It's Tensor Time!: A Computational Framework for Analyzing Structured Matrices

M.T. Scott

September 12, 2022

Abstract

Matrices show up in many different scientific applications, like discretized PDEs in fluid dynamics and financial mathematics. However, these matrices are dense and require substantial memory to store them, as well as computing power to manipulate and solve systems using them. However, if these matrices have some underlying structure, either by individual elements or on a block level, we can use specialized techniques to reduce memory, computing time, or (hopefully) a combination of both. One such method is to map this structured matrix into a 3+ order tensor, which can uncover structure that isn't seen on a lower order matrix. We then decompose the tensor, and map back into a matrix approximation. Variations of this framework are used to analyze error, storage, and time complexity.

1 Motivation

Matrices are everywhere. They can be used in strictly algebraic settings- matrix groups, linear algebra, etc- but they also show up in differential equations. For example, we have coupled Ordinary Differential Equations (ODEs)

$$\frac{\mathrm{d}x}{\mathrm{d}t} = ax + by \tag{1}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = cx + dy \tag{2}$$

that can be transformed into

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{\mathrm{d}x}{\mathrm{d}t} \\ \frac{\mathrm{d}y}{\mathrm{d}t} \end{bmatrix} \tag{3}$$

Another way matrices show up is through discretizing differential equations. Basically, this is used to solve the continuous ODE at discrete places. For example, to discretly solve $f''(x) = 0$, we could use the **second-order central difference approximation** to get.

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \tag{4}$$

This means that we can take meshpoints x_i, and get the approximate solution at those points, via

$$\frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{5}$$

If we want the error to be small, we would take $n \to \infty$. If we wanted to solve this using a direct solver, we would need n^3 operations, and n^2 pieces of memory. That's bad! We see just from looking at this matrix, that there is some structure, so using this structure we can speed up matrix-vector products (mat-vecs).

Main take away: We want to use matrices to model the world, but storage of a matrix $\mathbf{M} \in \mathbb{R}^{n\times n}$ takes $\mathcal{O}(n^2)$ units of memory and to solve them we need to do $\mathcal{O}(n^3)$ `flops`, or floating-point operations (+,-,×, / done on a computer), and assuming n is large, this is not feasible. We are trying to find a way that we get the same (or very good approximation to the) answer that takes less memory and less computing time. The way I (and my collaborators) decided to do this is exploit the structure, called "kronecker structure" of these matrices that arise from these problems. This project does that through the following schema:

$$\begin{array}{ccc} \mathbf{A} & \xrightarrow{mat2ten} & \mathcal{T} \\ \approx\uparrow & & \downarrow\texttt{tr-HOSVD} \\ \hat{A} & \xleftarrow[ten2mat]{} & \hat{\mathcal{T}} \end{array}$$

We take a matrix that is generated from some technique (in our case we are looking at random matrix kernels, regular mesh discretations of a fractional partial differential equation, as well as an adaptive discretation of an FDE), and we use a bijective mapping from block matrices to become lateral slices of a third order tensor. Then we use the Tucker, or truncated-Higher Order Singular Value Decomposition, to take this newly formed tensor and break it into factor matrices, which we can use to represent the original matrix as a sum of kronecker products. This technique may look weird but by going to a higher otder, we uncover this structure, and can use it for faster computations with less storage and the error in the approximation is the same as the error between the original matrix and the matrix approximation. All benefits, and the error is not being magnified!

Okay but what is Kronecker structure? What is a tensor? And why do we have to go up to higher order dimensions?

2 Introduction

2.1 Definitions and Notation

A scalar will be denoted with a lower case letter $a \in \mathbb{R}$, an n-dimensional vector will be denoted with an arrow over top $\vec{v} \in \mathbb{R}^n$, an $m \times n$ matrix will be denoted as a bold capital letter $\mathbf{M} \in \mathbb{R}^{m \times n}$, and lastly any d-way tensor (where $d \geq 3$) will be denoted as a capital caligraphic letter, $\mathcal{T} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$. A matrix transpose will be denoted as $\mathbf{M}^T$ or $\mathbf{M}'$. These both mean the element in the i^{th} row and thej^{th} column, namely a_{ij} is now in the a_{ji} position.

