SIAM J. SCI. COMPUT.
Vol. 42, No. 5, pp. A3516–A3539

TENSOR TRAIN CONSTRUCTION FROM TENSOR ACTIONS, WITH APPLICATION TO COMPRESSION OF LARGE HIGH ORDER DERIVATIVE TENSORS*

NICK ALGER†, PENG CHEN†, AND OMAR GHATTAS†

Abstract. We present a method for converting tensors into the tensor train format based on actions of the tensor as a vector-valued multilinear function. Existing methods for constructing tensor trains require access to "array entries" of the tensor and are therefore inefficient or computationally prohibitive if the tensor is accessible only through its action, especially for high order tensors. Our method permits efficient tensor train compression of large high order derivative tensors for nonlinear mappings that are implicitly defined through the solution of a system of equations. Array entries of these derivative tensors are not directly accessible, but actions of these tensors can be computed efficiently via a procedure that we discuss. Such tensors are often amenable to tensor train compression in theory, but until now no efficient algorithm existed to convert them into tensor train format. We demonstrate our method by compressing a Hilbert tensor of size $41 \times 42 \times 43 \times 44 \times 45$, and by forming high order (up to fifth order derivatives/sixth order tensors) Taylor series surrogates of the noise-whitened parameter-to-output map for a stochastic partial differential equation with boundary output.

Key words. tensor, tensor train, tensor action, randomized linear algebra, higher order derivative, uncertainty quantification

AMS subject classifications. 15A69, 35Q62, 35R30, 65C30, 65C60, 65F99, 68W20

DOI. 10.1137/20M131936X

1. Introduction.

To understand this paper, the reader should be familiar with tensors and tensor decompositions [22, 31], tensor trains [21, 39, 42], and randomized linear algebra [23]. Existing tensor train construction methods, including TT-SVD [39], TT-cross [40], Tucker-2/PVD [15], modified ALS [25], and other related methods [3, 16], view a dth order tensor as a multidimensional array,

$$T \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_d},$$

and construct a tensor train representation of T via processes that involve accessing many array entries of T. This is inefficient in applications where the tensor is only accessible through its *action* as a vector-valued multilinear function.

DEFINITION 1 (tensor action). *Let T be a dth order tensor. An action of T is a contraction of T with $d-1$ vectors.*

Tensor actions generalize the concept of the action of a matrix on a vector via matrix-vector multiplication. A matrix action takes one vector as input and returns

*Submitted to the journal's Methods and Algorithms for Scientific Computing section February 18, 2020; accepted for publication (in revised form) August 4, 2020; published electronically October 27, 2020.

https://doi.org/10.1137/20M131936X

Funding: This research was partially funded by the Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Mathematical Multifaceted Integrated Capability Centers (MMICCS) program under award DE-SC0019303, the Simons Foundation under award 560651, the Air Force Office of Scientific Research, Computational Mathematics program under award FA9550-17-1-0190, and the National Science Foundation, Division of Advanced Cyberinfrastructure under award ACI-1550593.

†Oden Institute for Computational Engineering and Sciences, The University of Texas as Austin, Austin, TX 78712 USA (nalger225@gmail.com, peng@ices.utexas.edu, omar@ices.utexas.edu).

one vector as output. A tensor action takes $d-1$ vectors as input and returns one vector as output.

To illustrate by example, suppose that $T \in \mathbb{R}^{N_1 \times N_2 \times N_3}$. Then an action of T is an evaluation of one of the following three functions:

$$A_1 : \mathbb{R}^{N_2} \times \mathbb{R}^{N_3} \to \mathbb{R}^{N_1},$$
$$A_2 : \mathbb{R}^{N_1} \times \mathbb{R}^{N_3} \to \mathbb{R}^{N_2},$$
$$A_3 : \mathbb{R}^{N_1} \times \mathbb{R}^{N_2} \to \mathbb{R}^{N_3},$$

which are defined by

$$A_1(v,w)_i := \sum_{j=1}^{N_2} \sum_{k=1}^{N_3} T_{i,j,k} v_j w_k, \quad i = 1, \ldots, N_1,$$
$$A_2(u,w)_j := \sum_{i=1}^{N_1} \sum_{k=1}^{N_3} T_{i,j,k} u_i w_k, \quad j = 1, \ldots, N_2,$$
$$A_3(u,v)_k := \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} T_{i,j,k} u_i v_j, \quad k = 1, \ldots, N_3.$$

For a dth order tensor, there are d functions A_i defined analogously. When we say that a tensor T is only accessible via its actions, we mean that we have algorithms or black-box computer codes that can evaluate the functions A_i on arbitrary input vectors, but we do not have any other information about the entries of T. Tensors that are only accessible through their action arise as tensors representing higher order derivatives of inverse and optimal control problem objective functions with respect to parameters. More generally, such tensors arise as tensor networks with a tree topology where at least one of the nodes in the network is a matrix inverse.

If a tensor is only available via actions, it is possible to compress it into tensor train format using existing algorithms such as TT-cross, but this process wastes information. Continuing the example where T is a 3-tensor, when the existing algorithm wishes to access tensor entry $T_{i,j,k}$, one computes the *fiber* $A_1(e_j, e_k) \in \mathbb{R}^{N_1}$ via a tensor action, then extracts $T_{i,j,k} = A_1(e_j, e_k)_i$ as the ith entry of the fiber. Here and throughout the paper, we write

$$e_m := \begin{bmatrix} 0 & \ldots & 0 & 1 & 0 & \ldots & 0 \end{bmatrix}^T$$

to denote a vector of the appropriate length (length N_2 and N_3 here) with mth entry equal to one and all other entries equal to zero. The fiber $A_1(e_j, e_k)$ can be stored and used later when other tensor entries in the fiber are needed (entries $T_{i,j,k}$ with the same i and j, but different k). However, existing tensor train compression methods access tensor entries in a *scattered* pattern: only a small number of entries in a given fiber are used. A large number of fibers must be computed, each of which requires a tensor action, and most of the entries in the computed fibers are ignored.

We present an efficient method to construct tensor train representations of tensors that are accessible only through their action. Our method (section 2) is based on a randomized algorithm, which has been used in [10, 26] for tensor train decomposition. While [10, 26] require the ability to perform array operations with the tensor, our method uses only the tensor action. Our key innovation is a method for implicitly computing the action of the remainder of a partially constructed tensor train, even

though we do not have access to the remainder directly. This allows us to use a "peeling process" to compute successive cores in the tensor train: we use a randomized range finder to compute the first core in the tensor train, then we use information from the first core and a randomized range finder to compute the second core, then we use information from the first and second cores and a randomized range finder to compute the third core, and so on. This "peeling process" continues until we have computed all cores in the tensor train. We were inspired by [33], in which randomized linear algebra and a peeling process are used to construct hierarchical matrices using only matrix-vector products. With our method, constructing the tensor train requires

$$O(\lceil r/N \rceil d r^2) \tag{1}$$

tensor actions, plus $O(dNr^2)$ memory for storage and $O(dNr^3)$ operations for linear algebra overhead. Here r is the tensor train rank (the maximum rank of the cores in the tensor train), $N = \max(N_1, \ldots, N_d)$, and $\lceil x \rceil$ is the smallest integer larger than x. The tensor actions are trivially parallelizable within each stage of the method. Once the tensor train has been constructed, the tensor may be manipulated efficiently using fast methods for tensor trains. In the case where $r < N$, we have $\lceil r/N \rceil = 1$, so the required number of tensor actions reduces to $O(dr^2)$. Tensor train compression is most effective when the tensor is large (large N and large d) and the tensor train rank is small (small r), so $r < N$ is a common use case. For tensors that arise in connection with integral or differential equations, N is typically the number of degrees of freedom in a discretization of a continuous function and is therefore on the order of thousands, millions, or more, while r may be on the order of ten to one hundred.

Our work was motivated by a desire to form high order (derivative order $k > 2$) Taylor series surrogate models of a quantity of interest $\mathcal{F} \in \mathbb{R}^{N_q}$ that depends on a parameter $m \in \mathbb{R}^{N_m}$ implicitly through the solution of a large system of nonlinear equations. Typically, computing a single entry of the quantity of interest, $(\mathcal{F}(m))_i$, requires essentially the same amount of work as computing the entire vector $\mathcal{F}(m)$, because one must solve the same system of nonlinear equations in either case. Analogously, computing an individual entry of the $(k+1)$th order tensor

$$\frac{d^k \mathcal{F}}{dm^k} \in \mathbb{R}^{N_m} \times \cdots \times \mathbb{R}^{N_m} \times \mathbb{R}^{N_q} \tag{2}$$

requires essentially the same work as computing the vector output of a tensor action. Tensor compression algorithms that operate by accessing scattered tensor entries (such as TT-cross) are therefore inefficient here. We show how to compute tensor actions for these higher order derivative tensors in section 3. These high order derivative tensors are often amenable to tensor train compression in principle, but until now no efficient algorithm existed for converting them into tensor train format. Quantized tensor train methods [37, 41, 38] can be highly effective for building surrogate models of the input to output map for the solution of a linear system [20, 28, 29, 30], but require access to tensor array entries, and therefore cannot be used to efficiently approximate higher order derivative tensors that arise when the system is nonlinear.

We demonstrate our method numerically in section 4.[1] First, we compress a Hilbert tensor of size $41 \times 42 \times 43 \times 44 \times 45$. We show that our method constructs nearly optimal tensor train approximations for this Hilbert tensor compared to TT-SVD and TT-cross. Second, we compress high order derivatives (up to fifth order

[1] Code for the methods and numerical results in this paper is available at https://github.com/NickAlger/TensorTrainHigherDerivatives.

derivatives or sixth order tensors) of the noise-whitened parameter-to-output map for a stochastic partial differential equation (PDE) with boundary output. We show that high order derivatives can be compressed into tensor train format with a low tensor train rank, and that the number of tensor actions needed to compress the high order derivative tensors is independent of the mesh used to discretize the problem. As the mesh is refined ($N_m \to \infty$), the required number of tensor actions remains roughly the same. We use these compressed derivative tensors to build Taylor series surrogate models for the noise-whitened parameter-to-output map, and find that including high order terms in the Taylor series yields more accurate surrogate models.

