University of Glasgow

Adam Smith Business School

WORKING PAPER SERIES

Value Function Iteration without the Curse of Dimensionality

Richard Dennis

Paper No. 2026-04
January 2026

Value Function Iteration without the Curse of Dimensionality[*]

Richard Dennis[†]
University of Glasgow and CAMA

First version: October, 2025
This version: January, 2026

Abstract

This paper presents a novel approach to solving dynamic programming problems using value function iteration based on the tensor train decomposition that is not subject to the curse of dimensionality. The tensor train decomposition approximates high-dimensional functions by expressing them as a series of interconnected cores, producing an approximation that separates by variables. This approach is well-suited for approximating and integrating a high-dimensional function such as a value function. I apply the method to a range of models and compare its performance against policy iteration and established sparse-grid techniques involving Smolyak and hyperbolic cross polynomials. For models with as few as four state variables, the tensor train method is shown to be faster and comparably accurate to leading sparse-grid alternatives. This paper introduces the first application of tensor trains to solve dynamic optimization problems in Economics, offering a powerful approach to solve high-dimensional macroeconomic models.

Keywords: *Value function iteration, tensor train decomposition, curse of dimensionality.*

JEL Classification: C61, C63.

[*]I would like to thank Yiannis Vailakis, John Stachurski, and Jesús Fernández-Villaverde for comments. The usual disclaimer applies.

[†]Economics, Adam Smith Business School, Adam Smith Building, 2 Discovery Place, University of Glasgow, Glasgow G11 6EY; Email: richard.dennis@glasgow.ac.uk.

1 Introduction

As macroeconomic models become larger and more realistic the difficulties encountered when solving and analyzing them grow more acute. Even relatively simple macroeconomic models with only a few few state variables can be time-consuming to solve numerically to any reasonable accuracy, particularly when some of the state variables are stochastic, introducing the need to form conditional expectations. This problem, which Richard Bellman (Bellman, 1957) dubbed the "curse of dimensionality", is a central reason why perturbation methods continue to be widely used to solve macroeconomic models. This is not to say that alternatives to perturbation are not available. Although standard collocation and Galerkin methods based on Chebyshev polynomials or other basis functions are subject to the curse of dimensionality, there is increasing recognition and use of sparse-grid methods, such as Smolyak (1963) polynomials and hyperbolic cross polynomials (Dũng et al. (2018)), introduced to Economics by Kruger and Kubler (2004) and Dennis (2024), respectively.

The main obstacles encountered when solving macroeconomic models center around the need to approximate and integrate high dimensional functions, such as decision rules or the value function in a dynamic programming problem. A tensor or multidimensional array can be viewed as a discretized function. One constructs a set of approximating points for each variable and then populates the array by evaluating the function at each point on the grid formed by the tensor product of the approximating points. Compression techniques that allow an array to be (approximately) recovered using a smaller set of points can be viewed as a form of function approximation. Depending on how the array is compressed, the approximation may lead to enormous economy regarding the number of approximation points and to rapid numerical integration.

This paper develops a method for solving macroeconomic models using value function iteration based on the tensor train decomposition of an array. Tensor trains have their origins in computational physics where they are known as Matrix Product State and are used to simulate quantum many-body systems. Rediscovered by Oseledets (2009, 2011) in the realm of computational linear algebra, there are now several established methods for undertaking tensor train decomposition (Oseledets 2011; Oseledets and Tyrtyshnikov, 2010; Savostyanov and Oseledets, 2011; Dolgov and Savostyanov, 2020) and standard algebraic operations involving tensor trains have been developed. The tensor train decomposition can be applied widely and delivers an approximation to the original array whose errors are bounded. Importantly, tensor-train approximations are often quickly obtained and algebraic operations involving tensor trains, such as addition, inner-products, and Kronnecker products, etc, and numeric operations, such as cubature, are not subject to the curse of dimensionality.

The tensor train decomposition gains its usefulness by expressing an approximated tensor through a series of interconnected carriages or cores, one core for each variable that enters the function. In effect, the tensor train decomposition performs a separation by variables. Where each core in the train takes the form of a three-dimensional array, I go a step further and develop a functional tensor train approximation whereby each core, each three-dimensional array, in the train is expressed in the form of a matrix of univariate functions, leading to a functional tensor train.[1] A functional tensor train allows us to approximate a function with another for which the variables are separated. This separation by variables allows the approximated function to be rapidly evaluated, differentiated, and integrated, while the compression step used to form the approximation disciplines the number of approximating points used.

I show how tensor trains can be brought to bear to solve dynamic programming problems using either value function iteration or policy iteration. For policy iteration I consider two schemes: one based on the Bellman equation the other on the advantage function associated with reinforcement learning. I illustrate the methods using a simple stochastic growth model that has been extended to accommodate a variable number of shocks. To underscore the advantages of using tensor trains, I also solve the model on a dense tensor-product grid using Chebyshev polynomials and on sparse grids using Smolyak polynomials and hyperbolic cross polynomials. Based on the tensor train approach, I find that value function iteration is faster than either form of policy iteration, while being equally accurate. Howard iterations did not improve the performance of the policy iteration schemes. Between the two policy iteration methods, I find that using the Bellman equation is faster than using the advantage function.

Having established that value function iteration using tensor trains performs well, I next compare it to methods involving Chebyshev polynomials on a tensor-product grid and Smolyak and hyperbolic cross polynomials on sparse grids. For the two-state version of the model,

[1]Bigoni, Engsig-Karup, and Marzouk (2016) and Gorodetsky, Karaman, and Marzouk (2019) also develop methods for constructing functional tensor trains. The former parameterize the functions in the tenor train and then solves directly for the parameters in a single step. The latter use a continuous version of the singular value decomposition to find a functional tensor train without the need to specify sampling points for each spatial variable. My approach has the advantage of being considerably more simple to implement.

all four methods work well, but the tensor train method is the slowest and arguably least accurate. For the three-state version of the model, all three sparse grid methods have accuracy equaling the solution obtained using Chebyshev polynomials on the tensor product grid, but are considerably faster. Among the three sparse-grid methods the tensor train method is faster than using Smolyak polynomials, but slower than using hyperbolic cross polynomials. However, for the four-state version of the model the tensor train method really begins to shine. Where using a tensor-product grid takes an inordinate amount of time, the tensor train approach is faster than using hyperbolic cross polynomials and considerably faster than using Smolyak polynomials, while delivering equivalent or better accuracy. In effect, for a model with just four state variables that requires quadrature over three of them, the advantages of the tensor train method are readily apparent.

Because it operates by compressing arrays, the tensor train solution method requires that the decision variables as well as the state variables be discretized, at least as an intermediate step to forming a functional tensor train. One advantage of this discretization is that the approach can be applied directly to models with discrete choice variables. In the case where the choice variables are continuous, we are interested in the extent to which discretizing the choice variables impairs accuracy. For our extended stochastic growth model, employing a finer discretization for the choice variables does improve accuracy at the margin, at the expense of slower solution times.

I am not aware of any other paper in Economics that uses tensor trains to solve dynamic stochastic optimization problems. Outside of Economics, there is a small literature that applies tensor train methods to the control of robotic systems using either dynamic programming or (stochastic) optimal control (Gorodetsky, Karaman, and Marzouk (2015); Boyko, Oseledets, and Ferrer (2021); Shetty, Lembono, Löw, and Calinon (2024); and Shetty, Xue, and Calinon (2024)). In contrast to ourselves, these papers explore continuous-time models while Shetty, Lembono, Löw, and Calinon (2024) and Shetty, Xue, and Calinon (2024) also focus on environments that are deterministic. Our analysis focuses on discrete-time stochastic models with forward-looking rational decision-makers and provides a comparison to other sparse grid approaches.

The remainder of this paper is organized as follows. In Section 2 I discuss the curse of dimensionality and two sparse-grid methods, Smolyak polynomials and hyperbolic cross polynomials, that can be used to mitigate its effects. In Section 3 I introduce reduced rank methods for approximating functions, illustrating the methods first in the context of a bivariate function and then extending the analysis to multivariate functions with an arbitrary number of variables. I describe here the principle tools needed to implement the tensor train decomposition and show how a functional tensor train can then be formed. I demonstrate the power of tensor train approximation in Section 4 by applying it to a high-dimensional function with as many as $1,000$ variables. In Section 5 I show how tensor train methods can be brought to bear to solve value function iteration and policy iteration problems. I apply the approach to an extended stochastic growth model and show that the tensor train approach works well, delivering equal accuracy together with faster solution times than other sparse-grid methods for models with as few as four state variables. Section 6 reports analogous results for a Ramsey policy problem and a multi-country international business cycle model, complementing the analysis of the extended stochastic growth model provided in Section 5. Section 7 concludes.

2 The curse of dimensionality

It is common practice to use orthogonal polynomials to approximate univariate continuous functions. The basis for this approach is the Weierstrass theorem, which states that any continuous function can be approximated to arbitrary accuracy using a polynomial. Orthogonal polynomials, such as Legendre polynomials or Chebyshev polynomials, are generally used in place of standard polynomials because, while they span the same space, they enjoy superior numerical stability properties.

In the context of multivariate functions, a natural and widely-used approach is to form the approximating polynomial from a tensor-product of univariate basis functions. Suppose that the number of spatial dimensions is denoted d and that the number of sampling points along each dimension is denoted n, then the number of sampling points needed to form the approximating polynomials involves n^d points, which can be large for moderate d even if n is relatively small and increases exponentially with d. This problem is known as the curse of dimensionality. An increasingly common approach when the curse of dimensionality bites is to use a sparse grid method.

2.1 Sparse grid interpolation

In many instances the functions that we need to interpolate are not just continuous, but also have some continuous derivatives. Functions that have bounded mixed derivatives open up the possibility of using effectively sparse grid interpolation methods. The smoothness of these functions allows them to be well-approximated while being sampled sparsely.

Suppose that $\Omega \in \Re^d$ is a compact set and that we seek to approximate $f(\mathbf{x}) \in \mathbb{W}_d^r$, $\mathbf{x} \in \Omega$, where:

$$\mathbb{W}_d^q = \left\{ f(\mathbf{x}) \to \Re, \| \frac{\partial^{\|\mathbf{i}\|_1} f(\mathbf{x})}{\partial \mathbf{x}^{\mathbf{i}}} \|_\infty < \infty, \|\mathbf{i}\|_\infty \leq q \right\}, \tag{1}$$

and $\mathbf{i}$ is a multi-index indicating the order of differentiation with respect to each variable in $\mathbf{x}$. Functions in this class have mixed-partial derivatives up to order q that exist and are bounded. Smolyak's (1963) construction, which operates well on functions in the class $\mathbb{W}_d^q$, builds on nested approximation layers and if the number of layers is denoted μ, then the number of interpolating points N_d^μ used is known to increase polynomially in d for given μ. For example, Kruger and Kubler (2004) report that the cardinalities for $\mu = 1$, 2, and 3 are:

$$N_d^1 = 1 + 2d, \tag{2}$$

$$N_d^2 = 1 + 4d + \frac{4}{2}d(d-1), \tag{3}$$

$$N_d^3 = 1 + 8d + \frac{12}{2}d(d-1) + \frac{8}{6}d(d-1)(d-2), \tag{4}$$

respectively, which are μ'th order polynomials in d.

An alternative to the Smolyak construction is the hyperbolic cross (see Dũng et al. (2018) for a survey). The hyperbolic cross approach requires $f(\mathbf{x}) \in \mathbb{W}_d^q$, but leads to grids with even lower cardinality than Smolyak's construction. Specifically, the grid for the standard hyperbolic cross[2] is constructed from the set of multi-indices, $\mathbb{S}^{d,h}$, that satisfy:

$$\mathbb{S}^{d,h} = \{\mathbf{i} = (i_1, \dots, i_d) \in \mathbb{Z}^d : \prod_{j=1}^{d} (|i_j| + 1) \leq h + 1\}, \tag{5}$$

where $h \in \{0, 1, 2, \dots\}$, with the number of candidate approximating points along each dimension given by:

$$n = 2h + 1. \tag{6}$$

[2]Dennis (2024) describes the hyperbolic cross approach, and its implementation, and develops a generalization that bridges from the standard cross to the full tensor-product grid.