Definition 2.1 (Matrix)**.** A matrix $\mathbf{M}$ is a rectangular array of numbers. We say that $\mathbf{M} \in \mathbb{F}^{m \times n}$, if $\mathbf{M}$ has m columns and n rows, and each of the m_{ij}, or the element of $\mathbf{M}$ in the i^{th} row and the j^{th} column, all belong to some field $\mathbb{F}$. In this research, we exclusively use $\mathbb{F} = \mathbb{R}$.

Definition 2.2 (Tensor)**.** We have already seen a "tensor" before. In fact, a scalar c, is a 0-way, or 0^{th} order tensor, a vector $\vec{v}$ is a 1-way, or 1^{st} order tensor, a matrix, $\mathbf{M}$ is just a 2-way or 2^{nd} order tensor. Normally, we use the word tensor to mean a 3+ order tensor. With that definition let $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ be a d-way tensor, with elements $a_{i_1, i_2, \cdots, i_d}$.

Definition 2.3 (Kronecker Product)**.** Let $\mathbf{A} \in \mathbb{R}^{m \times p}, \mathbf{B} \in \mathbb{R}^{n \times l}$. Then the Kronecker Product $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{(mn) \times (pl)}$ is denoted as

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1p}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2p}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mp}\mathbf{B} \end{pmatrix} \tag{6}$$

Example 2.1. Consider two vectors $\vec{a}, \vec{b} \in \mathbb{R}^n$, or maybe it would be helpful to write them as $\vec{a}, \vec{b} \in \mathbb{R}^{n \times 1}$. We could consider the following as an outer product,

or as a kronecker product.

$$\vec{a}\vec{b}^T = \sum_{i=1}^{n} a_i b_i \tag{7}$$

$$= \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix} \tag{8}$$

$$= \begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_1 & a_n b_2 & \cdots & a_n b_n \end{pmatrix} \tag{9}$$

$$= \begin{pmatrix} a_1\vec{b} \\ a_2\vec{b} \\ \vdots \\ a_n\vec{b} \end{pmatrix} \tag{10}$$

$$= \vec{a} \otimes \vec{b} \tag{11}$$

Remark 2.1. Maybe we don't want to deal with matrices, so we just make a new vector $\vec{c}^T := (\vec{a}^T, \vec{b}^T)$. Then we save memory as we only have to store $2n$ units of memory as opposed to n^2, but we lost the inherent structure of the matrix. This isn't good. Also kronecker products have some super cool properties, so we don't have to implicitly form the kronecker product matrix. As a result, with the kronecker product, we see that $mp+nl$ units of storage, we can generate an $(mn) \times (pl)$ matrix, which normally takes $mnlp$ units of storage to naively store. This is advantagous assuming that $mp+nl < mpnl$, as it saves us memory.

Another type of structure that shows up in our work is matrices that are Toeplitz on an entry and/or block level. What does that mean?

Definition 2.4 (Toeplitz Matrix)**.** This is also known as "constant diagonal" matrix. Let $\mathbf{A} \in \mathbb{R}^{m \times n}$, then $\mathbf{A}$ is called Toeplitz if it meets the following criteria:

$$\mathbf{A} = \begin{pmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & a_{-(n-2)} \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{m-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{pmatrix} \tag{12}$$

While finding structure in the elements is great, we also want to find structure on more of a block level, where "blocks" are submatrices of the overall matrix.