1.1. Isomorphism between arrays and multilinear functions. Tensor contraction establishes an isomorphism between multidimensional arrays and multilinear functions. Given a multilinear function,

$$T : \mathbb{R}^{N_1} \times \mathbb{R}^{N_2} \times \cdots \mathbb{R}^{N_d} \to \mathbb{R},$$

we may form an array representation of the function by applying the function to all possible combinations of standard unit basis vectors e_m. Given an array,

$$T \in \mathbb{R}^{N_1} \times \mathbb{R}^{N_2} \times \cdots \mathbb{R}^{N_d},$$

we may define a corresponding multilinear function that acts on vectors by contracting those vectors against the modes of the array. This associates each multilinear function with a unique array, and each array with a unique multilinear function. We use the word tensor to refer to both multilinear functions and multidimensional arrays. When a tensor is viewed as a multilinear function, we use parentheses to denote function arguments, as in $T(u, v, w)$. When a tensor is viewed as an array, we use subscripts to denote array entries, as in $T_{i,j,k}$.

We illustrate by continuing the example from the previous section where T is a 3-tensor. We have

$$T(u, v, w) = \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \sum_{k=1}^{N_3} T_{i,j,k} u_i v_j w_k$$

for vectors u, v, w, and

$$T_{i,j,k} := T(e_i, e_j, e_k)$$

for indices i, j, k.

The action of a multilinear function on vectors equals the contraction of the associated array with those vectors. We use the the "$\cdot$" symbol to denote incomplete contraction by currying. That is,

$$T(u, v, \ \cdot\ )$$

denotes the Riesz representation of the linear functional $w \mapsto T(u, v, w)$ with respect to the Euclidean inner product. If $x = T(u, v, \ \cdot\ )$, then

$$x_k = \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} T_{i,j,k} u_i v_j, \quad k = 1, \ldots, N_3.$$

Using this notation, actions of T in the above example may be written as

$$\begin{aligned} A_1(v,w) &= T(\ \cdot\ , v, w), \\ A_2(u,w) &= T(u, \ \cdot\ , w), \\ A_3(u,v) &= T(u, v, \ \cdot\ ). \end{aligned}$$

1.2. Tensor train from the multilinear function perspective. From the conventional array perspective, the cores of a tensor train representation of T are 2- and 3-dimensional arrays,

$$\begin{aligned} C_1 &\in \mathbb{R}^{N_1 \times r_1}, \\ C_{k+1} &\in \mathbb{R}^{r_k \times N_{k+1} \times r_{k+1}}, \quad k = 1, 2, \ldots, d-2, \\ C_d &\in \mathbb{R}^{r_{d-1} \times N_d}, \end{aligned}$$

such that

$$T_{i_1, i_2, \ldots, i_d} = \sum_{i_1=1}^{r_1} \sum_{j_2=1}^{r_2} \cdots \sum_{j_{d-1}=1}^{r_{d-1}} (C_1)_{i_1, j_1} (C_2)_{j_1, i_2, j_2} \ldots (C_{d-1})_{j_{d-2}, i_{d-1}, j_{d-1}} (C_d)_{j_{d-1}, i_d}.$$

From the multilinear function perspective, this is a factorization of T into the composition of functions,

$$T(x_1, x_2, \ldots, x_d) = C_d(C_{d-1}(\ldots C_2(C_1(x_1, \ \cdot\ ), x_2, \ \cdot\ ) \ldots, x_{d-1}, \ \cdot\ ), x_d),$$

where the output from the last mode of each core is used as an input for the first mode of the next core.

2. Method. We construct the cores of the tensor train one at a time. Without loss of generality we assume $d \geq 5$. If $d < 5$, one may proceed as described for cores 1 through $d-1$, then skip to the method for computing the last core for core d.

The basic idea of the method is as follows. At each step (other than the first), we start with a factorization of T into a partially constructed tensor train, T_k, composed with an unknown multilinear remainder function, R_k. We construct vectors that, when input into T_k, yield the standard unit basis vectors, e_j, as output (recall $e_1 = (1, 0, \ldots, 0)$, $e_2 = (0, 1, 0, \ldots, 0)$, and so on). This allows us to apply R_k to arbitrary vectors through an indirect process that involves computing actions of T. By using a randomized range finding procedure that involves applying R_k to random vectors, we construct the next core in the tensor train. This process repeats until all cores are computed.

2.1. Multilinear randomized range finder. In the randomized singular value decomposition (randomized SVD) [23], one constructs a basis for the range of a matrix by applying the matrix to random input vectors, then orthogonalizing the resulting output vectors. Here we use a multilinear generalization of this idea to construct a basis for the numerical range of a vector valued multilinear function. Similar multilinear randomized range finders have been used in [10, 26].

Let F be a vector-valued multilinear function that takes n vectors as input and returns one vector as output. Let r be the dimension of the numerical range of F (or the desired dimension of an approximation to this range). First, we compute

$$y^{(i)} = F(\omega_1^{(i)}, \omega_2^{(i)}, \ldots, \omega_n^{(i)}), \quad i = 1, \ldots, r+p,$$

where $\omega_1^{(i)}, \omega_2^{(i)}, \ldots, \omega_n^{(i)}$ are random vectors of the appropriate sizes with independent normally distributed entries, and p is a small oversampling parameter (we use $p = 5$). Then we form an orthonormal basis for the span of the $y^{(i)}$ by computing the thin singular value decomposition (SVD) of the matrix that has y_i as its columns. That is,

$$\begin{bmatrix} y^{(1)} & \ldots & y^{(r+p)} \end{bmatrix} = U \Sigma V^T.$$

Finally, we set U_r to be the matrix consisting of the first r columns of U. The span of these r columns approximates the range of F.

2.2. First core. Define F_1 to be the following vector valued multilinear map:

$$F_1(x_2, \ldots, x_d) := T(\ \cdot\ , x_2, \ldots, x_d). \tag{3}$$

We use the randomized range finding procedure from section 2.1 to compute an orthonormal basis, U_r, for the range of F_1. Then we set $C_1 := U_r$.

2.3. Second core. If the span of the columns of C_1 accurately captures the range of F_1, then T factors into the composition

$$T(x_1, x_2, \ldots, x_d) = R_1\left(T_1(x_1), x_2, \ldots, x_d\right), \tag{4}$$

where

$$T_1(x_1) := C_1^T x_1,$$

and the "remainder" $R_1 : \mathbb{R}^{r_1} \times \mathbb{R}^{N_2} \times \cdots \times \mathbb{R}^{N_d} \to \mathbb{R}$ is an unknown multilinear function. To construct the next core, we seek an orthonormal basis for the range of the multilinear function F_2 defined as

$$F_2(x_3, \ldots, x_d) := R_1(\ \cdot\ ,\ \cdot\ , x_3, \ldots, x_d), \tag{5}$$

where the output is vectorized.

Let

$$\eta_j := (C_1)_{:,j} \tag{6}$$

be the jth column of C_1 (we use the colon subscript ":" to denote all array entries in a given axis). By orthogonality of C_1, we have $C_1^T \eta_j = e_j$, which implies

$$T_1(\eta_j) = e_j. \tag{7}$$

Combining (7) with (4), we have

$$R_1(e_j,\ \cdot\ , x_3, \ldots, x_d) = T(\eta_j,\ \cdot\ , x_3, \ldots, x_d).$$

Stacking these vectors yields

$$R_1(\ \cdot\ ,\ \cdot\ , x_3, \ldots, x_d) = \begin{bmatrix} T(\eta_1,\ \cdot\ , x_3, \ldots, x_d) \\ T(\eta_2,\ \cdot\ , x_3, \ldots, x_d) \\ \ldots \\ T(\eta_{r_1},\ \cdot\ , x_3, \ldots, x_d) \end{bmatrix}. \tag{8}$$

We may therefore construct an orthonormal basis, U_r, for the range of F_2 by using the randomized range finding procedure described in section 2.1. Whenever the randomized range finding procedure requires evaluating the function F_2, we do so by forming the right-hand side of (8). This, in turn, is done by computing r_1 actions of T. The second core, C_2, is the $r_1 \times N_2 \times r_2$ third order tensor formed by reshaping the $(r_1 N_2) \times r_2$ matrix U_r. This process for constructing the second core is summarized in Algorithm 1.

Algorithm 1 Construction of the second core.

Require: Core C_1.
Ensure: Core C_2.
1: Form the vectors $\eta_1, \eta_2, \ldots, \eta_{r_1}$ according to (6).
2: Compute an orthonormal basis, $U_r \in \mathbb{R}^{(r_1 N_2) \times r_2}$, for the range of F_2 using the randomized range finder in section 2.1, in which (8) is used to evaluate F_2 as needed within the randomized range finding procedure.
3: Set C_2 to be the $r_1 \times N_2 \times r_2$ reshaped version of U_r.

2.4. Third core. Having computed the first and second cores, the tensor train now factors into the composition

$$T(x_1, x_2, x_3, x_4, \ldots, x_d) = R_2\left(T_2(x_1, x_2), x_3, x_4, \ldots, x_d\right), \tag{9}$$

where

$$T_2(x_1, x_2) := C_2\left(C_1^T x_1, x_2, \ \cdot\ \right),$$

and $R_2 : \mathbb{R}^{r_2} \times \mathbb{R}^{N_3} \times \cdots \times \mathbb{R}^{N_d} \to \mathbb{R}$ is an unknown multilinear remainder. Here $C_2\left(C_1^T x_1, x_2, \ \cdot\ \right)$ denotes the contraction of C_2 with $C_1^T x_1$ in the first mode and x_2 in the second mode.