In the bivariate case ($d = 2$) the grids for the layer-four Smolyak polynomial and the corresponding hyperbolic cross polynomials ($h = 8$) are shown in Figure 1. The two grids have the same number of points (17) along the main axes, but place the off-axis points in different locations and have different numbers of total points. Where Smolyak uses 65 points and tends to place the off-axis points out in the corners, the hyperbolic cross uses 57 points and groups the off-axis points around the center.

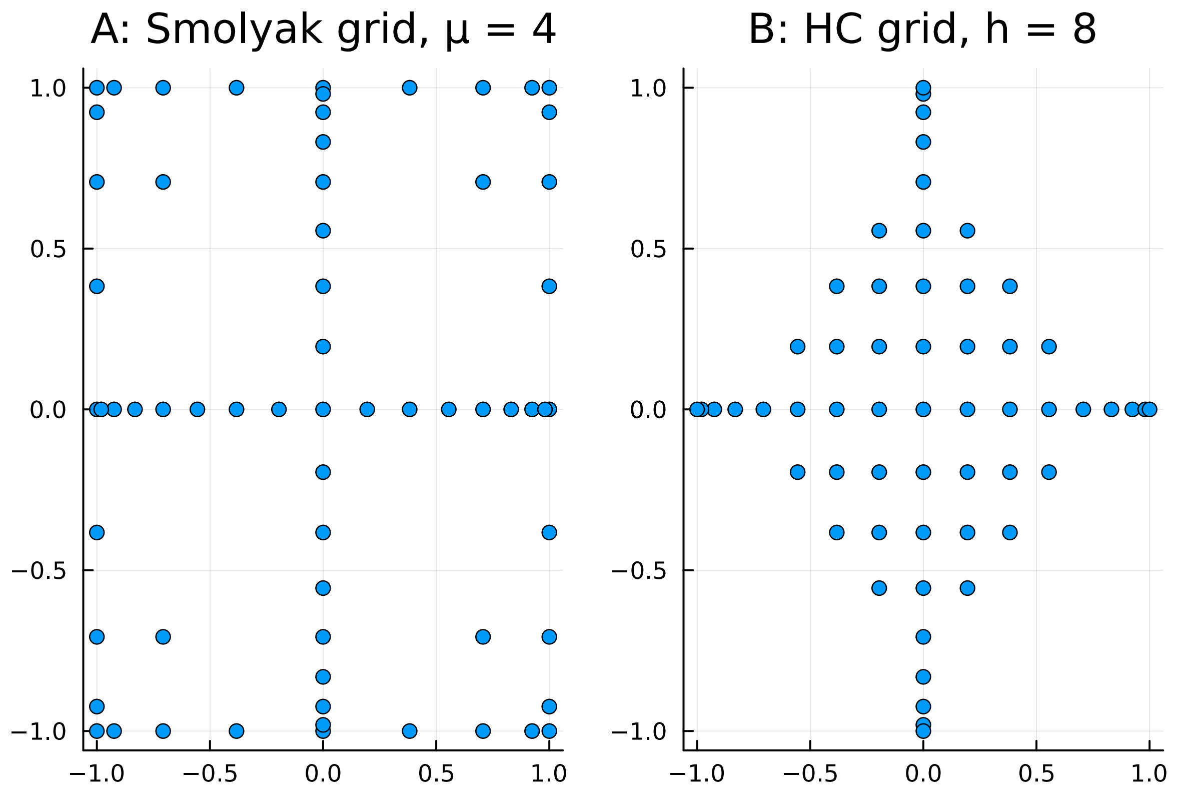

Figure 1: Smolyak and hyperbolic cross grids

Sparse grid approximation methods work well when q is high and can work poorly when applied to functions that do not have continuous derivatives.[3]

One type of function to which sparse grid approximation is well-suited are functions that separate by variables. Consider the bivariate function $f(x_1, x_2)$ and suppose that it can be

[3]For this reason sparse grid methods are generally not well-suited to models that, for example, have occasionally binding constraints, losing their theoretical advantages in such contexts.

expressed as:

$$f(x_1, x_2) = g_1(x_1)g_2(x_2). \tag{7}$$

In this case the function $f(x_1, x_2)$ is a rank-1 function that separates by variables. If $g_1(x_1)$ and $g_2(x_2)$ are both $\mathbb{W}_1^q$ functions, then $f(x_1, x_2) \in \mathbb{W}_2^q$ has many bounded mixed-partial derivatives, including some of order $2q$. Functions that separate by variables have higher smoothness, which can be exploited for interpolation. In practice it is unlikely that the function we are interpolating will completely separate by variables, but such strict separation is not needed. What is needed is that the function has bounded mixed-partial derivatives, which implies partial separation by variables facilitating the use of sparse interpolation (Novak and Ritter, 1998).

3 Reduced rank methods for function approximation

Suppose we have a multivariate function $f(\mathbf{x}) : \Omega \to \mathfrak{R}$, where $f(\mathbf{x}) \in C_d(\Omega)$ and Ω is compact. We will approximate $f(\mathbf{x})$ using a functional tensor train:

$$f(x_1, x_2, \ldots, x_d) \approx \sum_{\alpha_0=1}^{r_0} \sum_{\alpha_1=1}^{r_1} \sum_{\alpha_2=1}^{r_2} \cdots \sum_{\alpha_d=1}^{r_d} f_1^{(\alpha_0,\alpha_1)}(x_1) f_2^{(\alpha_1,\alpha_2)}(x_2) \cdots f_d^{(\alpha_{d-1},\alpha_d)}(x_d), \tag{8}$$

where $r_0 = r_d = 1$. This functional tensor train approximation can be seen to be sums of rank-1 functions, functions that separate by variables. Equation (8) can alternatively be expressed in terms of function-valued matrices:

$$f(x_1, x_2, ..., x_d) \approx \mathcal{F}_1(x_1)\mathcal{F}_2(x_2)...\mathcal{F}_d(x_d), \tag{9}$$

where:

$$\mathcal{F}_k(x_k) = \begin{bmatrix} f_k^{(1,1)}(x_k) & \cdots & f_k^{(1,r_k)}(x_k) \\ \vdots & & \vdots \\ f_k^{(r_{k-1},1)}(x_k) & \cdots & f_k^{(r_{k-1},r_k)}(x_k) \end{bmatrix}, \tag{10}$$

$k \in [1, \ldots, d]$, in which case evaluating the approximation at any point $\mathbf{x} \in \Omega$ simply requires matrix multiplication.

To construct a functional tensor train, we must determine the tensor train ranks, $r_1, ...r_{d-1}$ and all the univariate functions $f_k^{(\alpha_j,\alpha_l)}(x_k)$, where $k \in [1, \ldots, d]$, $j \in [0, \ldots, d-1]$, and $l \in [1, \ldots, d]$. We will construct a functional tensor train using a tensor train as an intermediate step, and this will require sampling points for each of the d variables, $x_k \in \{x_k^1, ...x_k^{n_k}\}$, where

$k \in [1, \ldots, d]$. Given these sampling points, we can in principle form the d-dimensional array, or tensor, $\mathbf{A}$, by evaluating $f(\mathbf{x})$ at each point on the tensor-product grid formed from the univariate sampling points.

The tensor train decomposition of $\mathbf{A}$ has the form:

$$\mathbf{A}[i_1, i_2, \cdots, i_d] \approx \sum_{\alpha_0=1}^{r_0} \sum_{\alpha_1=1}^{r_1} \cdots \sum_{\alpha_d=1}^{r_d} \mathbf{G}_1[\alpha_0, i_1, \alpha_1]\mathbf{G}_2[\alpha_1, i_2, \alpha_2] \cdots \mathbf{G}_d[\alpha_{d-1}, i_d, \alpha_d], \tag{11}$$

where the $\mathbf{G}_k[\alpha_{k-1}, i_k, \alpha_k]$, $k \in [1, \ldots, d]$, are the cores in the tensor train, also called tensor train carriages. Each carriage, or core, is a three dimensional array, and the k'th core has dimensions $r_{k-1} \times n_k \times r_k$.

Suppose, for simplicity, that $i_k \in [1, \ldots, n]$ for all $k \in [1, ..., d]$ and that $\alpha_k \in [1, ..., r]$, for all $k \in [1, ..., d]$, so there are n sampling points for each variable and the tensor ranks all equal r. Then the tensor, $\mathbf{A}$, which contains $N = n^d$ elements, is approximated by a tensor train consisting of $2nr + (d-2)nr^2 \leq dnr^2$ elements ($d \geq 2$). If r is small, then the compression can be large and the savings can be huge. Moreover, strikingly, the total number of elements in the tensor train, while increasing quadratically in r, increases linearly in d—essentially there is no curse of dimensionality.

3.1 The bivariate case

To illustrate how a functional tensor train approximation can be formed it is useful to first focus on the bivariate case. For a continuous bivariate function, $f(x_1, x_2)$, the functional tensor train approximation has the form:

$$f(x_1, x_2) \approx \sum_{\alpha_0=1}^{r_0} \sum_{\alpha_1=1}^{r_1} \sum_{\alpha_2=1}^{r_2} f_1^{(\alpha_0,\alpha_1)}(x_1) f_2^{(\alpha_1,\alpha_2)}(x_2), \tag{12}$$

where $r_0 = r_2 = 1$ and r_1 is ideally small.

Given vectors of sampling points $\mathbf{x}_1$ and $\mathbf{x}_2$, of length m and n, respectively, we can, in principle, apply $f(x_1, x_2)$ to each point on the sampling grid to form the $m \times n$ matrix, $\mathbf{A}$.

3.1.1 Truncated singular value decomposition

Any matrix $\mathbf{A} \in \mathfrak{R}^{m \times n}$ has a Singular Value Decomposition (SVD):

$$\mathbf{A} = \mathbf{USV}^T, \tag{13}$$

where $\mathbf{U} \in \Re^{m\times m}$ and $\mathbf{V} \in \Re^{n\times m}$ are orthonormal matrices and $\mathbf{S}$ is a diagonal matrix whose entries are the singular values, $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_s \geq 0$, where $s = min(m,n)$. If $\mathbf{A}$ is a rank-r matrix, $r < s$, then there will be only r non-zero singular values and $\mathbf{A}$ can be interpolated by:

$$\mathbf{A} = \tilde{\mathbf{U}}\tilde{\mathbf{S}}\tilde{\mathbf{V}}^T, \tag{14}$$

where $\tilde{\mathbf{U}}$ contains the first r columns of $\mathbf{U}$, $\tilde{\mathbf{S}}$ contains the first r singular values of $\mathbf{S}$, and $\tilde{\mathbf{V}}$ contains the first r columns of $\mathbf{V}$.

A full-rank matrix may, however, have some singular values that are "small", close to zero. The truncated SVD allows a matrix to be approximated to arbitrary accuracy by retaining the columns of $\mathbf{U}$ and $\mathbf{V}$ associated with the singular values that are greater than some threshold $\epsilon > 0$. The truncated SVD gives the approximation:

$$\mathbf{A} = \tilde{\mathbf{U}}\tilde{\mathbf{S}}\tilde{\mathbf{V}}^T + \mathbf{E}, \tag{15}$$

where $\tilde{\mathbf{U}}$ and $\tilde{\mathbf{V}}$ are orthonormal and the diagonal elements of $\tilde{\mathbf{S}}$ are the retained singular values, $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r \geq \epsilon$, where $r \leq s$, and $\mathbf{E}$ is the interpolation error. Using the Frobenius norm, the error introduced through the truncation is given by:

$$\begin{aligned} ||\mathbf{E}||_F &= \sqrt{\sum_{i=r+1}^{s} \sigma_i^2}, & (16) \\ &\leq \sqrt{(s-r)\epsilon^2}, & (17) \end{aligned}$$

which is governed by ϵ. Usefully, then, the truncated SVD facilitates matrix compression while allowing the magnitude of any accuracy loss to be controlled.

Three important aspects of the SVD are first that it always exists, second that it is numerically stable, and third that it is rank-revealing. The number of non-zero singular values is equal to the rank of $\mathbf{A}$. A drawback of the SVD is that it requires the entire matrix, $\mathbf{A}$, to be constructed and stored, which can be time-consuming and memory-demanding if $\mathbf{A}$ is large.