Definition 2.5 (Block Toeplitz)**.** The definition of the "Block Toeplitz" matrix is the exact same except we are replacing scalars with matrices themselves. Let $\mathbf{A} \in \mathbb{R}^{mp \times nl}$, with $m \times n$ blocks, each of these blocks $\mathbf{A}_i$ is of the size $p \times l$, then $\mathbf{A}$ is called a Block Toeplitz matrix if it meets the following criteria:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_0 & \mathbf{A}_{-1} & \mathbf{A}_{-2} & \cdots & \mathbf{A}_{-(n-1)} \\ \mathbf{A}_1 & \mathbf{A}_0 & \mathbf{A}_{-1} & \ddots & \mathbf{A}_{-(n-2)} \\ \mathbf{A}_2 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \mathbf{A}_1 & \mathbf{A}_0 & \mathbf{A}_{-1} \\ \mathbf{A}_{m-1} & \cdots & \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{A}_0 \end{pmatrix} \tag{13}$$

Sometimes we want to access an entire row of a matrix, say the i^{th} row of a matrix $\mathbf{M}$, we will denote that as $M_{i:}$, where the : means all the elements. Similarly, if we wanted to talk about the j^{th} column, we would denote that as $M_{:j}$. This same notation is used for accessing elements of a tensor. But first what is a tensor?

Remark 2.2. Sometimes it's hard to visualize these higher order tensors, so that's why we will define parts of tensors called sliced and reorganize them to get to a matrix which is easy to visual.

Definition 2.6 (Slices of a Third order Tensor)**.** A slice of a 3D tensor is simply a matrix, where we hold one index fixed, and access all of the other elements in the other two dimensions.

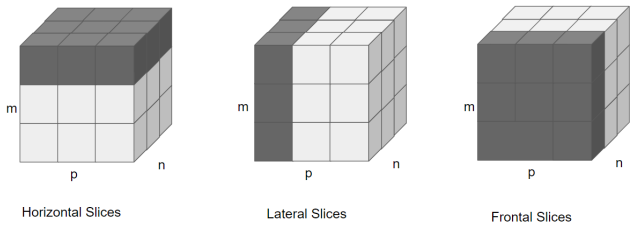

Figure 1: These are the three possible slices of a third order tensor, $\mathcal{A}$.

These slices are matrices of themselves. If we want to denote the i^{th} horizontal slice of a tensor $\mathcal{A}$, we'd write $\mathcal{A}_{i::}$, the j^{th} lateral slice, we'd write $\mathcal{A}_{:j:}$, and k^{th} frontal slice, we'd write $\mathcal{A}_{::k}$. The three slices (for the third order tensor we are dealing with) are visualized in figure 1.

Remark 2.3. There are also ways to access specific vectors in a tensor, which are called fibers. Also as we go to higher dimension tensors, there are different

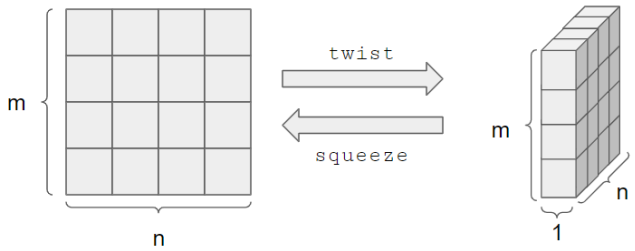

Figure 2: We can take a $m \times n$ matrix and "twist" it into a $m \times 1 \times n$ tensor. Similarly, we can take an $m \times 1 \times n$ tensor and "squeeze" out a dimension to get back to an $m \times n$ matrix.

ways to access the information you want, but the principle generalizes nicely. We will only talk about slices though as they are the matrices to do matric-matrix products.

2.2 Turning Matrices into Tensors and Back Again

The obvious question is why do we do this? Tensors are harder to view and visualize, tensor problems are NP-hard, tensors aren't nicely supported in scientific computing softwares like MATLAB. What is the point? Well all of these are true, but tensors have nice properties. Firstly, most tensor factorizations are unique, or they have some structure which is helpful in computations. Also, there are software packages like Tensor Lab and Tensor Toolbox that are supplimental to softwares. Last and most importantly to this project, there might be some structure that we don't see by having the data in a 2D matrix. Converting it to a higher order tensor might uncover that structure, then we can use specialized techniques and properties of this structure to make computations faster and more accurate. One such type of structure is Toeplitz and Block Toeplitz. Another is the Kronecker structure that we talked about.