We must now construct an orthonormal basis for the function F_3 defined as

$$F_3(x_4, \ldots, x_d) := R_2\left(\ \cdot\ ,\ \cdot\ , x_4, \ldots, x_d\right), \tag{10}$$

where we view the vectorization of the first two modes of R_2 as the output and the remaining modes as the inputs. We seek to find a small number, τ, of vectors $\{\xi_i\}_{i=1}^{\tau}$ and $\{\eta_{i,j}\}_{i=1\ j=1}^{\tau\ \ r_2}$ so that

$$\sum_{i=1}^{\tau} T_2(\xi_i, \eta_{i,j}) = e_j, \qquad j = 1, \ldots, r_2, \tag{11}$$

because then (9) implies

$$R_2\left(e_j, \ \cdot\ , x_4, \ldots, x_d\right) = \sum_{i=1}^{\tau} T(\xi_i, \eta_{i,j}, \ \cdot\ , x_4, \ldots, x_d),$$

which implies

$$R_2\left(\ \cdot\ ,\ \cdot\ , x_4, \ldots, x_d\right) = \sum_{i=1}^{\tau} \begin{bmatrix} T(\xi_i, \eta_{i,1}, \ \cdot\ , x_4, \ldots, x_d) \\ T(\xi_i, \eta_{i,2}, \ \cdot\ , x_4, \ldots, x_d) \\ \vdots \\ T(\xi_i, \eta_{i,r_2}, \ \cdot\ , x_4, \ldots, x_d) \end{bmatrix}. \tag{12}$$

Given ξ_i, $\eta_{i,j}$ satisfying (11), we compute an $(r_2 N_3) \times r_3$ orthonormal basis, U_r, for the range of F_3 using the randomized range finder from section 2.1. Then we set C_3 to be the $r_2 \times N_3 \times r_3$ reshaping of U_r into a 3-tensor. Whenever the randomized range finder requires evaluating F_3, we perform the evaluation by computing the right-hand side of (12) for $j = 1, \ldots, r_2$ by computing τr_2 actions of T.

We now describe how to find ξ_i, $\eta_{i,j}$ satisfying (11). We choose

$$\xi_i := (C_1)_{:,i} \tag{13}$$

as the ith column of C_1. Other choices are possible. We choose (13) since (a) the randomized range finding procedure is likely to be more accurate for vectors in the span of the first columns of C_1 and less accurate for vectors in the span of the later columns, and (b) with the core C_1 already computed, the following cores need only be accurate for vectors x_1 in the column space of C_1.

Given these ξ_i vectors, we may now solve a least squares problem to construct acceptable $\eta_{i,j}$. Let M_i, $i = 1, \ldots, \tau$, be the $N_2 \times r_2$ matrices

$$M_i := T_2(\xi_i, \ \cdot \ ) \tag{14}$$

formed by contracting the existing partially constructed tensor train with the vectors ξ_i. We may write (11) as the linear system equation

$$\begin{bmatrix} M_1^T & M_2^T & \ldots & M_\tau^T \end{bmatrix} \begin{bmatrix} \eta_{1,j} \\ \eta_{2,j} \\ \vdots \\ \eta_{\tau,j} \end{bmatrix} = e_j, \quad j = 1, \ldots, r_2. \tag{15}$$

We choose τ sufficiently large so that (15) is underdetermined and therefore generically solvable. Then for $j = 1, \ldots, r_2$, we find a least squares solution to (15) by QR factorization. This linear system has $\tau r_2 N_2$ variables satisfying r_2^2 equations, and is therefore underdetermined if $\tau \geq \lceil r_2/N_2 \rceil$. We recommend $\tau = \lceil r_2/N_2 \rceil + 1$ and use this in our numerical results. In the case when $r_2 < N_2$, this reduces to $\tau = 2$. Choosing $\tau = \lceil r_2/N_2 \rceil + 1$ instead of $\tau = \lceil r_2/N_2 \rceil$ improves performance of the method considerably in our numerical examples, while choosing larger τ does not. This process for constructing the third core is summarized in Algorithm 2.

A caveat here is that we cannot choose τ larger than r_1 because there are only r_1 fibers ξ_i to choose from C_1. If $\tau > r_1$ is required, the algorithm should backtrack and increase r_1. Backtracking may be avoided by setting a minimum rank for all cores. In our numerical results, we have $r_i < N_i$ and $\tau = \lceil r_i/N_i \rceil + 1 = 2$ for all cores, so we ensure that backtracking never occurs by setting a minimum rank of $r_i = 2$ for all of the cores.

2.5. Fourth through $(d-1)$th cores. The process for constructing the fourth through $(d-1)$th cores is similar to the process for constructing the third core. But now there are more modes that must be saturated with specially chosen vectors.

Suppose that we have already computed cores $C_1, C_2, \ldots, C_k$ for some k in the range $3 \leq k \leq d-2$, and let $T_k : \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_k} \to \mathbb{R}^{r_k}$ be defined recursively as

$$T_l(x_1, \ldots, x_{k-1}, x_k) := \begin{cases} C_1^T x_1, & l = 1, \\ C_l(T_{l-1}(x_1, \ldots, x_{k-1}), x_k, \ \cdot \ ), & l = 2, \ldots, k, \end{cases}$$

Algorithm 2 Construction of the third core.

Require: Cores C_1, C_2.
Ensure: Core C_3.
1: Form the vectors $\xi_1, \xi_2, \ldots, \xi_\tau$ according to (13).
2: Form the matrices $M_1, M_2, \ldots, M_\tau$ according to (14).
3: Find least-squares solutions to (15) for $j = 1, \ldots, r_2$ to get $\{\eta_{i,j}\}_{i=1 \; j=1}^{\tau \quad r_2}$.
4: Compute an orthonormal basis, $U_r \in \mathbb{R}^{(r_2 N_3) \times r_3}$, for the range of F_3 using the randomized range finder in section 2.1, in which (12) is used to evaluate F_3 as needed within the randomized range finding procedure.
5: Set C_3 to be the $r_2 \times N_3 \times r_3$ reshaped version of U_r.

where $C_l(T_{l-1}(x_1, \ldots, x_{k-1}), x_k, \, \cdot \,)$ denotes contraction of C_l with $T_{l-1}(x_1, \ldots, x_{k-1})$ in the first mode and x_k in the second mode. We have the factorization

$$T(x_1, \ldots, x_k, x_{k+1}, \ldots, x_d) = R_k \left(T_k(x_1, \ldots, x_k), x_{k+1}, \ldots, x_d \right), \tag{16}$$

where $R_k : \mathbb{R}^{r_k} \times \mathbb{R}^{N_{k+1}} \times \cdots \times \mathbb{R}^d \to \mathbb{R}$ is an unknown multilinear function.

Given vectors $\psi_1, \psi_2, \ldots, \psi_{k-2}$, $\{\xi_i\}_{i=1}^{\tau}$, and $\{\eta_{i,j}\}_{i=1 \; j=1}^{\tau \quad r_k}$ satisfying

$$\sum_{i=1}^{\tau} T_k \left(\psi_1, \ldots, \psi_{k-2}, \xi_i, \eta_{i,j} \right) = e_j, \tag{17}$$

we have

$$R_k \left(\, \cdot \, , \, \cdot \, , x_{k+2}, \ldots, x_d \right) = \sum_{i=1}^{\tau} \begin{bmatrix} T(\psi_1, \ldots, \psi_{k-2}, \xi_i, \eta_{i,1}, \, \cdot \, , x_{k+2}, \ldots, x_d) \\ T(\psi_1, \ldots, \psi_{k-2}, \xi_i, \eta_{i,2}, \, \cdot \, , x_{k+2}, \ldots, x_d) \\ \vdots \\ T(\psi_1, \ldots, \psi_{k-2}, \xi_i, \eta_{i,r_k}, \, \cdot \, , x_{k+2}, \ldots, x_d) \end{bmatrix} \tag{18}$$

by the same argument as the analogous result we presented for the third core. We construct an orthonormal basis, U_r, for the range of F_{k+1} defined as

$$F_{k+1}(x_{k+2}, \ldots, x_d) := R_k(\, \cdot \, , \, \cdot \, , x_{k+2}, \ldots, x_d) \tag{19}$$

using the randomized range finder from section 2.1. Within the randomized range finder we use identity (18) to evaluate F_{k+1} as needed. We set C_{k+1} to be the $r_k \times N_{k+1} \times r_{k+1}$ reshaping of U_r.

For the vectors $\psi_1, \psi_2, \ldots, \psi_{k-2}$, we choose ψ_j to be the "first fibers" of the corresponding cores C_j, which represent the "highest energy" in each subspace formed by C_j, i.e.,

$$\psi_l := \begin{cases} (C_1)_{:,1} \, , & l = 1, \\ (C_l)_{1,:,1} \, , & l = 2, \ldots, k-2. \end{cases} \tag{20}$$

For the vectors $\{\xi_i\}_{i=1}^{\tau}$ and $\{\eta_{i,j}\}_{i=1 \; j=1}^{\tau \quad r_k}$ satisfying (17), we use the same process as that for the third core, i.e., we specify

$$\xi_i := (C_{k-1})_{1,:,i} \, , \quad i = 1, \ldots, \tau, \tag{21}$$

and form the $N_k \times r_k$ matrices

$$M_i := T_k\left(\psi_1, \ldots, \psi_{k-2}, \xi_i, \ \cdot\ \right), \tag{22}$$

and then use a QR factorization to find the least-squares solutions of the linear systems

$$\begin{bmatrix} M_1^T & M_2^T & \ldots & M_\tau^T \end{bmatrix} \begin{bmatrix} \eta_{1,j} \\ \eta_{2,j} \\ \vdots \\ \eta_{\tau,j} \end{bmatrix} = e_j \tag{23}$$

for $j = 1, \ldots, r_k$. As in the case of the third core, we must have $\tau \geq \lceil r_k / N_k \rceil$ for this system to be solvable; in our numerical experiments we observe good results with $\tau = \lceil r_k / N_k \rceil + 1$. This process for computing the kth through $(d-1)$th cores is summarized in Algorithm 3. We use graphical tensor notation to illustrate this process in Figure 1.

Since we cannot choose τ larger than r_{k-1}, if $\tau > r_{k-1}$ is required, then the algorithm should backtrack and increase r_{k-1}. In our numerical results, backtracking is avoided by setting a minimum rank of $r_i = 2$ for all of the cores.

Algorithm 3 Construction of one of the fourth through $(d-1)$th cores.

Require: Cores $C_1, C_2, \ldots, C_k, \quad 3 \leq k \leq d-2$.
Ensure: Core C_{k+1}.