3.1.2 Skeleton decomposition/approximation

Suppose that $\mathbf{A}$ is an $m \times n$ matrix with rank r. Without loss of generality, $\mathbf{A}$ can be block-partitioned according to:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \tag{18}$$

where $\mathbf{A}_{11}$ is $r \times r$ and has full rank. By construction, it must be the case that:

$$\mathbf{A}_{22} = \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}. \tag{19}$$

It follows that $\mathbf{A}$ can be interpolated by the skeleton decomposition:

$$\mathbf{A} = \mathbf{C}\hat{\mathbf{A}}^{-1}\mathbf{R}, \tag{20}$$

where $\mathbf{C} = \mathbf{A}[:, \mathbb{J}]$ is a matrix containing r columns of $\mathbf{A}$, where the column-indices are contained in $\mathbb{J}$, $\mathbf{R} = \mathbf{A}[\mathbb{I}, :]$ is a matrix containing r rows of $\mathbf{A}$, where the row-indices are contained in $\mathbb{I}$, and $\hat{\mathbf{A}} = \mathbf{A}[\mathbb{I}, \mathbb{J}]$. Thus, provided r is known, we can interpolate $\mathbf{A}$ by computing the $m \times r$ matrix $\mathbf{C}$ and the $r \times n$ matrix $\mathbf{R}$ (the intersection of $\mathbf{C}$ and $\mathbf{R}$ determines $\hat{\mathbf{A}}$). Moreover, any r columns of $\mathbf{A}$ can be used to construct $\mathbf{C}$ and any r rows of $\mathbf{A}$ can be used to construct $\mathbf{R}$, provided the implied $\hat{\mathbf{A}}$ matrix has rank r.

In the case that $\mathbf{A}$ actually has rank s, but the assumed rank in the skeleton approximation is r ($r < s$), Goreinov and Tyrtyshnikov (2001) proved using the Chebyshev norm that:

$$||\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}||_C \leq (r+1)\sigma_{r+1}(\mathbf{A}), \tag{21}$$

where $\sigma_{r+1}(\mathbf{A})$ is the $r+1$'th singular value for the matrix $\mathbf{A}$, provided that $\mathbf{A}_{11}$ has the largest volume[4] of any of $\mathbf{A}$'s $r \times r$ submatrices. It follows that the rank-r skeleton approximation:

$$\mathbf{A} = \mathbf{C}\hat{\mathbf{A}}^{-1}\mathbf{R} + \mathbf{E}, \tag{22}$$

produces an interpolation error, $||\mathbf{E}||_C$, whose magnitude is affected by the choice of $\hat{\mathbf{A}}$. Accordingly, the precise rows, $\mathbb{I}$, and columns, $\mathbb{J}$, used to construct $\mathbf{R}$ and $\mathbf{C}$, respectively, matter for the interpolation error. However, in the Chebyshev norm, the magnitude of the interpolation error is smallest when the volume of $\hat{\mathbf{A}}$ is largest.

The advantage of the skeleton approximation is that it can be computed without forming the entire $\mathbf{A}$ matrix (one just needs r rows and r columns of $\mathbf{A}$). Disadvantages are that the approximation requires that the value for r be provided. The skeleton approximation is not rank-revealing and, for accuracy, it requires finding the r rows and r columns of $\mathbf{A}$ that produce the $\hat{\mathbf{A}}$ submatrix with largest volume.

[4]The volume of a matrix is defined to equal the absolute value of its determinant.

3.1.3 A functional approximation

Suppose we have an $m \times n$ matrix $\mathbf{A}$ with approximate rank r, i.e., there are r singular values greater than or equal to the chosen tolerance, ϵ. For the matrix $\mathbf{A}$, we can construct the truncated singular value decomposition:

$$\begin{aligned} \mathbf{A} &\approx \tilde{\mathbf{U}}\tilde{\mathbf{S}}\tilde{\mathbf{V}}^T, && (23)\\ &\approx \mathbf{G}_1\mathbf{G}_2. && (24) \end{aligned}$$

where $\mathbf{G}_1 = \tilde{\mathbf{U}}$ is an $m \times r$ array and $\mathbf{G}_2 = \tilde{\mathbf{S}}\tilde{\mathbf{V}}^T$ is an $r \times n$ array. The matrices $\mathbf{G}_1$ and $\mathbf{G}_2$ are the cores in a discrete tensor-train approximation of $\mathbf{A}$, an approximation requiring $(m+n)r$ values where $\mathbf{A}$ contains mn elements. Provided r is small, the tensor train provides a compression that allows $\mathbf{A}$ to be accurately reconstructed using vastly fewer points.

Alternatively, given r and the function to produce each element in $\mathbf{A}$, we can construct the skeleton approximation:

$$\begin{aligned} \mathbf{A} &\approx \mathbf{C}\hat{\mathbf{A}}^{-1}\mathbf{R}, && (25)\\ &\approx \tilde{\mathbf{G}}_1\tilde{\mathbf{G}}_2, && (26) \end{aligned}$$

where the cores in the tensor train are now given by $\tilde{\mathbf{G}}_1 = \mathbf{C}\hat{\mathbf{A}}^{-1}$ and $\tilde{\mathbf{G}}_2 = \mathbf{R}$ and whose construction requires $(m+n)r$ values.[5]

The final step is to construct from the tensor train a functional tensor train. Consider either $\mathbf{G}_1$ or $\tilde{\mathbf{G}}_1$. Each of these matrices has dimensions $m \times r$, where m is the number of sampling points for the first spatial dimension and r is the approximated/given rank. For each column of the first core the sampling points are known and ordered from 1 to m. This allows us to use the values in each column of the first core to construct an interpolating function. The exact interpolating function used will depend on the precise sampling points used. If, for example, the sampling points are the Chebyshev extrema or the Chebyshev roots, then each column of $\mathbf{G}_1$ or $\tilde{\mathbf{G}}_1$ can be interpolated using a univariate Chebyshev polynomial with order $m-1$. An analogous procedure can be used to construct univariate polynomials that

[5]The cores in a tensor train are not unique, so there is no reason to expect $\mathbf{G}_1 \approx \tilde{\mathbf{G}}_1$, for example.

interpolate the rows of $\mathbf{G}_2$ or $\tilde{\mathbf{G}}_2$. The result is an approximation:

$$f(x_1, x_2) \approx \begin{bmatrix} f_1^{(1,1)}(x_1) & \cdots & f_1^{(1,r)}(x_1) \end{bmatrix} \begin{bmatrix} f_2^{(1,1)}(x_2) \\ \vdots \\ f_2^{(r,1)}(x_2) \end{bmatrix}, \tag{27}$$

$$\approx \sum_{\alpha_0=1}^{1} \sum_{\alpha_1=1}^{r} \sum_{\alpha_2=1}^{1} f_1^{(\alpha_0,\alpha_1)}(x_1) f_2^{(\alpha_1,\alpha_2)}(x_2), \tag{28}$$

where the approximating function is a functional tensor train.

3.2 The d-dimensional case

The previous section showed how to construct a tensor train to approximate a matrix, and how that tensor train could then be used to produce a functional tensor train. The success of the approach rested on the generality of the SVD and the skeleton approximation. We now return to the d-variable case where we are interested in approximating a continuous multivariate function $f(\mathbf{x}) : \Omega \rightarrow \Re$, where $f(\mathbf{x}) \in C_d(\Omega)$ by a functional tensor train in the form of equation (8) or equation (9). As stated previously, we will first construct a tensor train based on sampling points for each spatial dimension and then we will use univariate interpolating functions to construct a functional tensor train from the tensor train.

Creating a functional tensor train from a tensor train is relatively straightforward, even if it involves many univariate functions and many variables, so we will focus primarily on how to undertake the tensor train approximation, how to form the tensor train decomposition. To fix ideas we will begin with the case where (conceptually at least) the full d-dimensional array has been constructed using the function to be approximated, which allows the tensor train to be constructed through successive applications of the truncated singular value decomposition (TT-SVD), developed in Oseledets and Tyrtyshnikov (2010) and Oseledets (2011). Subsequently, we will look at the case where the full d-dimensional array is not needed, with the tensor train constructed by combining the rank-revealing SVD decomposition with aspects of the skeleton approximation (DMRG cross). The DMRG cross algorithm was developed in the context of numerical linear algebra by Savostyanov and Oseledets (2011), however the presentation below draws on Dolgov and Savostyanov (2020).

3.2.1 TT-SVD

The k-th unfolding matrix of the d-dimensional array $\mathbf{A}$ is given by:

$$\mathbf{A}_k = [A[i_1 \cdots i_k, i_{k+1} \cdots i_d]], \tag{29}$$

where $i_k \in \mathbb{I}_k = \{1, \ldots, n_k\}$. Each index $i_1 \cdots i_k \in \mathbb{I}_1 \times \cdots \times \mathbb{I}_k$ points to a row of $\mathbf{A}_k$ while each index $i_{k+1} \cdots i_d \in \mathbb{I}_{k+1} \times \cdots \times \mathbb{I}_d$ points to a column. These unfolding matrices can be constructed by simply reshaping $\mathbf{A}$ appropriately. Let's suppose for now that:

$$rank(\mathbf{A}_k) = r_k, \tag{30}$$

for $k = 1, \ldots d-1$.

From the SVD, $\mathbf{A}_1$ can be interpolated by:

$$\mathbf{A}_1 = \mathbf{U}_1 \mathbf{S}_1 \mathbf{V}_1^T, \tag{31}$$

where $\mathbf{U}_1$ has dimensions $n_1 \times r_1$ and $\mathbf{M}_1 = \mathbf{S}_1 \mathbf{V}_1^T$ has dimensions $r_1 \times n_2 \cdots n_d$. The matrix $\mathbf{U}_1$ can be trivially reshaped into a three dimensional array with dimensions $1 \times n_1 \times r_1$ and forms the first core in the tensor train decomposition, $\mathbf{G}_1$.

We can now reshape $\mathbf{M}_1$ to have dimensions $r_1 n_2 \times n_3 \cdots n_d$ and apply the SVD to interpolate $\mathbf{M}_1$:

$$\mathbf{M}_1 = \mathbf{U}_2 \mathbf{S}_2 \mathbf{V}_2^T, \tag{32}$$

where $\mathbf{U}_2$ has dimensions $r_1 n_2 \times r_2$, which can be reshaped into a 3-dimensional array with dimensions $r_1 \times n_2 \times r_2$ and forms the second core in the tensor train, $\mathbf{G}_2$. We can further construct $\mathbf{M}_2 = \mathbf{S}_2 \mathbf{V}_2^T$ and reshape into a matrix with dimensions $r_2 n_3 \times n_4 \cdots n_d$. Continuing in this fashion, applying the SVD successively, each of the d cores in the d-dimensional tensor train decomposition can be constructed. The final core, $\mathbf{G}_d$, is formed by reshaping $\mathbf{S}_d \mathbf{V}_d^T$ into an three dimensional array of size $r_{d-1} \times n_d \times 1$. Modeled on Oseledets (2011), a numerical algorithm to construct the tensor train decomposition for an array, $\mathbf{A}$, using the truncated SVD is given in Algorithm 1.

If the unfolding matrices are not only singular, but their ranks r_k are known, then the tensor train decomposition can be formed through an analogous procedure employing the skeleton decomposition.

Algorithm 1 TTsvd

```
1: Inputs: $\mathbf{A}$ and the accuracy tolerance, $\epsilon$.
2: $\delta \leftarrow \frac{\epsilon}{\sqrt{d-1}}||\mathbf{A}||_F$.
3: $\mathbf{M} \leftarrow \mathbf{A}$.
4: $r_0 \leftarrow 1$.
5: for $k \leftarrow 1$ to $d-1$ do
6:     $\mathbf{M} \leftarrow reshape(\mathbf{M}, r_{k-1}n_k, \frac{cardinality(\mathbf{M})}{r_{k-1}n_k})$.
7:     $\mathbf{USV}^T, r_k \leftarrow tSVD(\mathbf{M}, \delta)$.
8:     $\mathbf{G}_k \leftarrow reshape(\mathbf{U}, r_{k-1}, n_k, r_k)$.
9:     $\mathbf{M} \leftarrow \mathbf{SV}^T$.
10: end for
11: $r_d \leftarrow 1$.
12: $\mathbf{G}_d \leftarrow reshape(\mathbf{M}, r_{d-1}, n_d, r_d)$.
13: return $\{\mathbf{G}_k\}_{k=1}^d$, $\{r_i\}_{i=0}^d$.
```

Oseledets (2011, theorem 2.1) establishes that if each unfolding matrix $\mathbf{A}_k$ of a d-dimensional array has:

$$rank(\mathbf{A}_k) = r_k, \tag{33}$$

for $k = 1, \ldots, d-1$, then there exists a tensor train decomposition whose ranks are not higher than r_k. This result makes clear that if the unfolding matrices are singular so that the ranks are revealed by the SVD, then the ranks in the tensor train decomposition will not be inefficiently high. This is important because excessively sized cores in a tensor train add unnecessarily to the train's storage requirements and they slow computation times.

