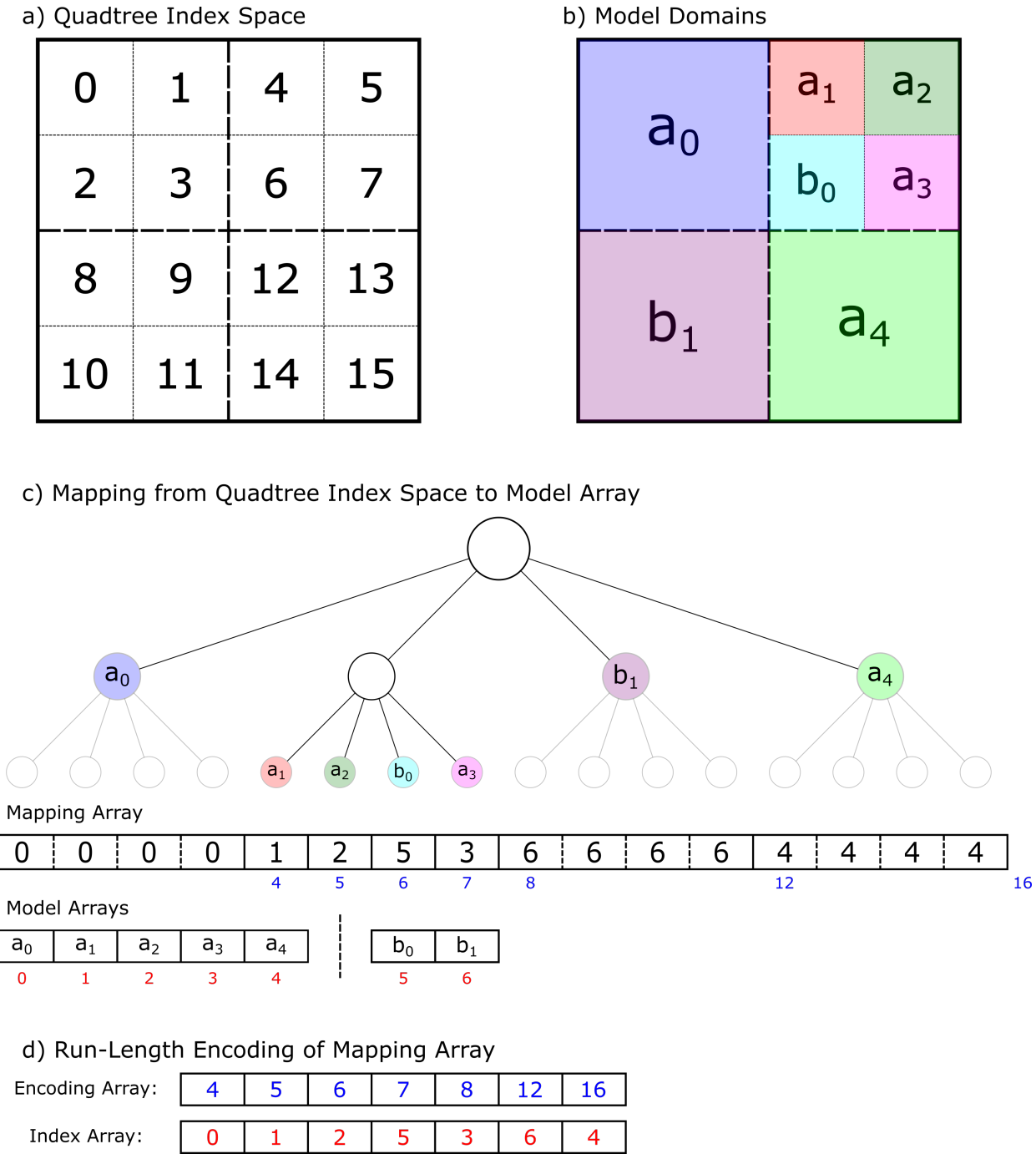


Figure 2.4: An example of the compact mapping scheme of a quadtree with $D_{\max}$=2. (a) The 2D domain is discretized into a uniform grid corresponding to a quadtree that is fully refined at $D_{\max}$=2, with indices matching the quadtree index space of its leaf nodes. (b) The domain of each model after the quadtree emulator has been trained. (c) The mapping array stores the index mapping of the quadtree index space to the model array. (d)) The mapping array is compactly represented using run-length encoding.

is the max depth of the quadtree). Each entry's index in this array corresponds to a cell on a fully refined quadtree, shown in Figure 2.4(c). The value of each entry is an integer that gives the index of the model in the model array. When multiple models are being used at once, offsets are saved that separate the arrays. For example, if one model array has 5 models and the second one had 2, then the offset would be 5. If the integer in M is 6 it would map to 1 in the second model array, e.g., b_1 in the example in 2.4(c).

To reduce the size of M, which is $4^{D_{\max}}$, we use run-length encoding to reduce the redundancy in the mapping array. This results in two arrays. The first is an array of indices corresponding to where values in M change, i.e., the indices where $M[k-1] \neq M[k]$. We call this the encoding array. The second array contains the values of M, which we call the index array. An example of this is shown in Figure 2.4(d).

Using run-length encoding eliminates all repeated values in our mapping scheme since it is laid out with the same ordering as the quadtree. This gives both the index and encoding arrays the same number of entries as the total number of models m. Using this encoding scheme adds an additional cost of searching over the encoding array. Since the encoding array is sorted, this is a $\mathcal{O}(\log_2 m)$ search.