1: Form the vectors $\psi_1, \psi_2, \ldots, \psi_{k-2}$ according to (20).
2: Form the vectors $\xi_1, \xi_2, \ldots, \xi_\tau$ according to (21).
3: Form the matrices $M_1, M_2, \ldots, M_\tau$ according to (22).
4: Find least-squares solutions to (23) for $j = 1, \ldots, r_k$ to get $\{\eta_{i,j}\}_{i=1}^{\tau}{}_{j=1}^{r_k}$.
5: Compute an orthonormal basis, $U_r \in \mathbb{R}^{(r_k N_{k+1}) \times r_{k+1}}$, for the range of F_{k+1} using the randomized range finder in section 2.1, in which (18) is used to evaluate F_{k+1} as needed within the randomized range finding procedure.
6: Set C_{k+1} to be the $r_k \times N_{k+1} \times r_{k+1}$ reshaped version of U_r.

2.6. Last core.

For the last core, we have

$$T(x_1, \ldots, x_{d-1}, \ \cdot\ ) = R_{d-1}\left(T_{d-1}(x_1, \ldots, x_{d-1}), \ \cdot\ \right).$$

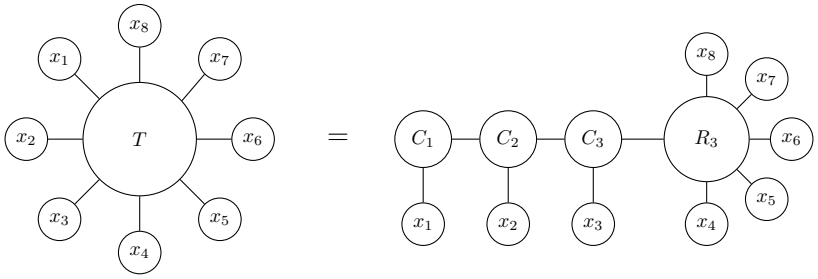
We specify the vectors $\psi_1, \ldots, \psi_{d-3}$, $\{\xi_i\}_{i=1}^{\tau}$, and find vectors $\{\eta_{i,j}\}_{i=1}^{\tau}{}_{j=1}^{r_{d-1}}$ such that

$$\sum_{i=1}^{\tau} T_{d-1}(\psi_1, \ldots, \psi_{d-3}, \xi_i, \eta_{i,j}) = e_j, \quad j = 1, \ldots, r_{d-1},$$

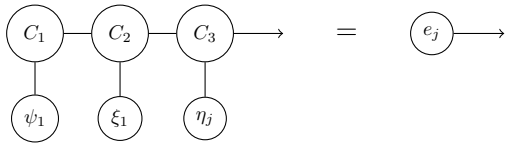
using the same process as described in section 2.5 for previous cores. Then we directly form the last core, $C_d \in \mathbb{R}^{r_{d-1} \times N_d}$, as follows:

$$C_d := \begin{bmatrix} R_{d-1}\left(e_1, \ \cdot\ \right)^T \\ R_{d-1}\left(e_2, \ \cdot\ \right)^T \\ \vdots \\ R_{d-1}\left(e_{r_{d-1}}, \ \cdot\ \right)^T \end{bmatrix} = \sum_{i=1}^{\tau} \begin{bmatrix} T(\psi_1, \ldots, \psi_{d-3}, \xi_i, \eta_{i,1}, \ \cdot\ )^T \\ T(\psi_1, \ldots, \psi_{d-3}, \xi_i, \eta_{i,2}, \ \cdot\ )^T \\ \vdots \\ T(\psi_1, \ldots, \psi_{d-3}, \xi_i, \eta_{i,r_{d-1}}, \ \cdot\ )^T \end{bmatrix}. \tag{24}$$

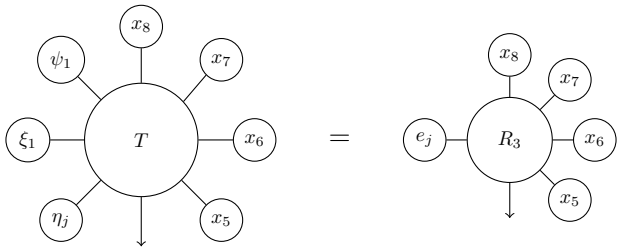
There is no orthogonalization for the last core. This process for constructing the last core is summarized in Algorithm 4.



(a) Intermediate factorization of T after three cores have been computed.



(b) We solve for η_j so that this equality holds.



(c) Implicit application of R_3 via application of T to specially chosen vectors.

FIG. 1. *Illustration of an intermediate step in the peeling process in the case where* $\tau = 1$ *(typically* $\tau > 1$*, and the left hand sides of* (b) *and* (c) *are sums of tensors of the form shown here). Given an intermediate tensor train factorization,* (a)*, we find certain vectors so that equality holds in* (b)*; then we apply the unknown right factor to vectors implicitly by applying* T *to certain vectors,* (c).

2.7. Adaptive range finding. The randomized range finding procedure may be performed in a sequential manner so that the rank of a core, r_k, is found while the orthogonal basis for that core is constructed. Starting with a small r (say, $r = 2$), compute $y^{(i)}, i = 1, \ldots, r+p$ and U_r, and compute the error estimate

$$\mathcal{E} := \max_{i=1,\ldots,r+p} \left\| y^{(i)} - U_r U_r^T y^{(i)} \right\|_2 . \tag{25}$$

If $\mathcal{E}$ is less than a predetermined tolerance, set $r_k = r$ and stop the range finding procedure, and reshape U_r to construct the current core. If $\mathcal{E}$ is greater than the tolerance, increase r (say, $r \leftarrow r + 1$), compute more vectors $y^{(i)}$, recompute U_r, and repeat the process. Error estimator (25) generalizes the a posteriori error estimator in section 4.3 of [23] to tensors. For the numerical results presented in section 4, we fix the rank r beforehand and report results based on other more robust and accurate (but more expensive) methods for computing error.

Algorithm 4 Construction of the last core.

Require: Cores $C_1, C_2, \ldots, C_{d-1}$.
Ensure: Core C_d.
1: Form the vectors $\psi_1, \psi_2, \ldots, \psi_{d-3}$ according to (20).
2: Form the vectors $\xi_1, \xi_2, \ldots, \xi_\tau$ according to (21).
3: Form the matrices $M_1, M_2, \ldots, M_\tau$ according to (22).
4: Compute least-squares solutions to (23) for $j = 1, \ldots, r_{d-1}$ to get $\{\eta_{i,j}\}_{i=1}^{\tau}{}_{j=1}^{r_{d-1}}$.
5: Construct C_d by evaluating the right-hand side of (24).

3. Computing the action of high order derivative tensors. In this section we present efficient methods for computing the action of high order derivative tensors for a quantity of interest

$$\mathcal{F}(m, u(m)) \in \mathbb{R}^{N_q}, \tag{26}$$

which depends on a parameter $m \in \mathbb{R}^{N_m}$ implicitly through the solution (state variable) $u \in \mathbb{R}^{N_u}$ of a nonlinear state equation

$$\mathcal{G}(m, u) = 0. \tag{27}$$

Computation of the entries of the derivative tensor

$$\frac{d^k \mathcal{F}}{dm^k} \in \mathbb{R}^{N_m} \times \cdots \times \mathbb{R}^{N_m} \times \mathbb{R}^{N_q} \tag{28}$$

for large N_m and k is prohibitive. We show how to compute the action of $S := \frac{d^k \mathcal{F}}{dm^k}$ as a multilinear function. In section 3.1, we show how to compute the action of the derivative tensor when the output mode is free, that is,

$$S(p_1, \ldots, p_k, \ \cdot\ ) = \frac{d^k \mathcal{F}}{dm^k}\ p_1\ \ldots\ p_k.$$

In section 3.2, we show how to compute the action of the derivative tensor when a derivative mode is free, that is,

$$S(\ \cdot\ , p_2, \ldots, p_k, q) := q^T \left(\frac{d^k \mathcal{F}}{dm^k}\ p_2\ p_3\ \ldots\ p_k \right).$$

Typically one can evaluate directional partial derivatives of $\mathcal{F}$ and $\mathcal{G}$ with respect to m and u with automatic differentiation, but one cannot easily evaluate total derivatives of $\mathcal{F}$ with respect to m, because that would require automatically differentiating through the solution procedure for the state equation (e.g., differentiating through an iterative Newton–Krylov solver). The basic idea of this section, therefore, is to convert the problem of computing total derivatives into a problem of computing partial derivatives and solving linear systems. We show how one can evaluate the action of S via procedures that involve solving a sequence of linear systems of the forms

$$0 = \frac{\partial \mathcal{G}}{\partial u}\ w + b \qquad \text{and} \qquad 0 = \left(\frac{\partial \mathcal{G}}{\partial u} \right)^T w + b, \tag{29}$$

where constructing the right-hand sides b only requires evaluating directional partial derivatives of $\mathcal{F}$ and $\mathcal{G}$, and quantities that have already been computed. For more

on automatic differentiation, we recommend [36]. A related implementation of computational methods for PDE-constrained high order derivative actions in the finite element package FEniCS [34] can be found in [35].

3.1. High order derivative actions with the output mode free. To compute the action of S when the output mode is free, we will repeatedly differentiate $\mathcal{F}$ using the chain rule for total derivatives. This will allow us to express high order total derivatives of $\mathcal{F}$ in terms of partial derivatives of $\mathcal{F}$ and total derivatives of u. Repeatedly differentiating the state equation, $0 = \mathcal{G}$, in the same manner yields equations that may be solved to determine the required total derivatives of u.

Zeroth derivative. To compute $\mathcal{F}(m, u(m))$, we solve $\mathcal{G}(m, u) = 0$ for u, then compute $\mathcal{F}(m, u)$.

First derivative. The chain rule for total derivatives yields

$$\frac{d\mathcal{F}}{dm}\, p_a = \frac{\partial \mathcal{F}}{\partial m}\, p_a + \frac{\partial \mathcal{F}}{\partial u}\frac{du}{dm}\, p_a = \frac{\partial \mathcal{F}}{\partial m}\, p_a + \frac{\partial \mathcal{F}}{\partial u}\, u^{\{a\}}, \tag{30}$$

where p_a is the direction in which the derivative is taken, and

$$u^{\{a\}} := \frac{du}{dm}\, p_a$$

is unknown. Differentiating both sides of the state equation, $0 = \mathcal{G}$, yields

$$\frac{d}{dm}(0)\, p_a = \frac{d}{dm}(\mathcal{G})\, p_a$$

or

$$0 = \frac{\partial \mathcal{G}}{\partial m}\, p_a + \frac{\partial \mathcal{G}}{\partial u}\, u^{\{a\}}, \tag{31}$$

which is a new equation that may be solved for $u^{\{a\}}$. To compute $\frac{d\mathcal{F}}{dm}\, p_a$ we solve (31) for $u^{\{a\}}$, then evaluate the right-hand side of (30).