If, alternatively, the unfolding matrices are not singular, but contain singular values that are "small", then the truncated SVD can be applied to find the approximating tensor train, at the cost of some interpolation error.

Oseledets and Tyrtyshnikov (2010, theorem 2.2) prove that for any tensor $\mathbf{A}$ there exists a tensor train approximation $\mathbf{T}$ with ranks $\{r_k\}_{k=1}^{d-1}$ such that:

$$||\mathbf{A} - \mathbf{T}||_F \leq \sqrt{\sum_{k=1}^{d} \epsilon_k^2} \tag{34}$$

where ϵ_k is the distance in the Frobenius norm from $\mathbf{A}_k$ to its best rank-r_k approximation, $\mathbf{B}$,

$$\epsilon_k = \underset{rank(\mathbf{B}) \leq r_k}{min} ||\mathbf{A}_k - \mathbf{B}||_F. \tag{35}$$

This theorem speaks to the fact that the tensor train approximation has an interpolation error that is bounded, and related to the sum of the interpolation errors associated with approximating each of the unfolding matrices. Importantly, the interpolation error for each of the unfolding matrices can be controlled through the tolerance, ϵ, used to truncate the SVD.

The TTsvd algorithm can be a very useful technique for compressing tensors. However, it requires the full array, $\mathbf{A}$, and, for this reason, it does not circumvent the curse of dimensionality, even if the resulting compressed array and the resulting approximated function are fast to work with.

3.2.2 DMRG cross

In this section we show how a tensor train approximation can be formed without having the full array, $\mathbf{A}$, by using the function to be approximated, $f(\mathbf{x})$, to populate only those parts of the array that are needed. Because the approach does not require the full d-dimensional array, only small parts of it, it can circumvent the curse of dimensionality. The basic tool we will use is the DMRG cross, which combines ideas from both the truncated SVD decomposition and the skeleton decomposition. From the truncated SVD decomposition we get rank-revelation and from the skeleton approximation we get the ability to approximate without forming the entire array.[6]

Let the index sets $\mathcal{J}_k$ and $\mathcal{I}_k$ be given by:

$$\mathcal{J}_k = \{i_{k+1}^{\alpha_k} \dots i_d^{\alpha_k}\}_{\alpha_k=1}^{r_k}, \tag{36}$$

$$\mathcal{I}_k = \{i_1^{\alpha_k} \dots i_k^{\alpha_k}\}_{\alpha_k=1}^{r_k}, \tag{37}$$

then $\mathbf{A}_k[\mathcal{I}_k, \mathcal{J}_k]$ refers to an $r_k \times r_k$ submatrix of the k'th unfolding matrix. With the skeleton approximation used to approximate each successive unfolding matrix, Dolgov and Savostyanov (2020) show that the array $\mathbf{A}$ can be approximated by the tensor train:

$$\mathbf{A}[i_1, \dots, i_d] \approx \mathbf{A}[i_1, \mathcal{J}_1]\mathbf{A}[\mathcal{I}_1, \mathcal{J}_1]^{-1}\mathbf{A}[\mathcal{I}_1, i_2, \mathcal{J}_2]\mathbf{A}[\mathcal{I}_2, \mathcal{J}_2]^{-1} \cdots \mathbf{A}[\mathcal{I}_{d-1}, i_d], \tag{38}$$

where the tensor train cores are given by:

$$\mathbf{G}_1 = \mathbf{A}[i_1, \mathcal{J}_1]\mathbf{A}[\mathcal{I}_1, \mathcal{J}_1]^{-1}, \tag{39}$$

$$\mathbf{G}_k = \mathbf{A}[\mathcal{I}_{k-1}, i_k, \mathcal{J}_k]\mathbf{A}[\mathcal{I}_k, \mathcal{J}_k]^{-1}, \ k = 2, \dots, d-1, \tag{40}$$

$$\mathbf{G}_d = \mathbf{A}[\mathcal{I}_{d-1}, i_d]. \tag{41}$$

[6]An alternative to DMRG cross is Alternating Least Squares (ALS) (Oseledets and Tyrtyshnikov, 2010). Because the skeleton approximation and is not rank revealing, ALS begins with an over-estimate of the tensor train ranks and then employs a QR decomposition to reveal the rank of the matrix and then finds the maximum volume of the $\mathbf{Q}$ matrix. ALS is generally less efficient than the DMRG cross method.

Constructing the tensor train for a given array, $\mathbf{A}$, therefore, boils down to finding the ranks r_k and the index sets $\mathcal{J}_k$ and $\mathcal{I}_k$, for $k = 1, \ldots, d-1$. These ranks and index-sets are identified by the DMRG cross algorithm.

For given $\mathcal{I}_{k-1}$ and $\mathcal{J}_{k+1}$, consider the following 4-dimensional array $\mathbf{A}[\mathcal{I}_{k-1}, i_k, i_{k+1}, \mathcal{J}_{k+1}]$, which can be reshaped into a matrix $\mathbf{B}_k$ that has size $r_{k-1}n_k \times n_{k+1}r_{r+1}$. Now perform the truncated singular value decomposition: $\mathbf{B}_k \approx \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^T$. This decomposition reveals the approximate rank r_k. We now find the r_k indices $\mathbb{I}_k^*$ and $\mathbb{J}_k^*$ that maximize the volumes of $\mathbf{U}_k$ and $\mathbf{V}_k$, respectively. From $\mathbb{I}_k^*$ and $\mathcal{I}_{k-1}$ we can construct the nested set $\mathcal{I}_k$. Similarly, from $\mathbb{J}_k^*$ and $\mathcal{J}_{k+1}$ we can construct the set $\mathcal{J}_k$. I implement these ideas through the DMRGcross algorithm summarized below (c.f. Savostyanov and Oseledets (2011); Dolgov and Savostyanov, (2020)):

Algorithm 2 DMRGcross

```
1: Inputs $\{\mathcal{I}_k, \mathcal{J}_k, r_k\}_{k=1}^{d-1}$, $f(\mathbf{x})$, and the accuracy tolerance, $\epsilon$.
2: sweeps $\leftarrow 0$.
3: **while** The index sets $\{\mathcal{I}_k, \mathcal{J}_k\}_{k=1}^{d-1}$ have not converged **do**
4:     **for** $k \leftarrow 1$ to $d-1$ **do** (Sweep left-to-right)
5:         Use $f(\mathbf{x})$ to populate the 4-dimensional array $\mathbf{A}[\mathcal{I}_{k-1}, i_k, i_{k+1}, \mathcal{J}_{k+1}]$.
6:         Construct $\mathbf{B}_k \leftarrow \mathbf{A}[\mathcal{I}_{k-1}, i_k, i_{k+1}, \mathcal{J}_{k+1}]$ with dimensions $r_{k-1}n_k \times n_{k+1}r_{k+1}$.
7:         $\delta \leftarrow \frac{\epsilon}{\sqrt{d-1}}||\mathbf{B}_k||_F$.
8:         $\mathbf{USV}^T, r_k \leftarrow tSVD(\mathbf{B}_k, \delta)$.
9:         $\mathbb{I}_k^* \leftarrow maxvol(\mathbf{U})$.
10:        $\mathbb{J}_k^* \leftarrow maxvol(\mathbf{V})$.
11:        $\mathcal{I}_k \leftarrow (\mathbb{I}_k^*, \mathcal{I}_{k-1})$.
12:        $\mathcal{J}_k \leftarrow (\mathbb{J}_k^*, \mathcal{J}_{k+1})$.
13:    **end for**
14:    Sweep right-to-left through the cores to recover nestedness of $\{\mathcal{J}_k\}_{k=1}^{d-1}$.
15:    sweeps += 1.
16: **end while**
17: return $\{\mathcal{I}_k, \mathcal{J}_k, r_k\}_{k=1}^{d-1}$.
```

A few comments about the DMRG cross algorithm, Algorithm 2, are in order. First, the algorithm essentially sweeps from left to right through the cores of the tensor train, constructing the indices for the k'th core while conditioning upon the remaining cores. Second, the entire array $\mathbf{A}$ is not required to construct the four dimensional array in step five. Instead, $f(\mathbf{x})$ (the function being approximated) is used to populate only that part of the array that is being worked on. Third, once the ranks and indices $\{\mathcal{I}_k, \mathcal{J}_k, r_k\}_{k=1}^{d-1}$ have been determined, it is straightforward to construct each core of the tensor train. Fourth, as the algorithm sweeps left to right through the cores nestedness of the row indices, $\mathcal{I}_k \subset \mathcal{I}_{k-1} \times \mathbb{I}_k$, is ensured by construction, but nestedness of the column indices $\mathcal{J}_k$ is not. However, nestedness of the column indices can be recovered at the end of each sweep. Fifth, although the algorithm typically converges in relatively few sweeps—with convergence assisted by the use of submatrices that have maximum volume—it is not guaranteed to converge.

We have the following two results from Dolgov and Savostyanov (2020). First, if $rank(\mathbf{A}_k) = r_k$, for $k = 1, \ldots, d-1$, and the submatrices $\mathbf{A}[\mathcal{I}_k, \mathcal{J}_k]$, are full rank, then equation (38) interpolates $\mathbf{A}$. This result shows that if the array, $\mathbf{A}$, being approximated has singular unfolding matrices, then the tensor train interpolates $\mathbf{A}$. In other words, $\mathbf{A}$ can be compressed into a tensor train containing far fewer elements without any loss in accuracy.

Second, for $k = 1, \ldots, d-1$, if the indices $\{\mathcal{I}_k, \mathcal{J}_k\}_{k=1}^{d-1}$ are nested: $\mathcal{I}_k \subset \mathcal{I}_{k-1} \times \mathbb{I}_k$ and $\mathcal{J}_k \subset \mathcal{J}_{k-1} \times \mathbb{J}_k$, then equation (38) interpolates the elements in $\mathbf{A}[\mathcal{I}_{k-1}, i_k, \mathcal{J}_k]$. Because the array $\mathbf{A}[\mathcal{I}_{k-1}, i_k, \mathcal{J}_k]$ is a key term needed to construct the k'th core in the tensor train, this results tells us that if the index sets are nested, then the tensor train approximation should provide a more accurate approximation of $\mathbf{A}$, and, therefore, of the function that populates $\mathbf{A}$.

3.2.3 A functional tensor train

The tensor train approximation of an array takes the form of equation (38), or, more generally, of equation (11). At the same time, the cores in a tensor train, which are three dimensional arrays, can also be expressed as indexed matrices, as follows:

$$\mathbf{A}[i_1, \ldots, i_d] \approx \mathbf{G}_1[i_1]\mathbf{G}_2[i_2] \cdots \mathbf{G}_d[i_d], \tag{42}$$

emphasizing that the structure of the tensor train isolates the indexes for each of the d variables. For a given index, $i_1, \ldots, i_d$, the tensor train approximation at that index can be formed by simply evaluating a matrix product. Alternatively, notice that for each core $\mathbf{G}_k[\alpha_{k-1}, i_k, \alpha_k]$, $k = 1, \ldots, d$, once we condition on particular $\alpha_{k-1} \in [1, \ldots, r_{k-1}]$ and $\alpha_k \in [1, \ldots, r_k]$, describes a vector with length n_k whose elements are associated with the nodes along the k'th spatial dimension. This vector can be interpolated by a univariate function. Computing the univariate interpolating functions for each of the cores leads to a functional tensor train in the form of equation (8) or equation (9).

Many choices for the type of univariate interpolating function are possible. One could use a Legendre polynomial, a Chebyshev polynomial, splines, or a piecewise linear function, to name a few. What is required, however, is that the sampling points used in the DMRG cross algorithm are consistent with the sampling points needed to construct the univariate interpolating function. Thus, if one plans to use Legendre polynomials to form the functional tensor train, then one should use Legendre points to sample the function in the DMRG cross algorithm.

It is also worth noting that the interpolation that takes place to build a functional tensor train occurs core-by-core. This means that some cores can be interpolated and others not, allowing the approach to accommodate functions that depend on both continuous and discrete variables.