To further reduce the memory cost of our mapping scheme, the integer size of each array is determined by the number of models and the maximum number of cells. For the encoding array, this is the smallest unsigned int such that $4^{D_{\max}}$ can be represented. For the index array, the smallest unsigned int is used such that m can be represented.

2.7 Emulating the Helmholtz Free Energy

We choose a density ρ range of $[10^{-10.6}, 10^{15}]$ (g/cm^3) and a temperature T range of $[10^{4.4}, 10^{10.8}]$ (K). This is a subset of the ρ and T ranges spanned by the default TS00 implementation, $[10^{-12}, 10^{15}]$ (g/cm^3) and $[10^3, 10^{13}]$ (K) respectively. We choose these ranges because they capture the different characteristic regions of f and because we can directly use the function values and derivatives used in TS00. The latter allows for direct comparison between our method and TS00,

as we can use the values stored in TS00's data file as our training data. This choice of ranges also allows us to separate the table into four square sections, each of which is decomposed by a quadtree.

Two different $D_{\max}$values are used on this domain, 7 and 9. With $D_{\max}$=7, we have the same cell size and locations as TS00 in areas where the quadtree is fully refined. When $D_{\max}$=9, the cell edges are four times smaller than TS00's cells. When doing cell refinement and mapping, the domain is viewed in log-space.

The training data that is used for the quadtree emulator with $D_{\max}$=7 is obtained using the exact equation of state [20] for up to and including the second-order derivatives, with the higher-order derivatives being computed using a finite difference method as described in Section 2.1. At full refinement and using the linear-space model class, this quadtree emulator is equivalent to TS00, allowing for direct comparison. The spacing in both dimensions is 1/20 (0.05) ($\log_{10}(g/cm^3)$ and $\log_{10}(K)$). Similarly, the training data for the quadtree emulator with $D_{\max}$=9 is generated in the same way, except the spacing of data in both dimensions is 1/160 (0.00625) ($\log_{10}(g/cm^3)$ and $\log_{10}(K)$), which is half the size of a cell at max refinement (1/80) ($\log_{10}(g/cm^3)$ and $\log_{10}(K)$).

For each $D_{\max}$, we run at a ε_{th} of 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8} and 10^{-9}. At each refinement step, we first select the points from the training set that lie within the cell's domain. If more than 100 data points are selected, we randomly choose 100 of them. The predicted values for these points are computed and compared to the true values. The maximum relative error of these predictions is then computed and used as an estimate of the cell's error.

Each combination of $D_{\max}$ and ε_{th} produces four different emulators, one for each of the four sections of the domain[2]. These emulators are then saved in HDF5 files[3] using the mapping and encoding scheme discussion in Section 2.6. The memory cost for each $D_{\max}$ and ε_{th} combination is computed by the sum of the four corresponding HDF5 files' sizes[4]. For TS00 we compute the memory cost analytically by examining the number of double precision variables used for the

[2]The domain we wish to model is rectangular, so stacking four quadtrees side-by-side in the density axis covers the domain of interest.

[3]No data compression is used when saving the emulators using the HDF5 format.

[4]This is a slight overestimate of the memory cost because it includes metadata stored in the HDF5 file as part of its self-describing file format, but which is not part of the table itself. There is also no data compression used when saving these HDF5 files.

electron-positron Helmholtz free energy interpolation, which are stored in a plain text file.

To estimate the error over the domain, we create a test set of data. This data is generated over the same domain except it is offset by (1/480) and has a spacing of (1/240) ($\log_{10}(K)$ and $\log_{10}(g/cm^3)$). This guarantees that the test and training sets are disjoint sets and that there are 9 test points in each cell for the smallest possible cell. These 9 points also sample near the midpoints between the cells and at the cell centers, which are observed to be the regions of the highest error. We then compute the error at each point and use three different norms, defined in Equations 2.17-2.19, to estimate the global error.

CHAPTER 3

RESULTS

Using the setup outlined in Section 2.7, we compare the expected relative norm errors between different emulators. Figures 3.1 and 3.2 show the estimated norm relative error vs. total table size in megabytes (MBs).

Figure 3.1 shows the results for a quadtree emulator with $D_{\max}$=7. When at full refinement the domain decomposition is equivalent to TS00, i.e., the domain decomposition is a uniform grid of cells with edge lengths of $1/2^7$ the size of the full domain. Three different norm errors, as defined by Equations (2.16)-(2.19), are represented by different colors. The solid lines correspond to using both the log-space and linear-space model classes, where the model class that has the lowest expected error is chosen on a cell-by-cell basis. The dashed lines correspond to just using the linear-space model class, which is equivalent to the TS00 interpolation scheme. Each point along these lines corresponds to a different ε_{th}=[10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}], from left to right respectively.

The stars correspond to the memory and error of TS00. The triangles correspond to an emulator that has the same uniform domain decomposition as TS00 but uses both the log-space and linear-space model classes[1]. Figure 3.2 is similar to Figure 3.1 except $D_{\max}$=9.

In both Figures 3.1 and 3.2, the error stops decreasing at a given table size. This is due to the maximum refinement being reached in many regions before ε_{th} is achieved, i.e, there are regions where the error cannot be reduced further given the maximum depth of the quadtree $D_{\max}$. In Figure 3.1, this happens early, where the data size has reached ~ 2 MBs. In Figure 3.2 the error stops decreasing when the data size has reached ~ 10 MBs. After these points, the error in the regions that have reached maximum refinement dominate. Lowering the ε_{th} further thus has little effect on the norm errors, but does increase the size of the table.

[1]The slight increase in table size of these over TS00 is due to the extra sign factor that must be stored for the log-space models.