Once we have these blocks of structure, we can take them and convert them into lateral slice of a tensor. This process is bijective. We do this by performing a bijective function on these sub blocks. To turn a matrix into a tensor, we "twist" it into a higher dimension, and to turn a tensor into a matrix, "squeeze" out a dimension. This bijective action is visualized in figure 2. So we take these blocks, twist them into lateral slices and then concatenate these lateral slices to form a tensor as seen in figure 3.

2.3 Tensor Decompositions

Once we make a tensor framework that has hopefully generated this underlying structure, it is time to find it and illuminate it. This is done through a tensor decomposition. While there are many different kinds, we are only go-

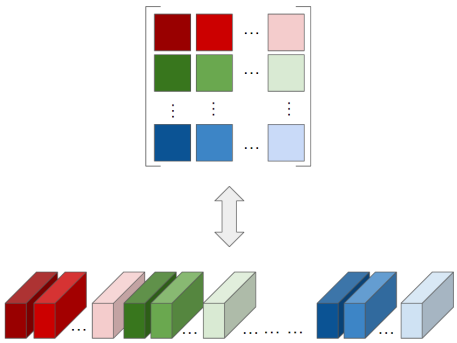

Figure 3: We take these structured submatrices and make them lateral slices of a tensor. Of course we can take the lateral slices of a tensor and turn them back into a matrix.

ing to talk about the `tr-HOSVD`, or the truncated-Higher Order Singular Value Decomposition. First let's talk about what is the "normal" SVD.

Definition 2.7 (Singular Value Decomposition)**.** Let $\mathbf{M} \in \mathbb{R}^{m\times n}$ be a rank-r matrix. Then the singular value decomposition (SVD) is $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, where $\mathbf{U} \in \mathbb{R}^{m\times m}, \mathbf{V}^T \in \mathbb{R}^{n\times n}$ are both real, orthogonal matrices, namely $\mathbf{U}^T\mathbf{U} = \mathbf{I}_m, \mathbf{V}^T\mathbf{V} = \mathbf{I}_n$. These are known as the left and right singular vectors, respectively. Lastly, $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ is rectangular diagonal matrix with entries $\sigma_1 \geq \sigma_2 \geq \cdots \sigma_r > 0$. These are called the singular values of $\mathbf{M}$. Since $\mathbf{M}$ is real, then the SVD exist!

This is an expensive algorithm and is very costly, so we wouldn't want to actually carry out the entire algorithm. Instead, we just use the economy SVD.

Definition 2.8 ("Economy" SVD)**.** Let $k \leq \min\{m, n\}$. Then the "economy", or "thin", SVD of a matrix $\mathbf{M} \in \mathbb{R}^{m\times n}$ is $\mathbf{M} \approx \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^T$, where $\mathbf{U}_k \in \mathbb{R}^{m\times k}, \mathbf{\Sigma} \in \mathbb{R}^{k\times k}, \mathbf{V}_k^T \in \mathbb{R}^{k\times n}$.

Remark 2.4. While we now have a way of talking about these slices, we can use that to reorder the structure and make tensors matrices by unfolding them, and matrices tensors by refolding them. We do that through Tensor unfoldings, sometimes called "matricization". We see the different ways of unfolding a third order tensor in figure 4.

Definition 2.9 (k^{th}-mode Tensor Unfolding)**.** Let $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$ be a d-way tensor. Then the k^{th}-mode unfolding is defined as

$$\mathbf{A}_{(k)} \in \mathbb{R}^{n_k\times n_1 n_2\cdots n_{k-1}n_{k+1}\cdots n_d} \tag{14}$$

Note that while the ordering ultimately doesn't matter as they will just be different permutations of the same tensor, for consistency, we have chosen that mode $n_1 > n_2 > \cdots > n_d$, where the " $>$ " is used to denote ordering and not actually larger in magnitude.

This abstract definition might be hard to visualize, so we have provided an example 2.2.