3.3 Cubature

Once a tensor train or functional tensor train has been formed, numerically integrating that function with respect to one, many, or all variables is straightforward and remarkably fast. Take, for example, an array, $\mathbf{A}$ whose entries are populated by the function $f(\mathbf{x})$:

$$\mathbf{A}[i_1, \dots, i_d] = f(\mathbf{x}_1[i_1], \mathbf{x}_2[i_2], \dots, \mathbf{x}_d[i_d]). \tag{43}$$

Suppose we wish to integrate $f(\mathbf{x})$ using the tensor product of one-dimensional Gauss-Legendre quadrature rules[7], with the quadrature weights for each variable given by ω_k, $k = 1, \dots, d$. Gauss-Legendre quadrature combined with a tensor train approximation of $\mathbf{A}$, implies:

$$\int \dots \int f(\mathbf{x}) d\mathbf{x} \approx \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \cdots \sum_{i_d=1}^{n_d} \mathbf{G}_1[i_1]\mathbf{G}_2[i_2] \cdots \mathbf{G}_d[i_d]\omega_1[i_1]\omega_2[i_2] \cdots \omega_d[i_d], \tag{44}$$

which, due to the separation of indexes, can be written as:

$$\int \dots \int f(\mathbf{x}) d\mathbf{x} \approx \sum_{i_1=1}^{n_1} \mathbf{G}_1[i_1]\omega_1[i_1] \sum_{i_2=1}^{n_2} \mathbf{G}_2[i_2]\omega_2[i_2] \cdots \sum_{i_d=1}^{n_d} \mathbf{G}_d[i_d]\omega_d[i_d]. \tag{45}$$

Numerically integrating over all d variables, therefore, only requires d summations followed by evaluating the product of d matrices.

[7]By introducing the appropriate weighting function, one can easily adapt the presentation given here to implement Gauss-Chebyshev quadrature.

Alternatively, if $f(\mathbf{x})$ is approximated by a functional tensor train, then employing Gauss-Legendre quadrature we have:

$$\begin{aligned}
\int \cdots \int f(\mathbf{x}) d\mathbf{x} &\approx \int \cdots \int \mathcal{F}_1(x_1)\mathcal{F}_2(x_2)...\mathcal{F}_d(x_d) dx_1 dx_2 \cdots dx_d, && (46)\\
&\approx \int \mathcal{F}_1(x_1)dx_1 \int \mathcal{F}_2(x_2)dx_2 \cdots \int \mathcal{F}_d(x_d)dx_d, && (47)\\
&\approx \sum_{i_1=1}^{n_1} \mathcal{F}_1(x_1[i_1])\omega_1[i_1] \sum_{i_2=1}^{n_2} \mathcal{F}_2(x_2[i_2])\omega_2[i_2] \cdots \sum_{i_d=1}^{n_d} \mathcal{F}_d(x_d[i_d])\omega_d[i_d], && (48)
\end{aligned}$$

which, again, requires only d summations followed by evaluating the product of d matrices. The efficiency with which cubature can be performed stems entirely from the separation of indexes generated by the (functional) tensor train approximation.

Computing the conditional expectations needed to solve models is similarly straightforward. For a discrete state space, transition probabilities generated by a finite state Markov chain approximation, such as Tauchen (1986) or Rouwenhorst (1995), can be applied by approximating the function whose conditional expectation is sought by a tensor train. For a continuous state space, Gauss-Hermite quadrature can be applied with the function approximated in the form of a functional tensor train.

4 An illustration

To illustrate the accuracy and speed at scale of tensor train approximation, consider the Constant Elasticity of Substitution (CES) production function:

$$Y = \left(\frac{1}{N} \sum_{p=1}^{N} x_p^{\phi} \right)^{\frac{1}{\phi}}, \qquad (49)$$

For simplicity suppose that $x_p \in [1,2], \forall\, p = 1, \ldots, N$. Figure 2 reports the approximation time and the accuracy associated with the tensor train approximation of equation (49), when 21 uniformly spaced points are used for each x_p and ϕ is set to 2.

Figure 2 shows how the approximation time increases as the number of variables that enter the function increases (panel A), along with the associated approximated error (panel B). As N increases the approximation time rises approximately linearly, and, with N equal to $1,000$,

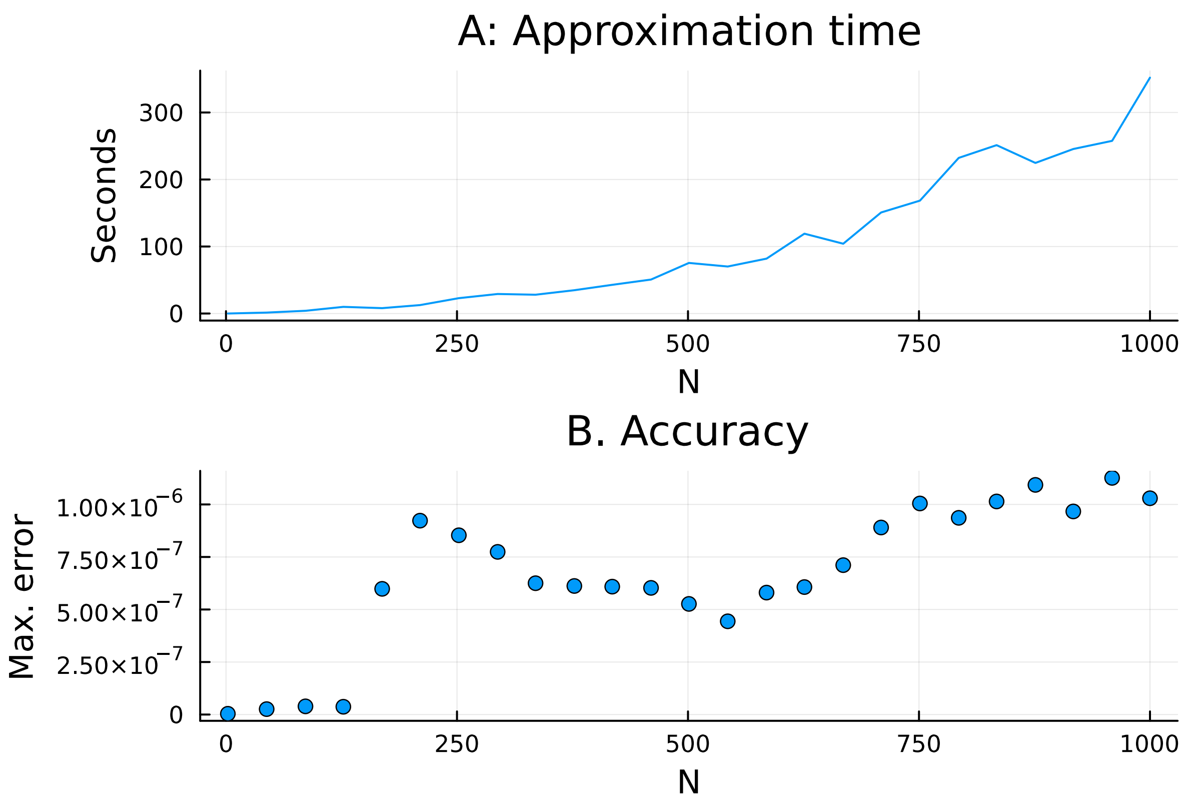

Figure 2: Tensor train approximation of CES function

the approximation time is under six minutes[8] with the largest absolute error of the order $1e^6$. Out of the 21^N points on the dense tensor-product grid, when $N = 1,000$ the approximation uses just $331,170$ of them, a minuscule proportion.

5 Model solving

We will illustrate how the tensor train approach can be used for model solving using an extended version of the stochastic growth model (Brock and Mirman, 1972); results for a Ramsey problem and a multi-country international business cycle model are provided in section 6. Consider an economy populated by a representative household that receives utility by consuming goods produced according to a neoclassical production technology and that saves by undertaking investment that augments the capital stock. Expressed as a sequence problem, the decision

[8]The times shown in Figure 2 were produced without parallelization using Julia 1.12.0 on a desktop using an Intel i9-14900K chip with 32GB RAM.

problem for the planner is to choose $\{c_t, i_t, k_{t+1}\}_0^\infty$ to maximize:

$$U = E_0 \left[\sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\sigma} - 1}{1-\sigma} \right], \tag{50}$$

with $\beta \in (0,1)$ and $\sigma \in (0,\infty)$, subject to the resource constraint:

$$e^{z_t} k_t^{\alpha} = c_t + i_t, \tag{51}$$

where $\alpha \in (0,1)$, and the capital accumulation equation:

$$k_{t+1} = (1 - \delta e^{d_t}) k_t + e^{q_t} i_t, \tag{52}$$

with $\delta \in (0,1)$.

The model accommodates shocks to aggregate technology, z_t, to investment-specific technology, q_t, and to the depreciation rate, d_t. We assume that the stochastic processes governing these three shocks are:

$$\begin{aligned} z_{t+1} &= \rho_z z_t + \sigma_z \epsilon_{z,t+1}, \qquad (53)\\ q_{t+1} &= \rho_q q_t + \sigma_q \epsilon_{q,t+1}, \qquad (54)\\ d_{t+1} &= \rho_d d_t + \sigma_d \epsilon_{d,t+1}. \qquad (55)\end{aligned}$$

where $\rho_j \in [0,1)$, $\sigma_j > 0$, and $\epsilon_{jt} \sim i.i.d.\ N(0,1)$, for $j \in \{z, q, d\}$.

This model contains an endogenous state variable, k_t, and up to three exogenous shocks. In our analysis below, we will consider three variants of this model, where the variants involve changing the number of shocks. As the number of shocks is varied so too is the number of state variables and the dimensions of cubature required to solve the model.

We will consider two broad techniques for solving this model. The first is value function iteration the second is policy iteration.

5.1 Value function iteration

To solve the model using Value Function Iteration (VFI) we first re-express the sequence problem in the form of a continuous state Markov decision problem.[9] For convenience, we also generalize the notation so that it relates to a class of problems for which the extended stochastic growth model is merely a special case.

[9]The conditions under which this can be done are well-known (see Stokey and Lucas (1989) or Miao (2014)).

Let $\mathbf{s}_t$ denote a vector of state variables and $\mathbf{a}_t$ denote a vector of choice variables. The policy function, or decision rule, $\pi(\mathbf{s}_t)$, is a vector-function that relates the choice variables to the state variables. Further, let $u(\mathbf{s}_t, \mathbf{a}_t)$ denote the household's felicity or momentary utility function and $\mathbf{s}_{t+1} = \mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \epsilon_{t+1})$, where $\epsilon_t \sim i.i.d.N[0, \Sigma]$, describe the stochastic law-of-motion for the state variables.

The value function associated with the decision rule $\mathbf{a}_t = \pi(\mathbf{s}_t)$ is given by:

$$V(\mathbf{s}_t) = E_t\left[\sum_{s=t}^{\infty} \beta^{t-s} u(\mathbf{s}_{t+s}, \pi(\mathbf{s}_{t+s}))\right], \quad \forall\ \mathbf{s}_t, \tag{56}$$

allowing the household's decision problem to be written recursively as:

$$V(\mathbf{s}_t) = \underset{\mathbf{a}_t}{max}\left[u(\mathbf{s}_t, \mathbf{a}_t) + \beta E_t\left[V\left(\mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \epsilon_{t+1})\right)\right]\right], \quad \forall\ \mathbf{s}_t, \tag{57}$$

with:

$$\pi^*(\mathbf{s}_t) = \underset{\pi}{argmax}\left[u(\mathbf{s}_t, \pi(\mathbf{s}_t)) + \beta E_t\left[V\left(\mathbf{f}(\mathbf{s}_t, \pi(\mathbf{s}_t), \epsilon_{t+1})\right)\right]\right], \quad \forall\ \mathbf{s}_t. \tag{58}$$

To solve the model using value function iteration together with tensor train approximation we use the procedure summarized in Algorithm 3. The "tt" subscript indicates that the object is in tensor train, or functional tensor train, format.