The normalized[2] density distribution of the $\log_{10}(|\cdot|)$ error is shown in Figure 3.3. Each row corresponds to a different ε_{th}, shown by the vertical red dashed line. The normalized error density distribution is shown for TS00 and quadtree emulators with D_{max} being 7 and 9, using linear-space and combined linear-space & log-space model classes. As ε_{th} decreases, so does the majority of the error. The cases where the error is exceeding ε_{th} are due to two factors: 1) inaccuracies in our estimated maximum error based on our training data, which is expected, and 2) cells having reached maximum refinement before the error is reduced below ε_{th}. In the first row, the errors above ε_{th} are due mainly to 1), while in the last rows it is mainly due to 2). The D_{max}=7 combined linear-space and log-space model class has noticeably less error above ε_{th} compared to TS00, while the D_{max}=9 quadtree emulator offers drastic improvements in errors above ε_{th}. The improvement of the D_{max}=7 linear-space model class above ε_{th} is due to the increased accuracy in computing the higher-order derivatives compared to TS00. In the case where the data used in the quadtree emulator is the same as used in TS00, the results are equivalent.

Figure 3.4 shows the $\log_{10}(|e|)$ over the domain for three different cases: (a) TS00, (b) a quadtree emulator that has the same norm error as TS00 (D_{max}=7 and ε_{th}=10^{-4}, corresponds to the fourth point from the left on the solid line in Figure 3.1), (c) a quadtree emulator that is the same size as TS00 (D_{max}=9 and ε_{th}=10^{-8} corresponds to the eighth point from the left in Figure 3.2). This Figure shows that the quadtree emulator can maintain roughly consistent accuracy over its domain, leading to either significant memory savings or increased accuracy when compared to interpolation on a standard data table that is uniformly spaced in the relevant independent variables.

[2]The integration of the distribution is unity.

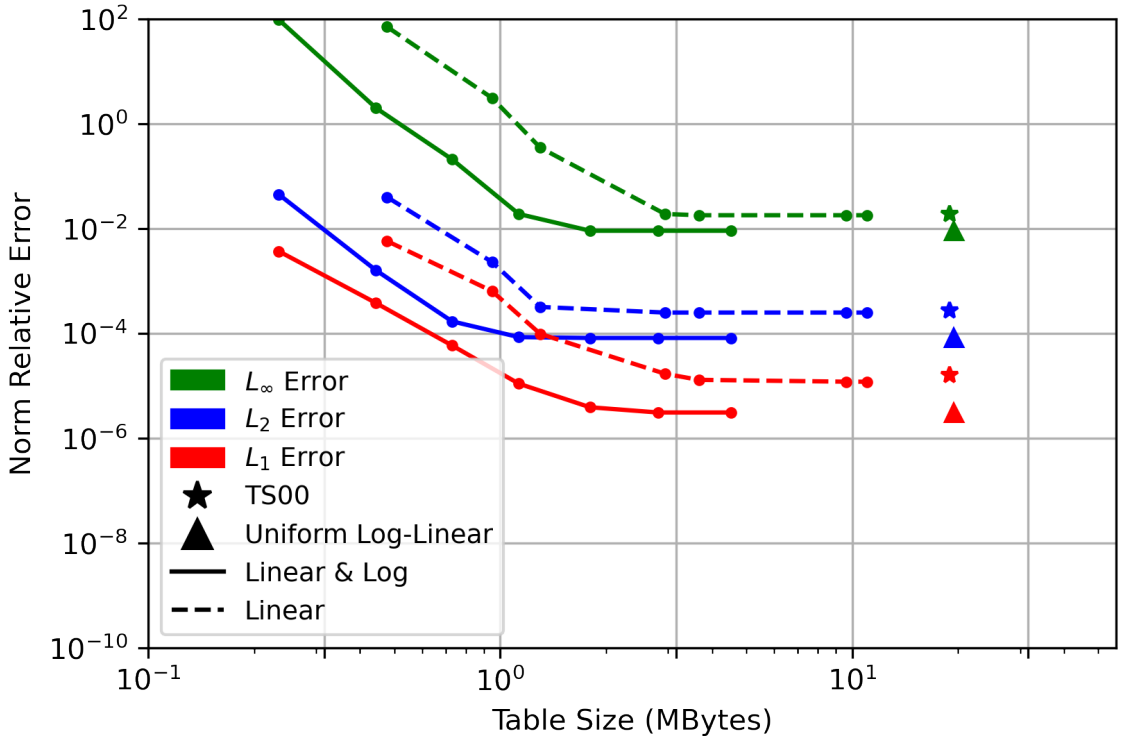


Figure 3.1: The error and memory cost for different emulators with $D_{\max}$=7. The dashed lines correspond to a quadtree emulator with just the linear-space model class and the solid lines correspond to a quadtree emulator using both linear-space and log-space model classes. For each type of error, the points going left to right correspond to the following ε_{th} of $[10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}]$. The stars correspond to TS00 and the triangles to TS00 but with log-space and linear-space model classes being used.