Second derivative. Let

$$\mathcal{F}^{\{a\}} := \frac{d\mathcal{F}}{dm}\, p_a.$$

Using the chain rule for total derivatives again, we have

$$\frac{d^2\mathcal{F}}{dm^2}\, p_a\, p_b = \frac{d\mathcal{F}^{\{a\}}}{dm}\, p_b = \frac{\partial \mathcal{F}^{\{a\}}}{\partial m}\, p_b + \frac{\partial \mathcal{F}^{\{a\}}}{\partial u}\, u^{\{b\}} + \frac{\partial \mathcal{F}^{\{a\}}}{\partial u^{\{a\}}}\, u^{\{a,b\}}, \tag{32}$$

where

$$u^{\{b\}} := \frac{du}{dm}\, p_b \quad \text{and} \quad u^{\{a,b\}} := \frac{d^2u}{dm^2}\, p_a\, p_b$$

are unknown. We may determine $u^{\{b\}}$ by solving a linear system of the form (31), except with p_b replacing p_a. To determine the equation that $u^{\{a,b\}}$ satisfies, we differentiate (31) in direction p_b. Let

$$\mathcal{G}^{\{a\}} := \frac{d\mathcal{G}}{dm}\, p_a,$$

so that (31) may be written as $0 = \mathcal{G}^{\{a\}}$. We have

$$\frac{d}{dm}(0)\, p_b = \frac{d}{dm}\left(\mathcal{G}^{\{a\}}\right) p_b$$

or

$$0 = \frac{\partial \mathcal{G}^{\{a\}}}{\partial m}\, p_b + \frac{\partial \mathcal{G}^{\{a\}}}{\partial u}\, u^{\{b\}} + \frac{\partial \mathcal{G}^{\{a\}}}{\partial u^{\{a\}}}\, u^{\{a,b\}}, \tag{33}$$

which may be solved for $u^{\{a,b\}}$. Once $u^{\{a\}}$ and $u^{\{a,b\}}$ have been determined, we compute $\frac{d^2\mathcal{F}}{dm^2}\, p_a\, p_b$ by evaluating the right-hand side of (32).

High order derivatives. We may repeatedly differentiate $\mathcal{F}$ and $\mathcal{G}$ to construct derivatives of any order. Define

$$\begin{aligned}
\mathcal{F}^{\alpha} &:= \frac{d^k \mathcal{F}}{dm^k}\, p_{\alpha_1}\, p_{\alpha_2}\, \cdots\, p_{\alpha_k}, \\
\mathcal{G}^{\alpha} &:= \frac{d^k \mathcal{G}}{dm^k}\, p_{\alpha_1}\, p_{\alpha_2}\, \cdots\, p_{\alpha_k}, \\
u^{\alpha} &:= \frac{d^k u}{dm^k}\, p_{\alpha_1}\, p_{\alpha_2}\, \cdots\, p_{\alpha_k},
\end{aligned}$$

where α is a multi-index of k derivative directions (e.g., $\alpha = \{a, b, b\}$, $k = 3$). If $\widetilde{\alpha} := \{\alpha_1, \alpha_2, \ldots, \alpha_{k-1}\}$ is a multi-index of $k-1$ derivative directions created by removing one derivative direction from α, then we may generate $\mathcal{F}^{\alpha}$ by differentiating $\mathcal{F}^{\widetilde{\alpha}}$ using the chain rule of total derivatives. This yields

$$\begin{aligned}
\mathcal{F}^{\alpha} = \frac{d\mathcal{F}^{\widetilde{\alpha}}}{dm}\, p_{\alpha_k} &= \frac{\partial \mathcal{F}^{\widetilde{\alpha}}}{\partial m}\, p_{\alpha_k} + \sum_{u^{\beta}} \frac{\partial \mathcal{F}^{\widetilde{\alpha}}}{\partial u^{\beta}} \frac{du^{\beta}}{dm} p_{\alpha_k} \\
&= \frac{\partial \mathcal{F}^{\widetilde{\alpha}}}{\partial m}\, p_{\alpha_k} + \sum_{u^{\beta}} \frac{\partial \mathcal{F}^{\widetilde{\alpha}}}{\partial u^{\beta}} u^{\beta \cup \{\alpha_k\}},
\end{aligned} \tag{34}$$

where the sums are taken over all variables u^{β} that $\mathcal{F}^{\widetilde{\alpha}}$ depends on. If we already have a computer code that computes $\mathcal{F}^{\widetilde{\alpha}}$, then we may use automatic differentiation to create a computer code that computes $\mathcal{F}^{\alpha}$ by using automatic differentiation for each partial derivative in the sum in (34). The code for computing any derivative $\mathcal{F}^{\alpha}$ may be built by repeated application of this process, by differentiating $\mathcal{F}$ to get $\mathcal{F}^{\{\alpha_1\}}$, differentiating again to get $\mathcal{F}^{\{\alpha_1,\alpha_2\}}$, differentiating again to get $\mathcal{F}^{\{\alpha_1,\alpha_2,\alpha_3\}}$, and so on.

A straightforward inductive argument[2] shows that $\mathcal{F}^{\alpha}$ depends only on u^{β} for all multiset subsets $\beta \subseteq \alpha$. For example, if $\alpha = \{a, b, b\}$, then $\mathcal{F}^{\alpha}$ depends on $u = u^{\{\}}$, $u^{\{a\}}$, $u^{\{b\}}$, $u^{\{a,b\}}$, $u^{\{b,b\}}$, and $u^{\{a,b,b\}}$. Thus (34) may be written as

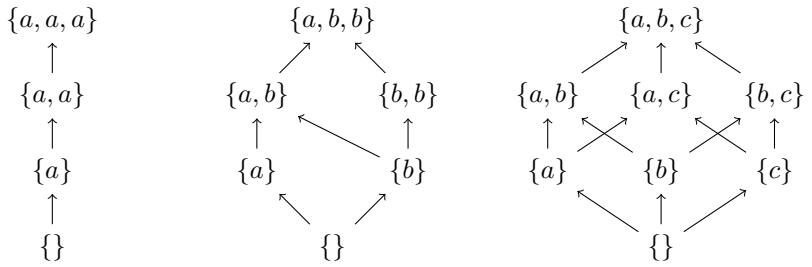
$$\mathcal{F}^{\alpha} = \frac{\partial \mathcal{F}^{\widetilde{\alpha}}}{\partial m}\, p_{\alpha_k} + \sum_{\beta \subseteq \widetilde{\alpha}} \frac{\partial \mathcal{F}^{\widetilde{\alpha}}}{\partial u^{\beta}} u^{\beta \cup \{\alpha_k\}}, \tag{35}$$

where the sum is explicitly written over all multiset subsets $\beta \subset \widetilde{\alpha}$.

We may form $\mathcal{G}^{\alpha}$ from $\mathcal{G}^{\widetilde{\alpha}}$ in the same manner as

$$\mathcal{G}^{\alpha} = \frac{\partial \mathcal{G}^{\widetilde{\alpha}}}{\partial m}\, p_{\alpha_k} + \sum_{\beta \subseteq \widetilde{\alpha}} \frac{\partial \mathcal{G}^{\widetilde{\alpha}}}{\partial u^{\beta}} u^{\beta \cup \{\alpha_k\}}. \tag{36}$$

[2]For the base case verify that $\mathcal{F}$ depends on u, and for the inductive step use (34).



(a) Symmetric (b) Partially symmetric (c) Nonsymmetric

FIG. 2. *High order quantities of interest, $\mathcal{F}^\alpha$, high order state variables, u^α, and high order state equations, $0 = \mathcal{G}^\alpha$, are indexed by multi-indices, α, consisting of derivative directions. These multi-indices form a lattice ordered by multiset inclusion. A variable u^α depends on the variables u^β for all β that precede α. We compute high order derivatives by working up the lattice, computing u^α at each node by solving $0 = \mathcal{G}^\alpha$. The more symmetric the multi-index α, the fewer the variables that must be computed. The equations associated with all nodes in a given level of the lattice may be solved in parallel.*

Another straightforward argument by induction[3] shows that there is only one term in the sum on the right-hand side of (36) that contains u^α, which takes the form $\frac{\partial \mathcal{G}}{\partial u} u^\alpha$. Thus the equation $0 = \mathcal{G}^\alpha$, for any $|\alpha| \geq 1$, may be written in the form

$$0 = \frac{\partial \mathcal{G}}{\partial u}\, u^\alpha + b^\alpha, \tag{37}$$

where b^α depends on u^β for all *strict* multiset subsets $\beta \subset \alpha$, that is, all multiset subsets of α, excluding α itself. We may construct b^α by repeatedly automatically differentiating $\mathcal{G}$ via (36), then excluding the term that contains u^α in the result.

To determine u^α, we need to solve (37), which requires u^β for all strict multiset subsets $\beta \subset \alpha$. The dependency structure for the variables u^α and equations $0 = \mathcal{G}^\alpha$ therefore forms a bounded lattice consisting of the multiset subsets of α, partially ordered by multiset inclusion (see Figure 2). Thus we may compute u^α by solving the equations $0 = \mathcal{G}^\beta$ for the variables u^β for all multiset subsets $\beta \subseteq \alpha$, in an order such that each variable u^β is solved for after all of the variables corresponding to multiset subsets of β have been solved for (e.g., in the order determined by topologically sorting the lattice). This is shown in Algorithm 5. Once these variables are computed, we may compute the desired result,

$$S(p_{\alpha_1}, \ldots, p_{\alpha_k}, \ \cdot\ ) = \mathcal{F}^\alpha, \tag{38}$$

by evaluating the code for $\mathcal{F}^\alpha$ that we generated by repeated automatic differentiation.