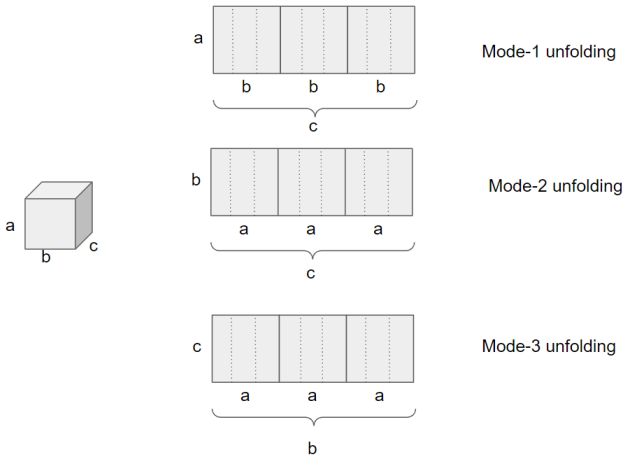

Figure 4: These are the three ways to turn a 3rd order tensor into matrices using the ordering we've chosen.

Example 2.2. Let's show what the mode three unfoldings could look like. Let $\mathcal{A} \in \mathbb{R}^{3\times4\times2}$, where the frontal slices of the tensor $\mathcal{A}_{::i}, i = 1, 2$ are:

$$\mathcal{A}_{::1} = \begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix} \tag{15}$$

$$\mathcal{A}_{::2} = \begin{pmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{pmatrix} \tag{16}$$

$$\tag{17}$$

Then the following are the k^{th}-mode unfoldings ("matricizations")

$$\mathbf{A}_{(1)} = \begin{pmatrix} 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 \end{pmatrix} \tag{18}$$

$$\in \mathbb{R}^{3\times(4)(2)} \tag{19}$$

$$\mathbf{A}_{(2)} = \begin{pmatrix} 1 & 2 & 3 & 13 & 14 & 15 \\ 4 & 5 & 6 & 16 & 17 & 18 \\ 7 & 8 & 9 & 19 & 20 & 21 \\ 10 & 11 & 12 & 22 & 23 & 24 \end{pmatrix} \tag{20}$$

$$\in \mathbb{R}^{4\times(3)(2)} \tag{21}$$

$$\mathbf{A}_{(3)} = \begin{pmatrix} 1 & 2 & 3 & \cdots & 10 & 11 & 12 \\ 13 & 14 & 15 & \cdots & 22 & 23 & 24 \end{pmatrix} \tag{22}$$

$$\in \mathbb{R}^{2\times(3)(4)} \tag{23}$$

Definition 2.10 (HOSVD)**.** Once we have unfolded the tensors all of the ways possible, we perform an SVD computation on all of the matricizations, keeping the left singular vectors in each case, denoted $\mathbf{U}_{(1)}, \mathbf{U}_{(2)}, \mathbf{U}_{(3)}$ for the first, second, and third matricization, respectively. Then the HOSVD is performed by computing the core tensor $\mathcal{G}$ by

$$\mathcal{G} := \mathbf{U}_{(3)}^T \mathbf{U}_{(2)}^T \mathbf{U}_{(1)}^T \mathcal{A} \tag{24}$$

Once we have the core tensor, we can truncated it in any mode possible, or we can keep it full rank, and then we "undo" the process to get a tensor approximation, namely

$$\hat{\mathcal{A}} \approx \mathbf{U}_{(3)} \mathbf{U}_{(2)} \mathbf{U}_{(1)} \mathcal{G} \tag{25}$$

Remark 2.5. Then we see from these factor matrices, we can squeeze them, reshape them, twist them, transpose them, etc, so that we get something of the form

$$\hat{\mathbf{A}} = \sum_{i=1}^{J} \mathbf{C}_i \otimes \mathbf{D}i \tag{26}$$

where J is the truncation rank of the `tr-HOSVD` algorithm, which is the upper summand of Kronecker products. This allows for nice properties so that we can use to speed up both the time of computation and save on memory storage as well.