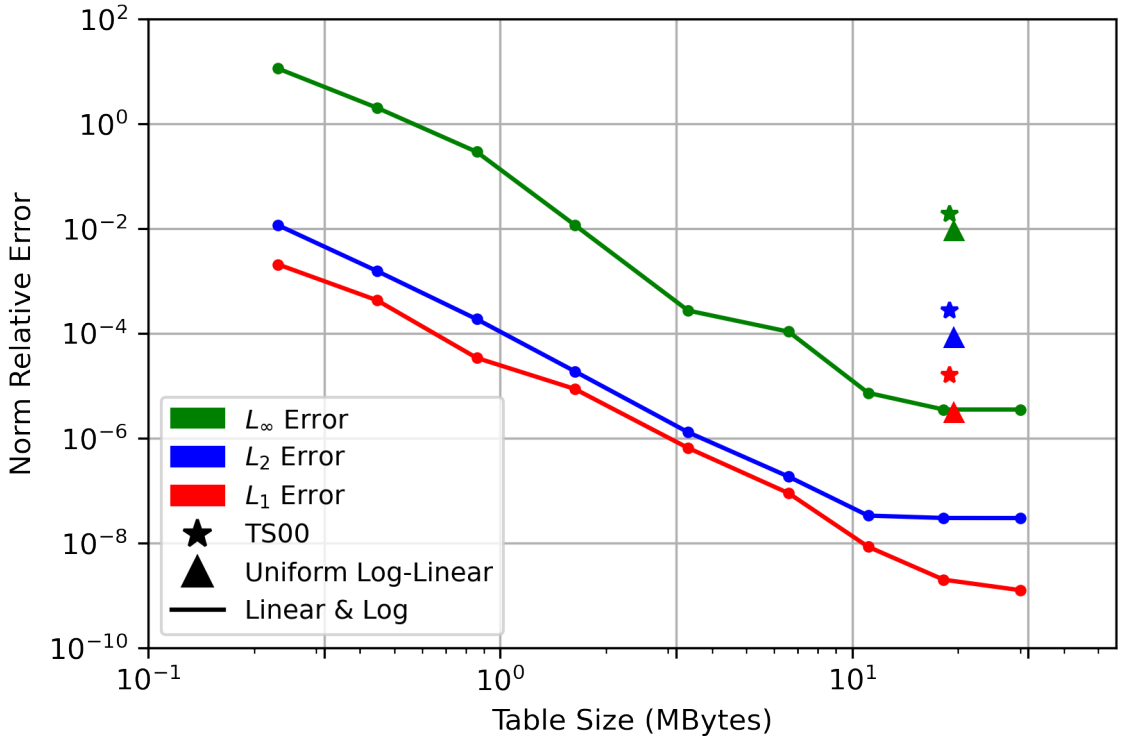


Figure 3.2: The error and memory cost for different emulators with $D_{\max}$=9. The dashed lines correspond to a quadtree emulator with just the linear-space model class and the solid to a quadtree emulator using both linear-space and log-space model classes. For each type of error, the points going left to right correspond to an ε_{th} of [10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}, 10^{-9}]. The stars correspond to TS00 and the triangles to TS00 but with log-space and linear-space model classes being used.