3.2. High order derivative actions with a derivative mode free. We may use a similar strategy to compute derivative tensor actions where one of the derivative modes is free. To that end, we define the Lagrangian

$$\mathcal{L}(m, u, \lambda) := q^T \mathcal{F}(u) + \lambda^T \mathcal{G}(m, u),$$

where $\lambda \in \mathbb{R}^{N_u}$ is the adjoint variable for enforcing the state equation constraint, and $q \in \mathbb{R}^{N_q}$ is an arbitrary direction in which we measure the quantity of interest.

[3]The base case is shown in (31), and the inductive step follows from the chain rule of total derivatives.

Algorithm 5 Computation of $\mathcal{F}^\alpha = \frac{d^k \mathcal{F}}{dm^k}\ p_{\alpha_1}\ \cdots\ p_{\alpha_k}$.

procedure COMPUTE_$\mathcal{F}$_DERIVATIVE(α)
    compute_u_derivatives(α)
    Construct $\mathcal{F}^\alpha$

procedure COMPUTE_u_DERIVATIVES(α)
    if u^α has already been computed **then**
        Do nothing
    else if $\alpha = \{\}$ **then**
        Solve $0 = \mathcal{G}(m, u)$ for u
    else
        for all multiset subsets $\beta \subset \alpha$ with $|\beta| = |\alpha| - 1$ **do**
            compute_u_derivatives(β)
        Construct b^α
        Solve (37) for u^α

From the theory of Lagrange multipliers, we have the following formula for the first derivative:

$$q^T \left(\frac{d\mathcal{F}}{dm} \right) = \frac{\partial \mathcal{L}}{\partial m} = \lambda^T \left(\frac{\partial \mathcal{G}}{\partial m} \right), \tag{39}$$

where $\frac{\partial \mathcal{L}}{\partial m}$ is evaluated at the state u, which solves the state equation

$$0 = \frac{\partial \mathcal{L}}{\partial \lambda} = \mathcal{G}(m, u), \tag{40}$$

and at the adjoint λ, which solves the adjoint equation

$$0 = \frac{\partial \mathcal{L}}{\partial u} = q^T \frac{\partial \mathcal{F}}{\partial u} + \lambda^T \left(\frac{\partial \mathcal{G}}{\partial u} \right). \tag{41}$$

This process of computing $q^T \left(\frac{d\mathcal{F}}{dm} \right)$ has the same structure as the process for computing $\mathcal{F}$: we solve a linear system, then evaluate a vector-valued function that depends on the result. Here $\frac{\partial \mathcal{L}}{\partial m}$ takes the place of $\mathcal{F}$, the combined vector (u, λ) takes the place of u, and the combined state and adjoint system (40) and (41) takes the place of the state equation. We may therefore compute the desired high order derivatives

$$q^T \left(\frac{d^k \mathcal{F}}{dm^k}\ p_2\ p_3\ \cdots\ p_k \right)$$

by differentiating (39), (40), and (41) and solving linear systems repeatedly, as was done in section 3.1, except with the replacements

$$\mathcal{F} \longrightarrow \frac{\partial \mathcal{L}}{\partial m}, \qquad u \longrightarrow \begin{bmatrix} u \\ \lambda \end{bmatrix}, \qquad 0 = \mathcal{G}(m, u) \longrightarrow \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \lambda}(m, u) \\ \frac{\partial \mathcal{L}}{\partial u}(m, u, \lambda) \end{bmatrix}.$$

The resulting high order forward equations,

$$0 = \frac{d^k}{dm^k} \left(\frac{\partial \mathcal{L}}{\partial \lambda} \right)\ p_{\alpha_1}\ \cdots\ p_{\alpha_{k-1}},$$

are the same as the high order forward equations (37). The resulting high order adjoint equations,

$$0 = \frac{d^k}{dm^k}\left(\frac{\partial \mathcal{L}}{\partial u}\right) \; p_{\alpha_1} \; \cdots \; p_{\alpha_{k-1}},$$

take the form

$$0 = \left(\frac{\partial \mathcal{G}}{\partial u}\right)^T \lambda^\alpha + c^\alpha,$$

where $\lambda^\alpha := \frac{d^k \lambda}{dm^k} \; p_{\alpha_1} \; \cdots \; p_{\alpha_{k-1}}$, and c^α depends on u^β for all multiset subsets $\beta \subseteq \alpha$ and depends on λ^γ for all strict multiset subsets $\gamma \subset \alpha$.

3.3. Cost to compress derivative tensors. The method we presented in section 2 requires $O(\lceil r/N \rceil d r^2)$ tensor actions to construct a tensor train. But what is the cost of each of these tensor actions? As per the discussion in sections 3.1 and 3.2, each high order derivative tensor action requires solving many linear systems of the forms shown in (29). The number of linear systems that must be solved depends on how symmetric the action of the derivative tensor is. When all the derivative directions are the same, $O(d)$ linear systems must be solved. When all of the derivative directions are distinct, $O(2^d)$ linear systems must be solved. When some directions are the same and some are distinct, an intermediate number of linear systems must be solved. For our method, all derivative directions are distinct, so the number of linear systems that must be solved to construct the tensor train scales as $O(2^d \lceil r/N \rceil d r^2)$.

Despite the exponential scaling 2^d for a dth order derivative, our method possesses many desirable properties for compressing derivative tensors:

- The $O(2^d)$ linear solves computing a tensor action are trivially parallelizable to $O(d)$, because the high order variables within each layer of the multiset lattice (e.g., Figure 2) do not depend on each other and can therefore be solved for in parallel.
- All of the linear systems that must be solved when constructing the tensor train have the same coefficient matrix; they differ only in the right-hand side vectors. We therefore construct solvers or preconditioners once, and reuse them for all of the linear systems that must be solved. Constructing solvers or preconditioners is often the most expensive step for solving linear systems.
- Up to, say, fifth or sixth derivatives (tensor orders $d = 6$ or $d = 7$), the cost is tractible despite the exponential scaling. This is a substantial improvement over existing methods (e.g., Tucker-based methods), which typically become intractible beyond two or three derivatives.

It is possible to increase the symmetry of the tensor actions within the tensor train construction process, and therefore make the method substantially cheaper, by using the same vector $\psi := \psi_1 = \psi_2 = \cdots = \psi_{d-3}$, and using the same random vector $\omega_1^{(i)} = \omega_2^{(i)} = \cdots = \omega_{d-1}^{(i)}$ for all the derivative modes. These modifications would reduce the number of linear solves per tensor action to $O(d)$, but they may also make the method less robust. We do not use them in our numerical results. At present it is unclear which vector ψ should be chosen. The performance and robustness of the method seem sensitive to the choice of the vectors ψ_l.

4. Numerical results. In section 4.1, we compress the Hilbert tensor (a standard test case for tensor compression methods). Tensor entries of the Hilbert tensor

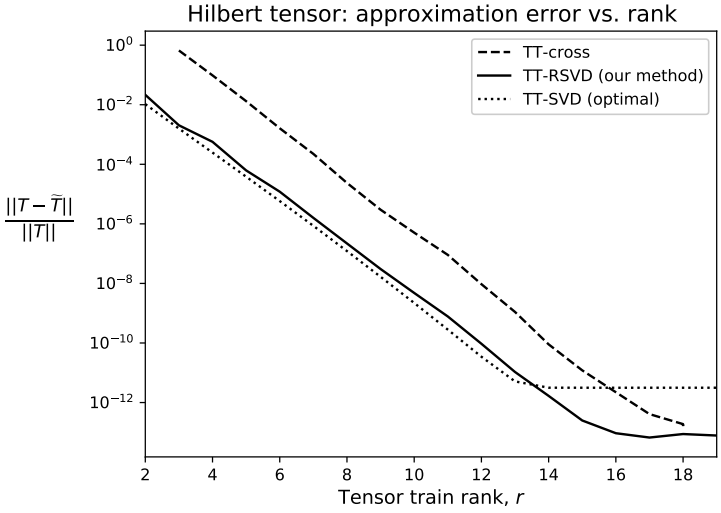


FIG. 3. Hilbert tensor. *Comparison of compression results for the Hilbert tensor using TT-cross, our method (TT-RSVD), and the conventional dense SVD-based algorithm (TT-SVD).*

are easily computable, so we use this test case to compare the accuracy of our method to that of conventional tensor train compression methods.

In section 4.2, we compress high order derivative tensors of a quantity of interest that depends on a parameter field implicitly through the solution of a partial differential equation. These derivative tensors are huge for fine meshes, and are available only through their action on vectors; they cannot be compressed into tensor train format efficiently using existing methods. The number of tensor actions for approximating the kth derivative tensor using our method is $O\left((k+1)r^2\right)$. For TT-cross, the number of tensor actions would be $O\left(N(k+1)r^2\right)$. We do not compare our method to TT-cross for the derivative tensors in section 4.2 because the factor of N makes TT-cross prohibitively expensive. For example, one of the tensors we will compress is a fourth derivative tensor with $r = 30$ and a 40×40 mesh (part of the Taylor series in Figure 4). This requires approximately 10^4 tensor actions for our method and would require approximately 10^7 tensor actions for TT-cross.

4.1. Hilbert tensor. Here we compress the $41 \times 42 \times 43 \times 44 \times 45$ Hilbert tensor, T, with entries

$$T\left[i_1, i_2, i_3, i_4, i_5\right] = \frac{1}{i_1 + i_2 + i_3 + i_4 + i_5}.$$

In Figure 3, we compare our method with the conventional dense SVD-based method for constructing tensor trains, and with TT-cross [40]. For TT-cross, we use the `dmrg_cross()` function in the TT-Toolbox software package[4] [44]. We compute tensor train approximations, $\widetilde{T}$, of T for a sequence of ranks r. The same rank r is used for all cores. Then we reconstitute full tensors from the tensor train approximations and compute the relative errors $||T - \widetilde{T}||/||T||$, where $||\cdot||$ is the Frobenius norm (square root of the sum of squares of all tensor entries). We plot the relative error

[4]Version 2.2.2, downloaded from https://github.com/oseledets/TT-Toolbox.

against the rank r. Both TT-cross and our method (TT-RSVD) achieve compression results that are close to the conventional dense SVD-based method (TT-SVD), which is theoretically optimal if one ignores rounding error due to numerical precision. Our method performs slightly better than TT-cross.