Remark 2.6. The error between $\frac{\|\mathcal{A}-\hat{\mathcal{A}}\|_F}{\|\mathcal{A}\|_F} = \frac{\|\mathbf{M}-\hat{\mathbf{M}}\|_F}{\|\mathbf{M}\|_F}$ proven by Kilmer and Saibaba.

Definition 2.11 (Frobenius Norm)**.** Let $\mathbf{A}$ be an $m \times n$ matrix. The Frobenius norm of this matrix is the square root of the sums of the absolute squares of all

of the elements.

$$\|\mathbf{A}\|_F := \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} \tag{27}$$

Remark 2.7. Since we are summing over each entry exactly once, it doesn't matter what order we are summing the elements in. In fact, we could rearrange them and as long as we are only summing them once, we are good to go!

Definition 2.12 (Vecotrization)**.** Assume we have some d-way tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$. By vectorizing, or turning it into a vector, we see that by stacking the column vectors, we can get a vector, $\texttt{vec}(v) \in \mathbb{R}^{n_1 n_2 \cdots n_d}$.

Definition 2.13 (Absolute and Relative Error)**.** Let $\hat{\mathcal{A}}$ be an approximation to $\mathcal{A}$, where here $\mathcal{A}$ can be a tensor of any order. Then the absolute error, $\left\|\mathcal{A} - \hat{\mathcal{A}}\right\|$ is just the distance between the tensor and its approximation using whatever norm we are using. The relative error is the absolute error inversely scaled by the norm of the original tensor, namely $\frac{\left\|\mathcal{A}-\hat{\mathcal{A}}\right\|}{\|\mathcal{A}\|}$.

3 New Work

This is where I picked up the project. There was proof of concept and some code to show that it worked, but only tested computationally for a small matrix. But as computational mathematicians, just because it is true mathematically, doesn't mean that it is true computationally. The first step of what I did was try to combine codes of the fractional partial differential equation (FDE) that was generated from Prof. Hu's research and see if it was compatable with the code from Prof. Kilmer and Prof. Saibaba. Seeing that the code to generate the FDE on a regular mesh is Toeplitz-like, it was no suprise that the code worked well. In fact, it only took $J = 7$ to get to machine precision. Now that we have shown that the regular mesh FPDE works well with this computational framework, the next question is how does this process scale for larger matrices. For our process to work we are dependent on a matrix that is a square matrix of the form $2^L \times 2^L$.

However, this begs the question: as we scale the matrix, is it better to increase the size of the submatrix blocks and keep the number of blocks constant so the tensor stays relatively square, or is it better to keep the size of the submatrix blocks constant and increase the number of blocks, so the tensor looks more like a hotdog as there are more lateral slices? In fact, we strive to find the best way of partitioning the matrix into sublocks so that we tease out as much structure as possible. We do this by changing the size of the submatrices creating either more or less blocks, stored as lateral slices.

It turns out this depends on the kernel, or the problem at hand. For the kernel which was used in the proof of concept, name;y $\exp\{|X - Y|\}$, we see that the bigger blocks method is actually a better method for compressing the

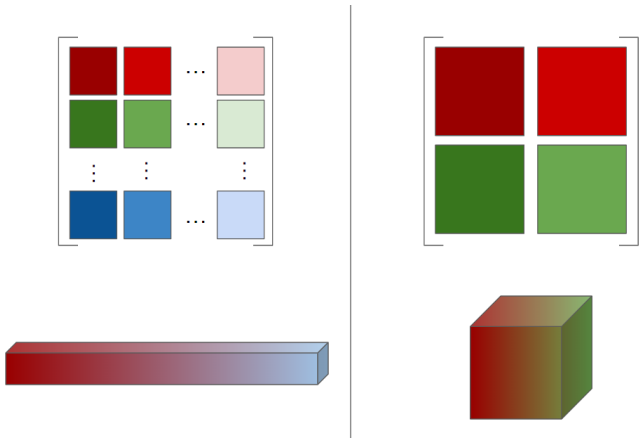

Figure 5: Left: As we partition the matrix into more subblocks all of the same size, there are more blocks which creates more lateral slice. This makes the tensor look like a hotdog. Right: If we keep a fixed number of blocks but let them scale up in size, then we end up with a bigger blocks which results in a hamburger shaped tensor.