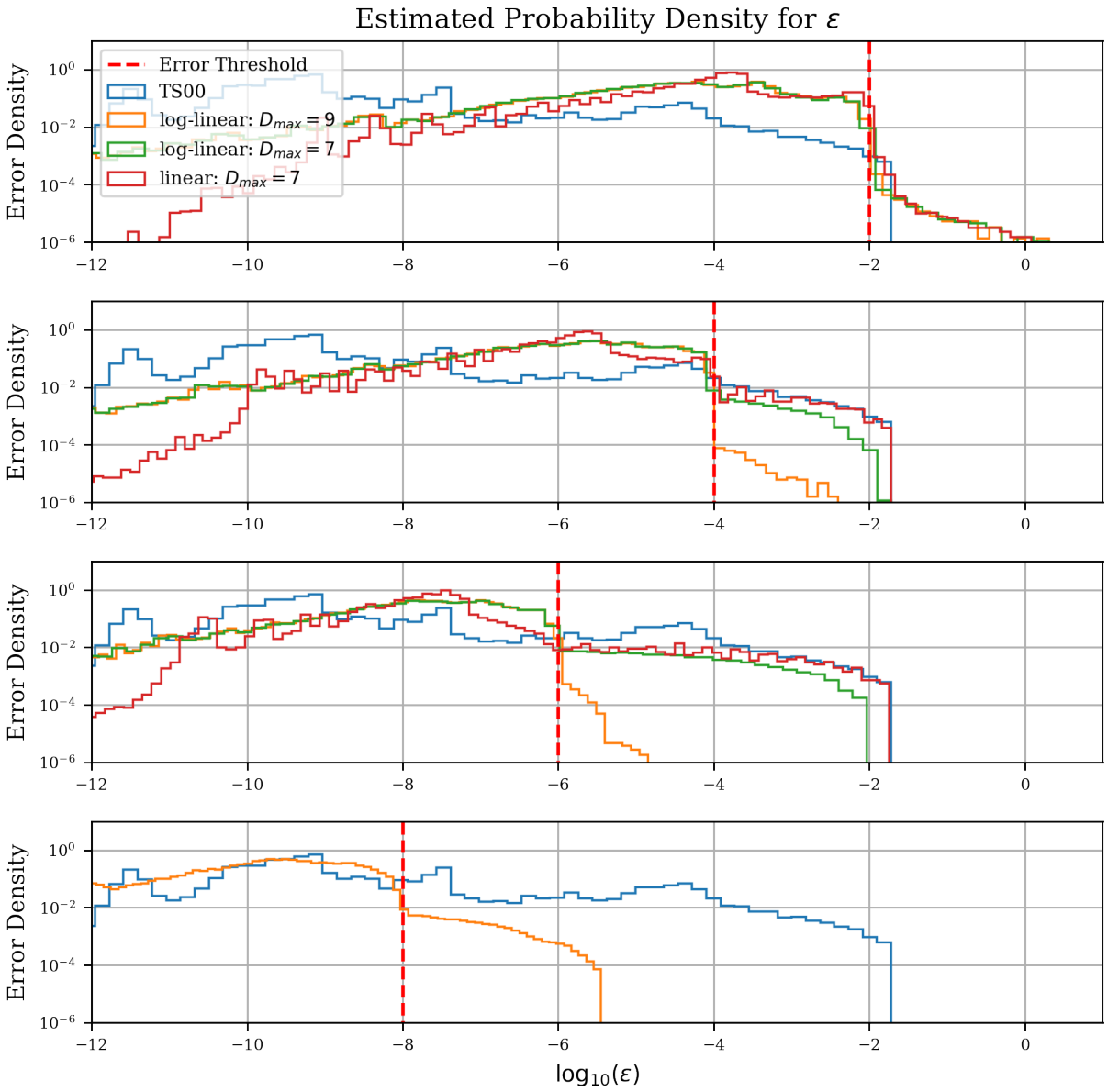


Figure 3.3: An estimate of the $\log_{10}(\varepsilon)$ density over the domain for four different cases: TS00, linear-space model class with $D_{\max}$=7, and log-space and linear-space model classes with a $D_{\max}$ of 7 and 9.

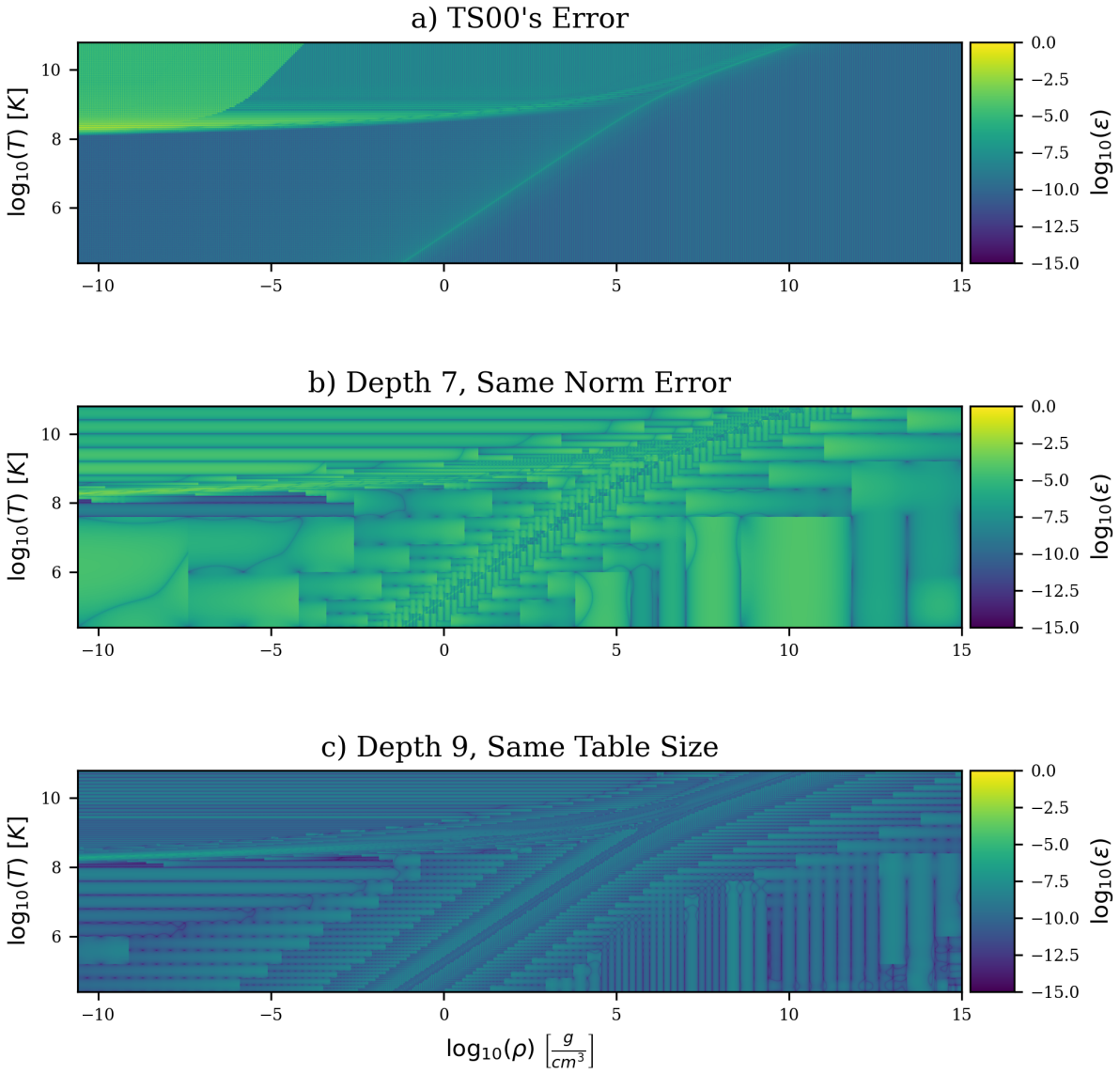


Figure 3.4: Spatial relative error for three different cases: (a) TS00's error, (b) Quadtree emulator that has the same norm error as TS00 ($D_{\max}$=7, ε_{th}=10^{-4}), (c) Quadtree emulator that has the same size as TS00 ($D_{\max}$=9, ε_{th}=10^{-8}).