4.2. High order derivatives of PDE-dependent quantity of interest. Here we use our method to build Taylor series surrogate models for the noise-whitened parameter-to-output map for a nonlinear stochastic PDE with boundary output. We define the state equation, $0 = \mathcal{G}(m, u)$, to be the following inhomogeneous nonlinear reaction-diffusion equation with Neumann boundary conditions:

$$\begin{cases} -\nabla \cdot e^m \nabla u + u^3 = \rho & \text{in } \Omega, \\ \nu \cdot e^m \nabla u = 0 & \text{on } \partial\Omega. \end{cases}$$

The parameter, $m \sim N(0, C)$, is a Gaussian random field with mean zero and covariance

$$C = (-\Delta + I)^{-2},$$

where Δ is the Laplacian defined in Ω with Neumann boundary conditions along $\partial\Omega$, and I is the identity operator. The domain is the unit square $\Omega = [0, 1]^2$, ν is the normal to the boundary, and the source term $\rho : \Omega \to \mathbb{R}$ is given by

$$\rho(c) = \exp\left(\frac{\|c - (0.5, 0.5)\|^2}{2\,(0.2)^2}\right),$$

where we use $c \in \Omega$ to denote a spatial point in Ω, to distinguish from x as the vector for the tensor used throughout this paper. We choose the quantity of interest, $\mathcal{F}$, to be the trace of u along the boundary as

$$q^T \mathcal{F}(u) := \int_{\partial\Omega} u\ q\ ds.$$

We assume $x \sim N(0, I)$ is a spatial white noise, or a Gaussian distribution with mean zero and covariance I, so that

$$p := C^{1/2} x = (-\Delta + I)^{-1} x$$

has a Gaussian distribution with mean zero and covariance C. The problem is discretized with P^1 finite elements on a mesh of triangles arranged in a regular grid.

We use our method to approximate the noise-whitened kth derivative tensor T, defined as

$$T(x_1, \ldots, x_k, q) := S(p_1, \ldots, p_k, q) = q^T \left(\frac{d^k \mathcal{F}}{dm^k}\ p_1\ \ \ldots\ \ p_k\right),$$

with a rank-r tensor train, $\widetilde{T}$. Note that for a kth order derivative, the tensor order is $d = k + 1$. By performing this tensor train compression for several of these derivative tensors, we approximate the noise-whitened parameter-to-output map,

$$f(x) := \mathcal{F}\left(u\left(C^{1/2} x\right)\right),$$

TABLE 1

Mesh scalability. *Rank r required to compress derivative tensors to relative error tolerance $\sigma_1(T-\widetilde{T})/\sigma_1(T) < \epsilon$, where $\epsilon = 10^{-2}$ or $\epsilon = 10^{-3}$, for a variety of mesh sizes and derivative orders. Meshes are triangles arranged in a regular rectilinear grid, ranging from 10×10 to 80×80. Derivative orders range from the first derivative (the Jacobian, a second order tensor) to fifth derivative (a sixth order tensor). For derivative orders $k = 2$ through $k = 5$, r is the tensor-train rank required using our method. For $k = 1$ (Jacobian), r is the matrix rank required using randomized SVD.*

	$\epsilon = 10^{-2}$					$\epsilon = 10^{-3}$				
Mesh	$k=1$	2	3	4	5	$k=1$	2	3	4	5
10×10	9	8	8	8	11	19	19	20	23	28
20×20	8	9	9	10	11	23	22	24	28	35
30×30	8	8	9	10	10	22	24	25	29	32
40×40	9	9	9	8	11	23	25	27	29	34
50×50	7	9	9	11	11	26	23	29	28	34
60×60	8	8	10	9	11	24	25	27	30	35
70×70	9	9	8	8	12	27	23	27	30	35
80×80	9	9	8	11	11	26	25	27	32	38

with a truncated Taylor series

$$f_k(x) = f(0) + \frac{df}{dx}(0)x + \frac{1}{2}\frac{d^2f}{dx^2}(0)x^2 + \cdots + \frac{1}{k!}\frac{d^kf}{dx^k}(0)x^k. \tag{42}$$

We write $\widetilde{f_k}$ to denote the truncated Taylor series f_k, with the derivative tensors in f_k replaced by their tensor train approximations.

Surrogate models based on truncation of Taylor series up to the linear term are common in Bayesian inversion, stochastic optimization, and model reduction, where they can be used directly as approximations of $\mathcal{F}(m)$, or used in Markov chain Monte Carlo proposals for Bayesian inversion as variance reduction devices [4, 5, 8, 9, 17, 27, 43, 45, 47]. Their advantage is that once constructed, no more PDEs need to be solved during the sampling or optimization process. Methods that truncate the Taylor series after the quadratic term have been investigated in [1, 11, 12, 13, 14]. In these papers, the second derivative (a third order tensor) is not compressed and stored; rather the action of this tensor is used in other ways. In [6, 7], high order derivatives for the parameter-to-solution map for the log-normal linear Darcy problem are compressed into tensor train format, and Taylor series are constructed. Their algorithm depends specifically on the linear Darcy problem with the solution map as the quantity of interest. With the methods in this paper, we now have a general-purpose algorithm for compressing and storing high order derivative tensors for nonlinear problems with arbitrary quantity of interest.

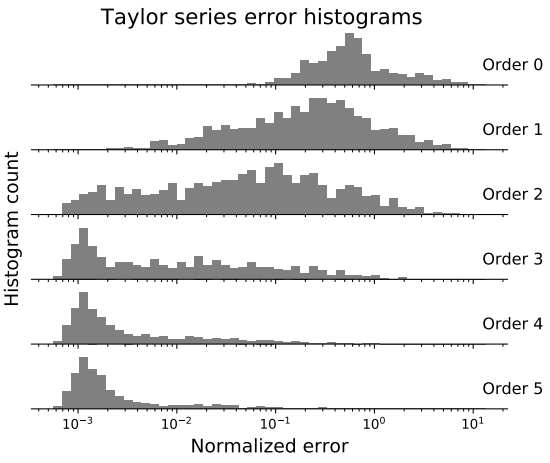
In Table 1, we report the rank required to achieve the relative error

$$\frac{\sigma_1\left(T-\widetilde{T}\right)}{\sigma_1\left(T\right)} < \epsilon,$$

where $\epsilon = 10^{-2}$ or $\epsilon = 10^{-3}$, and

$$\sigma_1(T) := \max_{\|x\|=1} \|T(x,\ldots,x,\ \cdot\ )\|. \tag{43}$$

We use σ_1 to measure the error because the form of the argument being maximized in (43) mimics the way the tensor is used within the Taylor series, (42). Computing σ_1 is NP-hard [24], so we estimate σ_1 by applying a shifted symmetric high order power



(a)

Order	Error mean	Error standard deviation
0	1.0	1.4
1	0.55	0.98
2	0.24	0.54
3	0.078	0.21
4	0.019	0.069
5	0.014	0.075

(b)

FIG. 4. Taylor series error. (a) *Error histograms for the normalized error* $\|f(x) - \widetilde{f}_k(x)\|/\mathbb{E}(\|f - f(0)\|)$*, for* 1000 *random samples* $x \sim N(0, I)$*, for Taylor series' of order* 0 *through* 5*, using a* 40×40 *mesh. Rank* 30 *is used for all derivative tensors. The high order derivative tensors (second through fifth derivatives) are approximated with our method. The Jacobian (first derivative) is approximated with randomized SVD.* (b) *The means and standard deviations of these normalized error distributions.*

method [32] to the function $x \mapsto T(x, \ldots, x, T(x, \ldots, x, \ \cdot \ ))$, with five random initial guesses.

We show results for derivative orders $k = 1$ through $k = 5$ (tensor orders $d = 2$ through $d = 6$) and meshes ranging from 10×10 to 80×80. For $k = 2$ through $k = 5$, the reported rank r is the tensor-train rank required using our method. For $k = 1$, r is the matrix rank required using randomized SVD. Our results show that the tensor train approximation is mesh-scalable for this problem. As the mesh is refined, the required tensor train rank remains roughly constant.

In Figure 4 we show histograms for the normalized error

$$\frac{\|f(x) - \widetilde{f}_k(x)\|}{\mathbb{E}(\|f - f(0)\|)}$$

for 1000 random samples $x \sim N(0, I)$ (i.e., the normalized error in the Taylor approximation of the noise-whitened parameter-to-output map, for samples drawn from the noise-whitened parameter distribution). We show Taylor series of order 0 through 5. Rank 30 approximation is used for all derivative tensors. We use a 40×40 mesh. The high order derivative tensors (second through fifth derivative tensors) are approximated with our method, while the Jacobian (first derivative) is approximated with randomized SVD. Constructing the fifth order Taylor series takes approximately 25 minutes on a laptop (HP model 17-CA1031DX).

Including high order derivatives increases the approximation accuracy. As the order of the Taylor series increases, the error in the approximation decreases. Also, as the order of the Taylor series increases, the normalized error distribution concentrates near 10^{-3}, which is roughly the error due to the rank-30 tensor train approximations of the derivative tensors.

5. Conclusion. We developed a new randomized algorithm for tensor train decomposition of tensors whose entries cannot be directly accessed or are very expensive to compute. Our method requires only the tensor's action on given vectors and uses a "peeling process" to successively compute the tensor train cores. This process requires $O(\lceil r/N \rceil dr^2)$ tensor actions for a dth order tensor with maximum rank r of the cores. We demonstrated the accuracy of this method compared to the conventional TT-SVD and TT-cross methods for a Hilbert tensor. Moreover, we applied this method to construct tensor train decompositions of PDE-constrained high order derivative tensors, for which we derived an efficient scheme to compute the action of arbitrary order derivative tensors. Our method now enables one to use Taylor series truncated to high order terms for uncertainty quantification [6, 7], Bayesian inversion [12, 14], stochastic optimization [1, 13], and model reduction [2, 11]. Furthermore, other fields offer promising potential applications of the tensor action-based tensor train decomposition, such as neural networks [18, 46] and model constrained high dimensional sampling and integration [19].

Acknowledgments. We thank Alen Alexandrian, J.J. Alger, Josh Chen, Tan Bui-Thanh, Andrew Potter, Keyi Wu, and Qiwei Zhan for helpful discussions.