Algorithm 3 Value function iteration: global maximization

```
 1: Inputs $V^0_{tt}(\mathbf{s}_t)$, the candidate grids for $\mathbf{s}$ and $\mathbf{a}$, and an accuracy tolerance, $\gamma$.
 2: $i \leftarrow 0$.
 3: while True do
 4:     Given $V^i_{tt}(\mathbf{s}_t)$, construct $J^i(\mathbf{s}_t, \mathbf{a}_t) = u(\mathbf{s}_t, \mathbf{a}_t) + \beta E_t\left[V^i_{tt}(\mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \epsilon_{t+1}))\right]$.
 5:     $J^i_{tt}(\mathbf{s}_t, \mathbf{a}_t) \leftarrow$ DMRGcross($J^i(\mathbf{s}_t, \mathbf{a}_t)$).
 6:     $V^{i+1}_{tt} \leftarrow$DMRGcross($\underset{\mathbf{a}_t}{max}[J^i_{tt}(\mathbf{s}_t, \mathbf{a}_t)]$).
 7:     $len \leftarrow \frac{||V^{i+1}_{tt}(\mathbf{s}_t) - V^i_{tt}(\mathbf{s}_t)||}{||V^i_{tt}(\mathbf{s}_t)||}$.
 8:     if $len \leq \gamma$ then
 9:         Break
10:     end if
11:     $i \leftarrow i + 1$.
12: end while
13: $A^i_{tt} \leftarrow$TTsvd($\underset{\mathbf{a}_t}{argmax}[J^i_{tt}(\mathbf{s}_t, \mathbf{a}_t)]$).
14: Return $V^i_{tt}(\mathbf{s}_t)$, $A^i_{tt}(\mathbf{s}_t)$.
```

With one exception, all of the steps in Algorithm 3 involve the approximation and integration tools discussed previously. The exception is the optimization undertaken in step 6, for which we employ a modified[10] version of the TTOptimization algorithm developed in Chertkov, Ryzhakov, Novikov, and Oseledets (2022), which seeks to locate the maximal element of an array that is represented in tensor train format.[11] Step 13 uses TTsvd, however DMRGcross could be used instead. Notice that the final step of the algorithm delivers not just the value function, but also the decision rules.

5.2 Policy iteration

We will consider two Policy Iteration (PI) approaches. The first works with the Bellman equation, equation (57), the second draws on reinforcement learning and utilizes the advantage function (see, for example, Sutton and Barto (2018)). When utilizing the Bellman equation, we solve:

$$\pi\left(\mathbf{s}_t\right) = \underset{\mathbf{a}_t}{argmax}\left[u(\mathbf{s}_t, \mathbf{a}_t) + \beta E_t\left[V\left(\mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \epsilon_{t+1})\right)\right]\right], \quad \forall\ \mathbf{s}_t, \tag{59}$$

and then update the value function according to:

$$V\left(\mathbf{s}_t\right) = \left[u(\mathbf{s}_t, \pi(\mathbf{s}_t)) + \beta E_t\left[V\left(\mathbf{f}(\mathbf{s}_t, \pi(\mathbf{s}_t), \epsilon_{t+1})\right)\right]\right], \quad \forall\ \mathbf{s}_t. \tag{60}$$

To tackle the problem via reinforcement learning we introduce the advantage (or gain) function:

$$G\left(\mathbf{s}_t, \mathbf{a}_t\right) = u(\mathbf{s}_t, \mathbf{a}_t) + \beta\left(E_t\left[V\left(\mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \epsilon_{t+1})\right)\right] - V\left(\mathbf{s}_t\right)\right), \tag{61}$$

which states that the gain from an action is the immediate payoff implied by the felicity function together with the discounted expected change in the value function. Then:

$$\pi^*\left(\mathbf{s}_t\right) = \underset{\pi}{max}\left[G\left(\mathbf{s}_t, \pi\left(\mathbf{s}_t\right)\right)\right], \quad \forall\ \mathbf{s}_t, \tag{62}$$

with the value function again updated according to equation (60).

To use tensor trains to solve the model via policy iteration we use the solution procedure described in Algorithm 4. In the case where the gain function is used in place of the Bellman equation, line 4 in Algorithm 4 changes to employ equation (61). As noted previously, DMRGcross could have been used in step 6.

[10] I have modified the original algorithm, which is designed for unconstrained optimization across all cores in the tensor train, to optimize over selected cores, which allows the state variables to be taken as given.

[11] An alternative at this step is maximize $J^i(\mathbf{s}_t, \mathbf{a}_t)$ using a local method such as Newton's method or BFGS. This alternative approach tends to work better when there are many choice variables and is used to solve the Ramsey problems and the multi-country models analyzed in Section 6.

Algorithm 4 Policy iteration: Bellman equation

```
1: Inputs $V_{tt}^0(\mathbf{s}_t)$, the candidate grids for $\mathbf{s}$ and $\mathbf{a}$, and an accuracy tolerance, $\gamma$.
2: $i \leftarrow 0$.
3: while True do
4:     Given $V_{tt}^i(\mathbf{s}_t)$, construct $J^i(\mathbf{s}_t, \mathbf{a}_t) = u(\mathbf{s}_t, \mathbf{a}_t) + \beta E_t\left[V_{tt}^i(\mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \epsilon_{t+1}))\right]$.
5:     $J_{tt}^i(\mathbf{s}_t, \mathbf{a}_t) \leftarrow$ DMRGcross($J^i(\mathbf{s}_t, \mathbf{a}_t)$).
6:     $A_{tt}^i \leftarrow$TTsvd($\underset{\mathbf{a}_t}{argmax}[J_{tt}^i(\mathbf{s}_t, \mathbf{a}_t)]$).
7:     Construct the approximate decision rule $\pi(\mathbf{s}_t)$ from $A_{tt}^i$.
8:     $V^{i+1}(\mathbf{s}_t) \leftarrow$ DMRGcross($J^i(\mathbf{s}_t, \pi(\mathbf{s}_t))$).
9:     $len \leftarrow \frac{||V_{tt}^{i+1}(\mathbf{s}_t) - V_{tt}^i(\mathbf{s}_t)||}{||V_{tt}^i(\mathbf{s}_t)||}$.
10:    if $len \leq \gamma$ then
11:        Break
12:    end if
13:    $i \leftarrow i + 1$.
14: end while
15: Return $V_{tt}^i(\mathbf{s}_t)$, $A_{tt}^i(\mathbf{s}_t)$.
```

5.3 Results

In this section we examine the performance of value function iteration and policy iteration based on tensor train approximation, as per Algorithms 3 and 4, relative to approaches based on Chebyshev polynomials with a tensor-product grid and Smolyak and hyperbolic cross approximation based on sparse grids. We consider three variants of the model that depend on the number of active shocks. Model (a) has only the aggregate technology shock, model (b) has the aggregate technology shock and the investment-specific technology shock, while model (c) has the two technology shocks and the shock to the depreciation rate. Importantly, these three models differ in regard to the number of state variables as well as the number of dimensions of cubature required to form conditional expectations.

5.4 The setup

For this exercise, we parameterize the model by setting $\beta = 0.99$, $\sigma = 2.0$, $\alpha = 0.36$, $\delta = 0.02$, $\rho_z = 0.99$, $\rho_q = 0.90$, $\rho_d = 0.50$, $\sigma_z = 0.01$, $\sigma_q = 0.01$, and $\sigma_d = 0.01$. We structure the setup around a layer-four Smolyak polynomial, which dictates that there be 17 candidate sampling points along each dimension of the state space.[12] Accordingly, the tensor-product grid used by the Chebyshev polynomial method and the candidate grid for the tensor trains consist of 17^d

[12]To ensure nested layers, the points used for the Smolyak polynomial are the Chebyshev extrema.

points, where the points along each dimension are Chebyshev roots; for the hyperbolic cross polynomial $h = 8$ (see equation 6). The tensor train method used also requires a discretized grid for the choice variable, for which 51 uniformly spaced points for consumption were used primarily (see Section 4.7).[13] Numerical integration employs Gauss-Hermite quadrature with 11 points for each shock. The Chebyshev polynomial has order 4 over each of the shocks and order 6 over capital, which also governs the orders of the univariate polynomials used to form the functional tensor trains. Convergence is based on the value function with a tolerance, γ, equaling $1e^{-6}$.

All of the solution methods were initialized using an estimate of the value function formed by a second-order accurate perturbation solution. Similarly, the Chebyshev, Smolyak, and hyperbolic cross approximations employed a second-order accurate initial estimate of the consumption decision rule.[14]

Numerical accuracy is assessed based on the Euler-equation errors as a proportion of consumption:

$$\mathbb{E} = \frac{c_t - \left(\beta e^{q_t} E_t \left[c_{t+1}^{-\sigma} e^{-q_{t+1}} \left(1 - \delta e^{d_{t+1}} + \alpha e^{q_{t+1}} e^{z_{t+1}} k_t^{\alpha-1}\right)\right]\right)^{-\frac{1}{\sigma}}}{c_t}, \tag{63}$$

an expression that is not used to solve the model, making it a reasonably challenging performance metric.

5.5 TT-VFI versus TT-PI

We begin by considering only the tensor train approximation method and comparing value function iteration to policy iteration, where policy iteration employs either the Bellman equation or the advantage function. The three solution methods were applied to versions of the model that differ according to the number of shocks. In Table 1, (and all the tables that follow) the model with only the aggregate technology shock is denoted (a), the model with the aggregate technology shock and the investment-specific technology shock is denoted (b), while the model with all three shocks is denoted (c). For each of the three solution methods and each of the three models we report the cardinality of the chosen sampling points, N, the

[13]The domains for the three shocks were ± three unconditional standard deviations, the domain for capital was $[40.0, 56.0]$ and the domain for consumption was $[2.5, 3.7]$.

[14]All of the computations performed in this section and section 6 took place on a Intel i0-14900K chip paired with 32GB ram. The calculations were performed using Julia 1.12, parallelized using multi-threading across 32 logical processors. The second-order accurate solutions used to initialize the value function and the decision rules were obtained using SolveDSGE v0.6.0.

average absolute Euler-equation error, $log_{10}(\overline{|\mathbb{E}|})$, the maximum absolute Euler-equation error, $log_{10}(||\mathbb{E}||_\infty)$, and the solution time in seconds. The results are reported in Table 1.

Table 1: Value Function Iteration versus Policy Iteration

	VFI			PI - Bellman			PI - Gain		
	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
N	68	255	255	68	255	408	68	255	408
$log_{10}(\overline{\lvert\mathbb{E}\rvert})$	-4.06	-3.96	-3.90	-4.04	-3.93	-3.90	-4.04	-3.94	-3.91
$log_{10}(\lvert\lvert\mathbb{E}\rvert\rvert_\infty)$	-3.01	-3.13	-3.18	-3.04	-3.12	-3.18	-3.04	-3.12	-3.16
Time (sec)	1	10	43	1	22	238	2	37	364

Table 1 shows that each of the three solution methods produce similar levels of numerical accuracy for the three models. However, there is considerable variation in the solution times. Policy iteration using the Bellman equation takes about half the time as using the advantage function, but is considerably slower than value function iteration. For the three-shock version of the model, value function iteration took only 43 seconds, whereas policy iteration using the Bellman equation took 238 seconds and policy iteration using the advantage function took 364 seconds. Experiments using Howard iteration did not improve the performance of the two policy iteration methods, possibly because the second-order accurate value function used for initialization was decently accurate to begin with.

5.6 TT-VFI solution performance

The previous section showed that among the three tensor train solution methods, value function iteration was the fastest and produced equivalent accuracy to policy iteration. In this section we compare four solution methods that all employ value function iteration but differ in regard to how the value function and the decision rules are approximated. Specifically, we consider value function iteration using: 1) Chebyshev polynomials on a dense tensor-product grid; 2) tensor train approximation on an endogenously chosen grid; 3) Smolyak approximation on a layer-four sparse grid; and 4) hyperbolic cross approximation on a sparse grid. We report the same metrics as for Table 1. The results are reported in Table 2.

Looking first at the one-shock version of the model (columns (a)), it is apparent that the tensor train solution method is the slowest and least accurate. Nonetheless, it still delivers around 4 decimal places of accuracy on average. Moving to the two-shock version of the model (columns (b)), the tensor train method is now faster than using either Chebyshev polynomials or Smolyak polynomials, while still slower than using hyperbolic cross polynomials. In regard

Table 2: VFI Performance

	Chebyshev			Tensor train			Smolyak			Hyperbolic cross		
	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
N	289	4,913	83,521	68	255	255	65	177	401	57	129	241
$log_{10}(\overline{\lvert\mathbb{E}\rvert})$	-4.46	-3.97	-3.97	-4.06	-3.96	-3.90	-4.52	-3.98	-3.98	-3.98	-3.71	-3.65
$log_{10}(\lvert\lvert\mathbb{E}\rvert\rvert_\infty)$	-3.71	-3.56	-3.53	-3.01	-3.13	-3.18	-3.78	-3.58	-3.54	-3.27	-3.17	-2.76
Time (sec)	0.2	36	12,297	1	10	43	0.3	25	1,455	0.1	5	134

to accuracy, however, the tensor train method has accuracy similar to Chebyshev and Smolyak, and is more accurate than the standard hyperbolic cross employed here. For the three-shock version of the model (columns (c)), where the approach using Chebyshev polynomials is so non-competitive that it is not worth computing, the tensor train method is easily the fastest of the three sparse grid methods while delivering broadly equivalent or superior accuracy.