CHAPTER 4

DISCUSSION

By increasing the complexity of the domain decomposition over the emulator's domain we can avoid excessive refinement in regions where sparse refinement produces acceptable errors as compared to the "truth" – i.e., the calculation that generates the tabular data. Similarly, we can increase the refinement in under-resolved regions that have relatively high error. This allows for the desired accuracy to be achieved throughout the domain without the excess memory requirements needed by a uniformly refined data table. Since the needed refinement of f varies drastically over the domain, we achieve substantial memory reductions from the increased domain decomposition complexity – a factor of about $20\times$ at a similar level of error to TS00.

The cost of increasing the complexity of the domain decomposition is an additional $\mathscr{O}(\log_2(m))$ search that must be performed for each input. Because the size of the array being searched is relatively small, and inputs close together in density vs temperature space are typically close together in memory, this operation will likely not suffer from excess memory movement from DRAM but will benefit greatly from cache reuse. Thus we predict the increased computational cost to be negligible compared to the memory savings.

An added benefit to the quadtree decomposition producing cells that are closer to their maximum size, while still being within an error tolerance, is that it effectively increases the arithmetic intensity by reducing the number of cells that need to be loaded for a given set of input values. Since simulations using tabular data are generally memory-bound, this should have the effect of increasing its overall performance. Examining the extent of this improvement and how it should influence the optimal size of cells is a topic of future research.

We have shown that allowing different model classes to be used throughout the table has increased the accuracy of the table while also decreasing memory cost. Although considerations of thermodynamic consistency restrain the types of model classes that can be used, the quadtree emulator framework allows for an arbitrary number of different model classes. Our future research

will investigate a variety of different model classes that may perform well in different regions of the domain.

This paper has focused on emulating a 2D domain using a quadtree. However, extending our framework to higher dimensions simply requires using a higher-order tree, e.g., an octree for 3D. The memory savings, compared to a uniform grid, in higher dimensions has the possibility of being even more significant. As a best-case scenario in terms of memory savings, assume a nD domain defined by a hypercube with corners $\vec{0}$ and $\vec{1}$, with an $(n-1)$D hyperplane discontinuity intersecting the $\vec{0}$ corner. Furthermore, assume that to resolve the discontinuity we need to have each cell that contains the discontinuity a certain size and that all other cells can be as big as possible. For the uniform case, this would require all cells to be small, i.e, 2^{ND} cells, where D is the dimension and N is the maximum depth of the tree needed to reach the smallest cell size. Using a tree decomposition, we find the cell count by recursively adding $2^D - 1$ for every refinement except for the last one, in which case we add 2^D. Thus the total cell count is $N(2^D - 1) + 1$ cells. A 2D example is shown in Figure 4.1(a). In this case, increasing the dimension of the problem further increases the memory savings, as shown in Figure 4.1(c).

For a prototypical case, assume the same hypercube is bisected by a $(n-1)$D hyperplane representing a discontinuity in the data values, which intersects the hypercube at half of its corners, and has similar cell resolution constraints as before. A 2D example is shown in Figure 4.1(b). In this case, we compute the number of cells recursively. Doing a single refinement on this hypercube using a 2^D-tree, we have 2^D cells, with 2^{D-1} being bisected by the hyperplane and 2^{D-1} having their corners intersected by the hyperplane. We have already done the case where a hyperplane intersects the corner of a hypercube, in which case $N(2^D - 1) + 1$ cells are needed. We have 2^{D-1} of these, each of which will have $N - 1$ more refinements. Therefore, we have $2^{D-1}((N-1)(2^D - 1) + 1)$ new cells. The 2^{D-1} that do not have their corner intersected but are bisected by the discontinuity can be treated recursively. Each refinement we decrease N by 1 and increasing the number of cells

that have their corners intersected by a multiplicative factor of 2^{D-1}. Thus we have

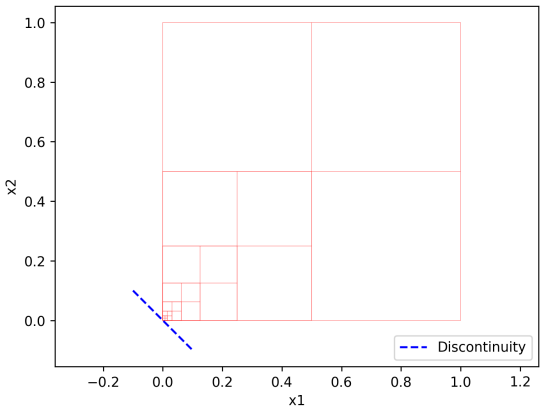
$$\sum_{i=1}^{N} 2^{i(D-1)}((N-i)(2^D-1)+1)$$

cells. On the last refinement, $i = N$, we need to add in the bisected cells, which is $2^{N(D-1)}$. All together we require a total of