REFERENCES

[1] A. Alexanderian, N. Petra, G. Stadler, and O. Ghattas, *Mean-variance risk-averse optimal control of systems governed by PDEs with random parameter fields using quadratic approximations*, SIAM/ASA J. Uncertain. Quantif., 5 (2017), pp. 1166–1192, https://doi.org/10.1137/16M106306X.

[2] C. Bach, D. Ceglia, L. Song, and F. Duddeck, *Randomized low-rank approximation methods for projection-based model order reduction of large nonlinear dynamical problems*, Internat. J. Numer. Methods Engrg., 118 (2019), pp. 209–241.

[3] J. Ballani, L. Grasedyck, and M. Kluge, *Black box approximation of tensors in hierarchical Tucker format*, Linear Algebra Appl., 438 (2013), pp. 639–657.

[4] O. Bashir, K. Willcox, O. Ghattas, B. van Bloemen Waanders, and J. Hill, *Hessian-based model reduction for large-scale systems with initial condition inputs*, Internat. J. Numer. Methods Engrg., 73 (2008), pp. 844–868.

[5] D. Bigoni, O. Zahm, A. Spantini, and Y. Marzouk, *Greedy Inference with Structure-Exploiting Lazy Maps*, preprint, https://arxiv.org/abs/1906.00031, 2019.

[6] F. Bonizzoni and F. Nobile, *Perturbation analysis for the Darcy problem with log-normal permeability*, SIAM/ASA J. Uncertain. Quantif., 2 (2014), pp. 223–244, https://doi.org/10.1137/130949415.

[7] F. Bonizzoni, F. Nobile, and D. Kressner, *Tensor Train Approximation of Moment Equations for the Log-normal Darcy Problem*, Tech. rep., Mathicse Technical Report, Lausanne, Switzerland, 2014.

[8] T. Bui-Thanh and O. Ghattas, *A scaled stochastic Newton algorithm for Markov chain Monte Carlo simulations*, unpublished, 2012.

[9] T. Bui-Thanh, O. Ghattas, J. Martin, and G. Stadler, *A computational framework for infinite-dimensional Bayesian inverse problems Part* I*: The linearized case, with application to global seismic inversion*, SIAM J. Sci. Comput., 35 (2013), pp. A2494–A2523, https://doi.org/10.1137/12089586X.

[10] M. Che and Y. Wei, *Randomized algorithms for the approximations of Tucker and the tensor train decompositions*, Adv. Comput. Math., 45 (2019), pp. 395–428.

[11] P. Chen and O. Ghattas, *Hessian-based sampling for high-dimensional model reduction*, Int. J. Uncertain. Quantif., 9 (2019), pp. 103–121.

[12] P. Chen, U. Villa, and O. Ghattas, *Hessian-based adaptive sparse quadrature for infinite-dimensional Bayesian inverse problems*, Comput. Methods Appl. Mech. Engrg., 327 (2017), pp. 147–172.

[13] P. Chen, U. Villa, and O. Ghattas, *Taylor approximation and variance reduction for PDE-constrained optimal control under uncertainty*, J. Comput. Phys., 385 (2019), pp. 163–186.

[14] P. Chen, K. Wu, J. Chen, T. O'Leary-Roseberry, and O. Ghattas, *Projected Stein Variational Newton: A Fast and Scalable Bayesian Inference Method in High Dimensions*, in

Proceedings of the 33rd Conference on Neural Information Processing Systems, 2019.
[15] A. Cichocki, N. Lee, I. Oseledets, A.-H. Phan, Q. Zhao, D. P. Mandic, et al., *Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions*, Found. Trends Mach. Learn., 9 (2016), pp. 249–429.
[16] E. Corona, A. Rahimian, and D. Zorin, *A tensor-train accelerated solver for integral equations in complex geometries*, J. Comput. Phys., 334 (2017), pp. 145–169.
[17] T. Cui, K. Law, and Y. Marzouk, *Dimension-independent likelihood-informed MCMC*, J. Comput. Phys., 304 (2016), pp. 109–137.
[18] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, *TIE: Energy-efficient tensor train-based inference engine for deep neural network*, in Proceedings of the 46th International Symposium on Computer Architecture, 2019, pp. 264–278.
[19] S. Dolgov, K. Anaya-Izquierdo, C. Fox, and R. Scheichl, *Approximation and sampling of multivariate probability distributions in the tensor train decomposition*, Stat. Comput., 30 (2020), pp. 603–625.
[20] S. V. Dolgov, B. N. Khoromskij, and I. V. Oseledets, *Fast solution of parabolic problems in the tensor train/quantized tensor train format with initial application to the Fokker–Planck equation*, SIAM J. Sci. Comput., 34 (2012), pp. A3016–A3038, https://doi.org/10.1137/120864210.
[21] P. Gelss, *The Tensor-Train Format and Its Applications: Modeling and Analysis of Chemical Reaction Networks, Catalytic Processes, Fluid Flows, and Brownian Dynamics*, Ph.D. thesis, Fachbereich Mathematik und Informatikder Freien Universität Berlin, Germany, 2017.
[22] L. Grasedyck, D. Kressner, and C. Tobler, *A literature survey of low-rank tensor approximation techniques*, GAMM-Mitt., 36 (2013), pp. 53–78.
[23] N. Halko, P. G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev. 53 (2011), pp. 217–288, https://doi.org/10.1137/090771806.
[24] C. J. Hillar and L.-H. Lim, *Most tensor problems are NP-hard*, J. ACM, 60 (2013), pp. 1–39.
[25] S. Holtz, T. Rohwedder, and R. Schneider, *The alternating linear scheme for tensor optimization in the tensor train format*, SIAM J. Sci. Comput., 34 (2012), pp. A683–A713, https://doi.org/10.1137/100818893.
[26] B. Huber, R. Schneider, and S. Wolf, *A randomized tensor train singular value decomposition*, in Compressed Sensing and Its Applications, Springer, Cham, 2017, pp. 261–290.
[27] T. Isaac, N. Petra, G. Stadler, and O. Ghattas, *Scalable and efficient algorithms for the propagation of uncertainty from data through inference to prediction for large-scale problems, with application to flow of the Antarctic ice sheet*, J. Comput. Phys., 296 (2015), pp. 348–368.
[28] V. Kazeev, M. Khammash, M. Nip, and C. Schwab, *Direct solution of the chemical master equation using quantized tensor trains*, PLoS Comput. Biol., 10 (2014), e1003359.
[29] V. Kazeev, O. Reichmann, and C. Schwab, *Low-rank tensor structure of linear diffusion operators in the TT and QTT formats*, Linear Algebra Appl., 438 (2013), pp. 4204–4221.
[30] B. N. Khoromskij, *$O(dlogN)$-quantics approximation of N-d tensors in high-dimensional numerical modeling*, Constr. Approx., 34 (2011), pp. 257–280.
[31] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Rev., 51 (2009), pp. 455–500, https://doi.org/10.1137/07070111X.
[32] T. G. Kolda and J. R. Mayo, *Shifted power method for computing tensor eigenpairs*, SIAM J. Matrix Anal. Appl., 32 (2011), pp. 1095–1124, https://doi.org/10.1137/100801482.
[33] L. Lin, J. Lu, and L. Ying, *Fast construction of hierarchical matrix representation from matrix–vector multiplication*, J. Comput. Phys., 230 (2011), pp. 4071–4087.
[34] A. Logg, K.-A. Mardal, and G. Wells, *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*, Lect. Notes Comput. Sci. Eng. 84, Springer, Cham, 2012.
[35] J. R. Maddison, D. N. Goldberg, and B. D. Goddard, *Automated calculation of higher order partial differential equation constrained derivative information*, SIAM J. Sci. Comput., 41 (2019), pp. C417–C445, https://doi.org/10.1137/18M1209465.
[36] U. Naumann, *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*, SIAM, Philadelphia, 2012, https://doi.org/10.1137/1.9781611972078.
[37] I. V. Oseledets, *Approximation of matrices with logarithmic number of parameters*, Dokl. Math., 80 (2009), pp. 653–654.
[38] I. V. Oseledets, *Approximation of $2^d \times 2^d$ matrices using tensor decomposition*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 2130–2145, https://doi.org/10.1137/090757861.
[39] I. V. Oseledets, *Tensor-train decomposition*, SIAM J. Sci. Comput., 33 (2011), pp. 2295–2317,

https://doi.org/10.1137/090752286.

[40] I. V. Oseledets and E. Tyrtyshnikov, *TT-cross approximation for multidimensional arrays*, Linear Algebra Appl., 432 (2010), pp. 70–88.

[41] I. V. Oseledets, E. Tyrtyshnikov, and N. Zamarashkin, *Tensor-train ranks for matrices and their inverses*, Comput. Methods Appl. Math., 11 (2011), pp. 394–403.

[42] D. Perez-Garcia, F. Verstraete, M. Wolf, and J. Cirac, *Matrix product state representations*, Quantum Inf. Comput., 7 (2007), pp. 401–430.

[43] N. Petra, J. Martin, G. Stadler, and O. Ghattas, *A computational framework for infinite-dimensional Bayesian inverse problems: Part* II. *Stochastic Newton MCMC with application to ice sheet inverse problems*, SIAM J. Sci. Comput., 36 (2014), pp. A1525–A1555, https://doi.org/10.1137/130934805.

[44] D. Savostyanov and I. Oseledets, *Fast adaptive interpolation of multi-dimensional arrays in tensor train format*, in The 2011 International Workshop on Multidimensional (nD) Systems, IEEE, 2011, pp. 1–8.

[45] A. Spantini, A. Solonen, T. Cui, J. Martin, L. Tenorio, and Y. Marzouk, *Optimal low-rank approximations of Bayesian linear inverse problems*, SIAM J. Sci. Comput., 37 (2015), pp. A2451–A2487, https://doi.org/10.1137/140977308.

[46] Y. Yang, D. Krompass, and V. Tresp, *Tensor-train recurrent neural networks for video classification*, in Proceedings of the 34th International Conference on Machine Learning, Volume 70, JMLR, 2017, pp. 3891–3900.

[47] H. Zhu, S. Li, S. Fomel, G. Stadler, and O. Ghattas, *A Bayesian approach to estimate uncertainty for full waveform inversion with a priori information from depth migration*, Geophysics, 81 (2016), pp. R307–R323.