Looking at the solution times, Table 2 shows clearly that the solution time required by the tensor train approach increases much less rapidly than either of the other two sparse-grid approaches, and much less rapidly than the tensor-product grid used by the Chebyshev polynomial. Much of the tensor train's speed advantage comes from how quickly it performs numerical integration, as the number of points at which the state is sampled remains broadly comparable to Smolyak and the hyperbolic cross. Not only does the solution time increase less rapidly as the model gets larger, but it is apparent that the tensor train approach delivers faster solution times than the other sparse grid methods even for models with just three or four state variables.

5.7 Fineness of decision grid and solution accuracy

In this section we focus on the grid used for consumption, specifically its fineness. The Chebyshev, Smolyak, and hyperbolic cross methods considered above treat consumption as a continuous variable and maximize the value equation using Newton's method. In contrast, the tensor train method introduces a grid for consumption and performs a global optimization of the value function for each chosen state while restricting the solution to the consumption grid. As the number of points in the consumption grid increases, becoming increasingly fine, we might expect the tensor train solution to become more accurate. In Table 3 we consider uniform grids for consumption involving 51, 201, 501, and $1,001$ points.

As anticipated, using a finer grid for consumption does improve numerical accuracy, at the cost of slower solution times. However, for all three versions of the models using just 51

Table 3: Tensor Train Accuracy

	$N^c = 51$			$N^c = 201$			$N^c = 501$			$N^c = 1,001$		
	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
N	68	255	255	68	255	408	68	255	408	68	255	408
$log_{10}(\overline{\|\mathbb{E}\|})$	-4.06	-3.96	-3.90	-4.38	-3.90	-3.98	-4.45	-3.97	-3.92	-4.46	-3.97	-3.98
$log_{10}(\|\|\mathbb{E}\|\|_\infty)$	-3.01	-3.13	-3.18	-3.70	-3.43	-3.29	-3.63	-3.52	-3.33	-3.67	-3.51	-3.52
Time (sec)	1	10	43	2	19	63	5	36	95	10	74	191

points delivers reasonable accuracy together with fast solution times. Interestingly, as the number of points for the choice variable is increased the cardinality of the sampling grid used to approximate the value function does not.

6 Further examples

We analyze a Ramsey problem and a multi-country international business cycle model in this section in order to demonstrate that the benefits identified above carry over to other types of models and decision problems. For the applications in this section we depart from Algorithm 3 in order to employ a local minimizer in place of the tensor train optimizer. The algorithm used is:

Algorithm 5 Value function iteration: local minimization

```
1: Inputs $V_{tt}^0(\mathbf{s}_t)$, the candidate grid for $\mathbf{s}$ and an accuracy tolerance, $\gamma$.
2: $i \leftarrow 0$.
3: while True do
4:     Given $V_{tt}^i(\mathbf{s}_t)$, construct $J^i(\mathbf{s}_t, \mathbf{a}_t) = -(u(\mathbf{s}_t, \mathbf{a}_t) + \beta E_t\left[V_{tt}^i(\mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \epsilon_{t+1}))\right])$.
5:     $V_{tt}^{i+1} \leftarrow$DMRGcross$(-\underset{\mathbf{a}_t}{min}[J_{tt}^i(\mathbf{s}_t, \mathbf{a}_t)])$.
6:     $len \leftarrow \frac{||V_{tt}^{i+1}(\mathbf{s}_t) - V_{tt}^i(\mathbf{s}_t)||}{||V_{tt}^i(\mathbf{s}_t)||}$.
7:     if $len \leq \gamma$ then
8:         Break
9:     end if
10:    $i \leftarrow i + 1$.
11: end while
12: $A_{tt}^i \leftarrow$DMRGcross$(-\underset{\mathbf{a}_t}{argmin}[J_{tt}^i(\mathbf{s}_t, \mathbf{a}_t)])$.
13: Return $V_{tt}^i(\mathbf{s}_t)$, $A_{tt}^i(\mathbf{s}_t)$.
```

in which steps 4 and 5 replaces steps 4—6 in Algorithm 3. For the minimization step in Algorithm 5 we use Newton's method for the Ramsey problem and BFGS for the multi-country model.

6.1 A Ramsey problem

The model is populated by households, firms (intermediate- and final-good producers) and a central bank. The market for intermediate goods is monopolistically competitive, but all other markets are perfectly competitive, and the central bank conducts policy by setting the nominal return on a bond that is traded among households and is in zero-net-supply. Intermediate goods prices are assumed to be sticky, subject to a Rotemberg (1982) price-adjustment cost.

Households consume final goods, earn income by supplying labor to firms and through owning firms, and can borrow/save through the purchase of one-period, default-free, nominal bonds. The decision problem for the representative household is to choose $\{c_t, h_t, b_{t+1}\}_{t=0}^{\infty}$, where c_t denotes consumption, h_t denotes labor supply, and b_{t+1} denotes next-period's bond holdings, to maximize:

$$U = E_t \left[\sum_{t=0}^{\infty} \beta^t \left(\frac{c_t^{1-\sigma} - 1}{1-\sigma} - \mu e^{l_t} \frac{h_t^{1+\chi}}{1+\chi} \right) \right], \tag{64}$$

$\beta \in (0,1)$, $\sigma > 0$, $\mu > 0$, and $\chi > 0$, subject to the flow budget constraint:

$$c_t + Q_t b_{t+1} = w_t h_t + b_t + D_t, \tag{65}$$

in which Q_t denotes the bond price, w_t denotes the real wage, and D_t denotes aggregate dividend income. The first-order conditions for the representative household can be written as:

$$\begin{aligned} \mu e^{l_t} c_t^{\sigma} h_t^{\chi} &= w_t, && (66) \\ c_t^{-\sigma} &= \beta E_t \left[c_{t+1}^{-\sigma} \frac{1+R_t}{\pi_{t+1}} \right], && (67) \end{aligned}$$

where $(1+R_t)^{-1} = Q_t$, so R_t is the (net) nominal interest rate, and π_t represents inflation.

Production and pricing decisions by firms lead to:

$$\begin{aligned} \omega_t &= \frac{w_t}{e^{z_t}}, && (68) \\ \pi_t (1+\pi_t) &= \frac{1-\epsilon_t}{\phi} + \frac{\epsilon_t}{\phi}\omega_t + \beta E_t \left[\frac{c_{t+1}^{-\sigma} e^{z_{t+1}} h_{t+1} \pi_{t+1} (1+\pi_{t+1})}{c_t^{-\sigma} e^{z_t} h_t} \right], && (69) \\ D_t &= \left(1 - \omega_t - \frac{\phi}{2}\pi_t^2 \right) e^{z_t} h_t, && (70) \end{aligned}$$

in which ω_t denotes real marginal costs, ϵ_t denotes a markup shock, D_t denotes aggregate dividends, and the production function is given by $y_t = e^{z_t} h_t$, where z_t denotes an aggregate technology shock. The parameter $\phi > 0$ governs the magnitude of the nominal price rigidity.

The central bank's decision problem is to choose $\{c_t, h_t, \pi_t, \Gamma_t, \Phi_t\}_{t=0}^{\infty}$ to maximize the present-value Lagrangian:

$$\mathfrak{L} = E_0 \left[\sum_{t=0}^{\infty} \beta^t \left(\begin{array}{c} \frac{c_t^{1-\sigma}-1}{1-\sigma} - \mu e^{l_t} \frac{h_t^{1+\chi}}{1+\chi} \\ -\Gamma_t \left(\begin{array}{c} \left(\frac{1-\epsilon_t}{\phi} + \frac{\epsilon_t}{\phi} \frac{\mu e^{l_t} c_t^{\sigma} h_t^{\chi}}{e^{z_t}} - \pi_t (1+\pi_t) \right) c_t^{-\sigma} e^{z_t} h_t \\ +\beta c_{t+1}^{-\sigma} e^{z_{t+1}} h_{t+1} \pi_{t+1} (1+\pi_{t+1}) \end{array} \right) \\ -\Phi_t \left(c_t - (1 - \frac{\phi}{2}\pi_t^2) e^{z_t} h_t \right) \end{array} \right) \right], \tag{71}$$

giving rise to the first-order conditions:

$$\begin{aligned} \frac{\partial \mathfrak{L}}{\partial c_t} &= c_t^{-\sigma} - \Phi_t + \sigma \left(\frac{1-\epsilon_t}{\phi} - \pi_t (1+\pi_t) \right) c_t^{-\sigma-1} e^{z_t} h_t \Gamma_t \\ &+ \sigma c_t^{-\sigma-1} e^{z_t} h_t \pi_t (1+\pi_t) \Gamma_{t-1} = 0, \qquad (72) \\ \frac{\partial \mathfrak{L}}{\partial h_t} &= -\mu e^{l_t} h_t^{\chi} + \left(1 - \frac{\phi}{2} \pi_t^2 \right) e^{z_t} \Phi_t - c_t^{-\sigma} e^{z_t} \pi_t (1+\pi_t) \Gamma_{t-1} \\ &- \left(\frac{1-\epsilon_t}{\phi} + \frac{\epsilon_t (1+\chi)}{\phi} \frac{\mu e^{l_t} h_t^{\chi}}{c_t^{-\sigma} e^{z_t}} - \pi_t (1+\pi_t) \right) c_t^{-\sigma} e^{z_t} \Gamma_t = 0, \qquad (73) \\ \frac{\partial \mathfrak{L}}{\partial \pi_t} &= (\Gamma_t (1+2\pi_t) - \Gamma_{t-1} (1+2\pi_t)) c_t^{-\sigma} - \phi \Phi_t \pi_t = 0, \qquad (74) \end{aligned}$$

which hold for all $t \geq 0$ with the initial condition $\Gamma_{-1} = 0$, and, of course, the two constraints themselves. We do not use these first order conditions to solve the model, however we do use them to assess the accuracy of our solution.

We assume that the processes for the technology, markup, and labor supply shocks are:

$$\begin{aligned} z_{t+1} &= \rho_z z_t + \sigma_z \varepsilon_{t+1}, \qquad (75) \\ \epsilon_{t+1} &= \rho_\epsilon \epsilon_t + \sigma_\epsilon \eta_{t+1}, \qquad (76) \\ l_{t+1} &= \rho_l l_t + \sigma_l \nu_{t+1}, \qquad (77) \end{aligned}$$

with $\rho_b \in (0,1)$ and $\sigma_b > 0$ for $b \in \{z, \epsilon, l\}$.

To express the central bank's decision problem in terms of a Bellman equation we follow Marcet and Marimon (2019). Specifically, we group the terms in the constraints according to time, so we have:

$$\begin{aligned} \theta_t = &- \Gamma_t \left(\frac{1-\epsilon_t}{\phi} + \frac{\epsilon_t}{\phi} \frac{\mu e^{l_t} c_t^{\sigma} h_t^{\chi}}{e^{z_t}} - \pi_t (1+\pi_t) \right) c_t^{-\sigma} e^{z_t} h_t \\ &- \Gamma_{t-1} \left(c_t^{-\sigma} e^{z_t} h_t \pi_t (1+\pi_t) \right) \\ &- \Phi_t \left(c_t - \left(1 - \frac{\phi}{2} \pi_t^2 \right) e^{z_t} h_t \right), \qquad (78) \end{aligned}$$

and write the Bellman equation as:

$$V\left(z_t,\epsilon_t,l_t,\Gamma_{t-1}\right)=\underset{(c_t,h_t,\pi_t,\Gamma_t,\Phi_t)}{max}\left[\frac{c_t^{1-\sigma}-1}{1-\sigma}-\mu e^{l_t}\frac{h_t^{1+\chi}}{1+\chi}+\theta_t+\beta E_t\left[V\left(z_{t+1},\epsilon_{t+1},l_{t+1},\Gamma_t\right)\right]\right], \tag{79}$$

with the initial condition $\Gamma_{-1}=0$.