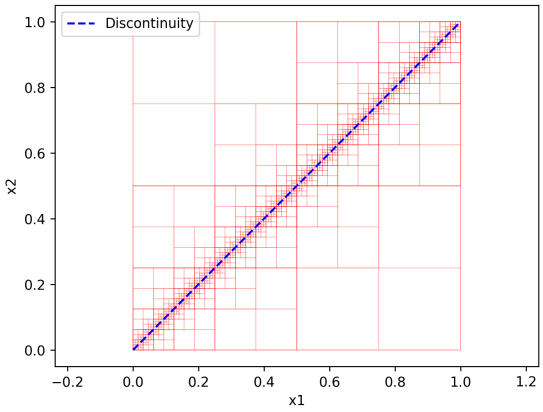
$$N_{\mathrm{c}} = \sum_{i=1}^{N} \left[2^{i(D-1)}((N-i)(2^D-1)+1)\right] + 2^{N(D-1)}$$

cells in the quadtree to refine the discontinuity, in comparison with the $N_{\mathrm{u}} = 2^{\mathrm{ND}}$ cells required to uniformly resolve the D-dimensional space with 2^{N} cells per edge, for a reduction in memory by a factor of N_c/N_u. We find that in this case, the memory savings as we increase the dimensions would be similar to the quadtree, as shown in Figure 4.1(c). This figure also includes the two quadtree emulators presented in Section 3 (D_{max}=7, $\varepsilon_{\mathrm{th}}$=$10^{-4}$ and D_{max}=9, $\varepsilon_{\mathrm{th}}$=$10^{-7}$), which are chosen at the points where the L_2 error has stopped decreasing. The behavior of the emulator for these cases is roughly in line with the prototypical case, which is consistent with the fact that the data tables have clearly visible discontinuities (see, e.g., Figure 2.1(a)). Data tables with multiple discontinuities would be expected to have somewhat higher memory consumption, but still have significant memory savings using the quadtree emulator compared to uniformly-spaced tabular data. We predict that the memory savings that come from using this framework to emulate higher dimensional data tables will typically result in progressively more significant savings in terms of memory consumption, with the same caveat regarding data discontinuities as in the 2D examples shown in this work.

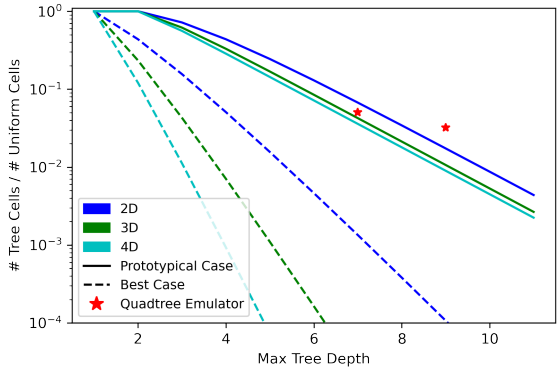
We have presented a proof-of-concept result that highlights the memory savings that can be gleaned from using tree-based methods to emulate tabular data. There is still room for further optimization to reduce memory costs and optimize the speed with which data can be retrieved from the emulator. In practice, modern multi- and many-core CPU and GPU architectures make a variety of tradeoffs between memory bandwidth, cache size, cache structure, cache bandwidth, and per-core arithmetic capabilities and speed. These details need to be considered when optimizing a tree-based emulator. Further optimizations could take into account the structure of the emulated data, as well as the simulation data itself (e.g., pre-loading branches of the emulator's tree into cache based on



(a)



(b)



(c)

Figure 4.1: An example of a quadtree refined region with a discontinuity crossing the (a) corner and (b) diagonal. (c) the compression ratio between using a tree based grid refinement method vs a uniform refinement for the cases shown in (a) and (b) but in varying dimensions. The red stars in (c)) show the compression of the $D_{\max}$=7, ε_{th}=10^{-4} and $D_{\max}$=9, ε_{th}=10^{-7}. The ε_{th} is chosen for each $D_{\max}$ such that the L_2 error has mostly stopped decreasing.

local simulation quantities in order to save lookup time). We expect that the increased flexibility provided by the tree-based method presented in this paper will ultimately allow for the utilization of all of these factors in lowering both the overall runtime and memory consumption of simulations utilizing tabular data during runtime. We plan to explore this in future work.

In the present work, we focus on the application of our quadtree emulator to the recovery of the equation of state of stellar plasmas from tabulated data, but our approach can be generalized to other table-based recovery schemes. Potential applications include other tabular equations of state, recovery of complex opacity data for radiative transfer calculations, or materials properties.

CHAPTER 5

CONCLUSIONS

We have presented a scheme for tabular data that uses a quadtree decomposition to reduce memory costs and keep the global accuracy consistent throughout the domain. For our proxy problem, the electron-positron Helmholtz free energy equation of state, we have achieved a memory reduction of 20x without loss of accuracy or four orders of magnitude increase in accuracy for the same size table. Although outlined in 2D, this method is easily modified for higher-dimensional problems and has the flexibility of using many types of interpolation schemes in coordination.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. S. Jermyn, J. Schwab, E. Bauer, F. X. Timmes, A. Y. Potekhin, Skye: A differentiable equation of state, arXiv:2104.00691 [astro-ph] (2021). arXiv:2104.00691.
URL http://arxiv.org/abs/2104.00691

[2] R. A. Finkel, J. L. Bentley, Quad trees a data structure for retrieval on composite keys, Acta Informatica 4 (1) (1974) 1–9. doi:10.1007/BF00288933.
URL https://doi.org/10.1007/BF00288933

[3] T. Markas, J. Reif, Quad tree structures for image compression applications, Information Processing & Management 28 (6) (1992) 707–721, publisher: Pergamon. doi:10.1016/0306-4573(92)90063-6.
URL http://www.sciencedirect.com/science/article/pii/0306457392900636

[4] A. Gersho, R. M. Gray, Vector quantization and signal compression, Vol. 159, Springer Science & Business Media, 2012.