information needed. However, this is not universal, as seen by the code for the FDE. This method actually works better with more blocks. However, we have a third category that works well with neither case which is an adaptive mesh code for the FDE. This is hard to investigate as adaptive code doesn't necessarily lend itself to being a square matrix of size 2^L. This means it takes a while of tuning these parameters to get one of that size to tensorize, factorize, and reconstruct. See future directions for more on this.

Mostly, I have spent time analyzing the storage complexity of this algorithm, which resulted in the following theorem.

Theorem 3.1 (tr-HOSVD approximation's Size Complexity)**.** *Let* $\mathbf{A} \in \mathbb{R}^{(mn)\times(mn)}$ *be a structured matrix such that it is block* $n \times n$ *with* $m \times m$ *sized blocks, and let* $\mathcal{A} \in \mathbb{R}^{m\times n^2\times m}$ *three way tensor that results from ordering each matrix block as a lateral slice. Then using the* `tr-HOSVD` *algorithm of truncation rank* (r_1, r_2, r_3) *which produces three matrices of left singular vecotrs (corresponding to* $\mathcal{A}$*'s three matricization modes), which are* $\mathbf{U}, \mathbf{V}, \mathbf{W}$*. We define* $\mathcal{C} := \mathbf{V}_{:\ell}^T$*. Similarly, define* $\mathcal{D} := \mathbf{U}\mathcal{G}_{:\ell:}\mathbf{W}^T$*. Then* $\mathcal{C} \in \mathbb{R}^{n\times n\times J}$ *and* $\mathcal{D} \in \mathbb{R}^{m\times m\times J}$*.*

Proof. We see from that $\mathbf{A}^{(1)} \in \mathbb{R}^{m\times n^2 m}$ elicits an orthogonal matrix of left singular vectors of $\mathbf{U} \in \mathbb{R}^{m\times m}$, as the economy SVD takes the left matrix to be square of size $\min\{d_1, d_2\}$, for the matrix of size $d_1 \times d_2$. Since $\mathcal{A}$ is $\{1, 3\}$ symmetric, then the economy SVD of $\mathbf{A}^{(3)}$, or third mode unfolding elicits $\mathbf{W} \in \mathbb{R}^{m\times m}$, as well. Upon truncating, we see $\mathbf{U} \in \mathbb{R}^{m\times r_1}, \mathbf{Q} \in \mathbb{R}^{m\times r_3}$.

However, $\mathbf{A}^{(2)} \in \mathbb{R}^{n^2\times m^2}$. Now assuming that we are in the "nice regime" $(n \leq m)$, or that the number of blocks are smaller than the size of the blocks, e.g. B8W64B, we get that $\mathbf{V} \in \mathbb{R}^{n^2}$, which can be reshaped into $\mathcal{C} \in \mathbb{R}^{n\times n\times J}$.

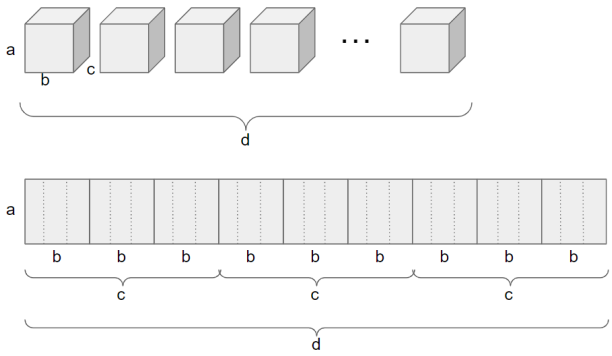

Figure 6: Visualizing three dimensions can be hard, but four dimensions is whoa! That is why I like to think of a fourth order tensor as a series of cubes. There are three dimensions to a cube, and the fourth dimension comes into counting which cube you are even looking at.