[5] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, H. Tufo, FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes, The Astrophysical Journal Supplement Series 131 (1) (2000) 273, publisher: IOP Publishing. doi:10.1086/317361.
URL http://iopscience.iop.org/article/10.1086/317361/meta

[6] Z. He, C. Wu, G. Liu, Z. Zheng, Y. Tian, Decomposition tree: a spatio-temporal indexing method for movement big data, Cluster Computing 18 (4) (2015) 1481–1492. doi:10.1007/s10586-015-0475-3.
URL https://doi.org/10.1007/s10586-015-0475-3

[7] F. D. Swesty, Thermodynamically consistent interpolation for equation of state tables, Journal of Computational Physics 127 (1) (1996) 118–127. doi:10.1006/jcph.1996.0162.
URL http://www.sciencedirect.com/science/article/pii/S002199919690162X

[8] F. X. Timmes, F. D. Swesty, The Accuracy, Consistency, and Speed of an Electron-Positron Equation of State Based on Table Interpolation of the Helmholtz Free Energy, The Astrophysical Journal Supplement Series 126 (2000) 501–516. doi:10.1086/313304.
URL http://adsabs.harvard.edu/abs/2000ApJS..126..501T

[9] M. J. Cawkwell, M. Zecevic, D. J. Luscher, K. J. Ramos, Complete equations of state for cyclotetramethylene tetranitramine, Propellants, Explosives, Pyrotechnics (2021). doi:https://doi.org/10.1002/prep.202000274.
URL http://onlinelibrary.wiley.com/doi/abs/10.1002/prep.202000274

[10] J. D. Coe, S. P. Rudin, B. Maiorov, Multiphase equation of state and thermoelastic data for polycrystalline beryllium, AIP Conference Proceedings 2272 (1) (2020) 070009, publisher: American Institute of Physics. doi:10.1063/12.0000902.
URL http://aip.scitation.org/doi/abs/10.1063/12.0000902

[11] A. L. Kuhl, J. B. Bell, D. Grote, Diffusion effects near discontinuities in explosions, AIP Conference Proceedings 2272 (1) (2020) 070024, publisher: American Institute of Physics. doi:10.1063/12.0001094.
URL http://aip.scitation.org/doi/abs/10.1063/12.0001094

[12] M. Oertel, M. Hempel, T. Klähn, S. Typel, Equations of state for supernovae and compact stars, Reviews of Modern Physics 89 (2017) 015007. doi:10.1103/RevModPhys.89.015007.
URL http://adsabs.harvard.edu/abs/2017RvMP...89a5007O

[13] D. Pochik, B. L. Barker, E. Endeve, J. Buffaloe, S. J. Dunham, N. Roberts, A. Mezzacappa, thornado-hydro: A discontinuous galerkin method for supernova hydrodynamics with nuclear equations of state, The Astrophysical Journal Supplement Series 253 (1) (2021) 21. doi:10.3847/1538-4365/abd700.
URL https://doi.org/10.3847/1538-4365/abd700

[14] B. Paxton, L. Bildsten, A. Dotter, F. Herwig, P. Lesaffre, F. Timmes, Modules for experiments in stellar astrophysics (MESA), The Astrophysical Journal Supplement Series 192 (2011) 3. doi:10.1088/0067-0049/192/1/3.
URL http://adsabs.harvard.edu/abs/2011ApJS..192....3P

[15] H. Shen, H. Toki, K. Oyamatsu, K. Sumiyoshi, Relativistic equation of state for core-collapse supernova simulations, The Astrophysical Journal Supplement Series 197 (2011) 20. doi:10.1088/0067-0049/197/2/20.
URL http://adsabs.harvard.edu/abs/2011ApJS..197...20S

[16] H. Shen, F. Ji, J. Hu, K. Sumiyoshi, Effects of symmetry energy on the equation of state for simulations of core-collapse supernovae and neutron-star mergers, The Astrophysical Journal 891 (2) (2020) 148, publisher: American Astronomical Society. doi:10.3847/1538-4357/ab72fd.
URL https://doi.org/10.3847/1538-4357/ab72fd

[17] S. P. Lyon, Sesame: the los alamos national laboratory equation of state database, Los Alamos National Laboratory report LA-UR-92-3407 (1992).

[18] A. W. Steiner, M. Hempel, T. Fischer, CORE-COLLAPSE SUPERNOVA EQUATIONS OF STATE BASED ON NEUTRON STAR OBSERVATIONS, The Astrophysical Journal 774 (1) (2013) 17, publisher: IOP Publishing. doi:10.1088/0004-637X/774/1/17.
URL http://iopscience.iop.org/article/10.1088/0004-637X/774/1/17/meta

[19] S. W. Bruenn, J. M. Blondin, W. R. Hix, E. J. Lentz, O. E. B. Messer, A. Mezzacappa, E. Endeve, J. A. Harris, P. Marronetti, R. D. Budiardja, M. A. Chertkow, C.-T. Lee, Chimera: A massively parallel code for core-collapse supernova simulations, The Astrophysical Journal Supplement Series 248 (1) (2020) 11, publisher: IOP Publishing. doi:10.3847/1538-4365/ab7aff.
URL http://iopscience.iop.org/article/10.3847/1538-4365/ab7aff/meta

[20] F. X. Timmes, D. Arnett, The Accuracy, Consistency, and Speed of Five Equations of State for Stellar Hydrodynamics, The Astrophysical Journal Supplement Series 125 (1999) 277–294.

doi:10.1086/313271.
URL http://adsabs.harvard.edu/abs/1999ApJS..125..277T

[21] M. Baer, findiff, https://github.com/maroba/findiff (2021).

[22] G. M. Morton, A computer oriented geodetic data base and a new technique in file sequencing (1966).

[23] N. Mamoulis, Spatial data management, Synthesis Lectures on Data Management 3 (6) (2011) 1–149.