If we enter the bad regime, where $\mathbf{V} \in \mathbf{R}^{\min\{m,n\}^2} = \mathbb{R}^{m \times m}$, then we cannot performed $\mathbf{T}_2 := \mathcal{T}_1 \times_2 \mathbf{V}$, as the dimensions don't match. This could be averted if we put a check in the code to see if $n^2 \leq m^2$, and if not, then switch `svd(A,'econ')` for `svd(A)`. □

Currently, the code from the FDE is "Toeplitz-like" and performs really well with smaller blocks and more of them. However, this process at the moment is computationally infeasible. Since we have a $2^L \times 2^L$ matrix, as we decrease the block size to $2^b \times 2^b$, this means we have $2^{L-b} \times 2^{L-b}$ blocks which have to be twisted and concatenated into a tensor. This makes the time complexity $\mathcal{O}\big(2^{2(L-b)}\big)$, which is an NP issue, which makes this prohibitably expensive. Hotdog tensors take more time to deal produce than hamburger tensors, but for some problems they are the better options. So if we maybe change the order of the tensor, we could make this more feasible. An example of a fourth order tensor and one possible unfolding (mode 1) is seen in figure 6.

4 Future Directions

On top of twisting all subblocks into a fourth tensor, we are also looking at recursively subdividing a matrix and applying the Haar Wavelet transform (I am being intensionally vague about not discussing what this is) The first application leads to a fourth order tensor where the first and last indice tells you the row and column of the element in the submatrix, while the second and third indice tells you what is the row and column of the submatrix you are in. Then we apply

If you really care, the Haar Wavelet Transform is one of the heavy lifters of this project. It is an orthogonal matrix that has only two non-zero elements per row (or column, after all it is orthogonal) and when you use this kronecker product'ed to the oringinal kernel matrix, we get a really structured matrix that has a lot of nonzero elements.

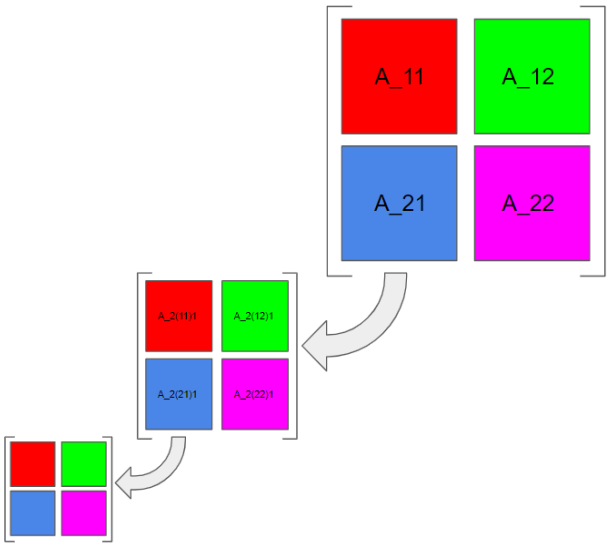

Figure 7

these Haar Wavelet transformation again, which produces four submatrices for each of the four original submatrices. The indices are as follow: the first and sixth index are the row and column of the submatrix, the second and fifth indice tells you which subblock you are in after the first Haar transform, and the third and fourth indices are the row and column of the submatrix you are in after the second Haar Transform. This is confusing, so I direct your attention to figure 7, where hopefully this makes more sense.

This is now a sixth order tensor, so good thing we have figured out how to write a code for any order tensor using the HOSVD algorithm.

5 Acknowledgements

First, I would like to acknowledge Prof. Misha Kilmer, Prof. Xiaozhe Hu, and Prof. Arvind Saibaba for laying the foundations of this project and supporting me as a burgeoning mathematician. This project is also partially supported by the National Science Foundation (NSF) grant # 1821148. Lastly, I want to acknowledge Taylor Swift for releasing amazing music that made TeX'ing this report easier.