In contrast to the main text where we maximized the value function using a modified version of Chertkov et. al. (2022), in what follows we use univariate Chebyshev polynomials to interpolate over the (continuous) choice variables and use Newton's method to maximize the value function in step 5 of Algorithm 5.

Table 4 reports results where the decision problem summarized by equations (74)—(78) is solving using Chebyshev polynomials, Smolyak polynomials, and hyperbolic cross polynomials, and using the tensor train decomposition. The accuracy of each solution is assessed using the errors associated with equations (71)—(73), which we denote by $\mathbb{E}_1$, $\mathbb{E}_2$, and $\mathbb{E}_3$, respectively. Notably, these equations are not employed to obtain the solution itself. We consider three versions of the models that differ according to the number of active shocks. Model (a) has only the aggregate technology shock, model (b) has the technology and markup shocks, while model (c) has all three shocks. Conditional expectations are computed using Gauss-Hermite quadrature with 11 points for each shock.

To keep the exercise manageable for all four solution methods, we base it around a three-layer Smolyak polynomial, so 9 nodes are available along each spatial dimension. We use 4'th order Chebyshev polynomials for the Chebyshev approximation and for the tensor train approximation, and set $h=4$ for the hyperbolic cross polynomials.[15]

Table 4: Ramsey Monetary Policy

	Chebyshev			Tensor train			Smolyak			Hyperbolic cross		
	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
N	81	729	6,561	36	99	180	29	69	137	21	37	57
$log_{10}(\|\|\mathbb{E}_1\|\|_\infty)$	-8.67	-4.42	-4.41	-8.62	-4.42	-4.10	-9.35	-4.77	-4.76	-6.15	-3.16	-3.14
$log_{10}(\|\|\mathbb{E}_2\|\|_\infty)$	-8.26	-4.00	-4.01	-8.27	-4.00	-3.68	-9.29	-4.53	-4.50	-6.11	-2.90	-2.88
$log_{10}(\|\|\mathbb{E}_3\|\|_\infty)$	-7.37	-4.72	-4.71	-7.36	-4.72	-4.38	-9.32	-4.72	-4.71	-6.54	-3.22	-3.18
Time (sec)	1	264	48,154	45	282	1,211	2	327	20,044	2	149	6,513

The results in Table 4 reinforce those for the stochastic growth model presented in the main text. For the one-shock policy problem (columns (a)) Chebyshev polynomials on a dense

[15]The model is parameterized by setting $\beta=0.99$, $\sigma=1.00$, $\mu=1.0$, $\chi=3.0$, $\varphi=80.0$, $\epsilon=11.0$, $\rho_z=0.95$, $\sigma_z=0.01$, $\rho_\epsilon=0.80$, $\sigma_\epsilon=0.12$, $\rho_l=0.70$, and $\sigma_l=0.01$.

grid provides an extremely fast and accurate solution and among the sparse grid methods the tensor train approach is the slowest. With two shocks (so three states, columns (b)), using hyperbolic cross polynomials yields the fastest solution, while using either Smolyak polynomials or the tensor train approach remains slower than using the dense grid. For the specification with three shocks, however, (columns (c)) the tensor train method is by far the fastest of the methods considered, five times faster than the hyperbolic cross, fifteen times faster than using Smolyak polynomials, and nearly forty times faster than using Chebyshev polynomials with a dense grid. The accuracy of the tensor train can be improved by using a higher-order Chebyshev polynomial when forming the functional tensor train and its relative time advantage can be improved by using more grid points (i.e., by basing the comparison around a four-layer Smolyak polynomial, rather than around a three-layer polynomial).

6.2 A multi-country real business cycle model

Now we solve a multi-country real business cycle model in the spirit of Backus, Kehoe, and Kydland (1992). Viewed as a planner's problem, the decision problem for the N-country international real business cycle model is as follows. The planner chooses $\{c_{1t}, ..., x_{Nt}, k_{1t+1}, ...k_{Nt+1}\}_{t=0}^{\infty}$ in order to maximize:

$$E_0 \left[\sum_{t=0}^{\infty} \beta^t \left(\sum_{i=1}^{N} \frac{c_{it}^{1-\sigma} - 1}{1-\sigma} \right) \right], \tag{80}$$

$\beta \in (0,1)$ and $\sigma > 0$, so the utility of each country is weighted equally, subject to the resource constraint:

$$\sum_{i=1}^{N} (c_{it} + k_{it+1}) = \sum_{i=1}^{N} \left((1-\delta)k_{it} + e^{z_{it}} k_{it}^{\alpha} \right), \tag{81}$$

$\alpha \in (0,1)$, where the N technology processes follow:

$$z_{it+1} = \rho_i z_{it} + \sigma\epsilon_i \epsilon_{it+1}, \quad i = 1, \ldots, N, \tag{82}$$

with $\rho_i \in (0,1)$ and $\sigma\epsilon_i > 0 \ \forall i \in [1, \ldots, N]$.

The Bellman equation is:

$$V(\mathbf{s}_t) = \underset{(c_{1t},\ldots,c_{Nt},k_{1t+1},\ldots,k_{Nt+1},\lambda_t)}{max} \left(\sum_{i=1}^{N} \frac{c_{it}^{1-\sigma} - 1}{1-\sigma} + \theta_t + \beta E_t \left[V(\mathbf{s}_{t+1}) \right] \right), \tag{83}$$

where $\mathbf{s}_t = [z_{1t}, \ldots, z_{Nt}, k_{1t}, \ldots, k_{Nt}]$ and:

$$\theta_t = \lambda_t \left(\sum_{i=1}^{N} \left((1-\delta)k_{it} + e^{z_{it}} k_{it}^{\alpha} - c_{it} - k_{it+1} \right) \right). \tag{84}$$

The decision problem can be simplified further by recognizing that consumption will be the same in each country due to international risk sharing. If we denote this shared level of consumption by c_t, then the Bellman equation becomes:

$$V(\mathbf{s}_t) = \max_{(c_t, k_{1t+1}, \ldots, k_{Nt+1}, \lambda_t)} \left(N \frac{c_t^{1-\sigma} - 1}{1-\sigma} + \theta_t + \beta E_t \left[V(\mathbf{s}_{t+1}) \right] \right), \tag{85}$$

with:

$$\theta_t = \lambda_t \left(\sum_{i=1}^{N} ((1-\delta)k_{it} + e^{z_{it}} k_{it}^{\alpha} - k_{it+1}) - N c_t \right). \tag{86}$$

The simplified problem has $2N$ state variables, N shocks, and $N+2$ choice variables.

We solve the simplified problem allowing 11 approximating points for each state variable and using 4'th order Chebyshev polynomials to form the functional tensor train approximation. The value function maximization step (step 5 in Algorithm 5) is performed using the BFGS method, whose complexity is $O(n^2)$ in the number of choice variables (in contrast to Newton's method which is $O(n^3)$). Eleven Gauss-Hermite points are used for each shock to compute conditional expectations. As previously, the solution algorithm was initialized using a second order accurate perturbation solution.[16]

Table 5: Tensor-Train Solution of the N-Country Model

No. countries (N)	2	3	4	5	6
$log_{10}(\lvert\overline{\mathbb{E}}\rvert)$	-2.96	-2.89	-2.86	-2.85	-2.83
$log_{10}(\lvert\lvert\mathbb{E}\rvert\rvert_{\infty})$	-2.26	-2.20	-2.19	-2.20	-2.20
Time (sec)	36	482	3,851	9,331	26,295

For the six-country model there are twelve state variables, but the solution is obtained in just over seven hours. Much of this time, and the reason for the polynomial increase in solution times as the number of countries increases, rests with the numerical optimization of the value function, which for this model is the BFGS method. With numerical accuracy evaluated using the consumption Euler equation errors, it is notable that accuracy does not decline as the number of countries (i.e, the number of states, shocks, and choice variables) increases.

7 Conclusion

Despite enormous advances in computing power, the curse of dimensionality remains a significant bottleneck when it comes to solving and analyzing macroeconomic models, limiting the

[16]We parameterize the model by setting $\sigma = 2$, $\beta = 0.99$, $\delta = 0.02$, $\alpha = 0.36$, $\rho_i = 0.95$, $\forall\ i = 1, \ldots, N$, and $\sigma\epsilon_i = 0.01$, $\forall\ i = 1, \ldots, N$.

size of the models we can analyze and the questions we can ask of them.

This paper presents an approach to solving dynamic optimization models that is based on the tensor train decomposition, an approach that is relatively immune to the curse of dimensionality. The approach views an array as a discretized function and uses compression as an approximation tool. The tensor train decomposition separates by variables to decompose an array into a series of interlinked carriages, or cores. The separation of variables brought about by the tensor train decomposition makes otherwise expensive operations, such as numerical integration, incredibly fast.

Our approach dovetails the tensor train decomposition with value function iteration and policy iteration, using a functional version of the tensor train decomposition to approximate the value function at an endogenously chosen set of sampling points. We illustrate the power of our approach using an extended version of the stochastic growth model that allows the number of shocks in the model to be varied. We found that value function iteration out-performed policy iteration, and for the versions of the model that had either two or three shocks, our tensor train approach out-performed the use of Chebyshev polynomials, Smolyak polynomials, and hyperbolic cross polynomials. Notably, the size of the sampling grid used by the tensor train solution to approximate the value function did not increase exponentially with the number of state variables, and nor did the solution times. With four state variables, value function iteration using the tensor train method was faster than the hyperbolic cross method and significantly faster than Smolyak's method, while delivering better or equal numerical accuracy. We showed that working with a finer grid for the choice variable improved accuracy.

Results for a Ramsey problem and for a multi-country international business cycle model support and underscore the findings from the stochastic growth model. Employing value function iteration we solved a four state stochastic growth model in 43 seconds, a four state Ramsey problem in 20 minutes, and a twelve state international business cycle model in just over seven hours. Our results demonstrate that tensor train methods are a viable and powerful alternative to existing approaches to solving dynamic programming problems with many state variables.

This paper has focused on the application of tensor trains to solve dynamic programming problems. Although the use of tensor trains to solve dynamic optimization problems at scale is an important and ongoing topic of study, the usefulness of tensor trains extends far beyond dynamic programming. Within Econometrics, mixed-logit estimation, Bayesian estimation, multivariate density approximation, applications involving the EM algorithm, variational inference, and nonlinear filtering are all areas where tensor trains can fruitfully be applied and represent promising topics for future research.

References

[1] Backus, D., Kehoe, P., and F. Kydland, (1992), "International Real Business Cycles", *Journal of Political Economy*, 100, 4, pp. 745–775.

[2] Bellman, R., (1957), *Dynamic Programming*, Princeton University Press, Princeton, New Jersey.

[3] Bigoni, D., Engsig-Karup, A., and Y. Marzouk, (2016), "Spectral Tensor-Train Decomposition", *SIAM Journal on Scientific Computing*, 38, 4, A2405–A2439.

[4] Boyko, A., Oseledets, I., and G. Ferrer, (2021), "TT-QI: Faster Value Iteration in Tensor Train Format for Stochastic Optimal Control", *Computational Mathematics and Mathematical Physics*, 61, 5, pp. 836–846.

[5] Brock, W., and L. Mirman, (1972), "Optimal Economic Growth and Uncertainty: The Discounted Case", *Journal of Economic Theory*, 4, pp. 479–513.

[6] Chertkov, A., Ryzhakov, G., Novikov, G., and I. Oseledets, (2022), "Optimization of Functions Given in the Tensor Train Format", *arXiv*, DOI:10.48550/arXiv.2209.14808.

[7] Dennis, R., (2024), "Using a Hyperbolic Cross to Solve Nonlinear Macroeconomic Models", *Journal of Economic Dynamics and Control*, 163, article 104860.

[8] Dolgov, S., and D. Savostyanov, (2020), "Parallel Cross Interpolation for High-Precision Calculation of High-Dimensional Integrals", *Computer Physics Communications*, 246, article 106869.

[9] Dũng, D., Temlyakov, V., and T. Ullrich, (2018), "Hyperbolic Cross Approximation", Advanced Courses in Mathematics, CRM Barcelona, Centre de Recerca Matemàtica, Birkhäuser.

[10] Gorodetsky, A., Karaman, S., and Y. Marzouk, (2015), "Efficient High-Dimensional Stochastic Optimal Motion Control using Tensor-Train Decomposition", *11'th Conference on Robotics: Science and Systems*, Rome, Italy.

[11] Gorodetsky, A., Karaman, S., and Y. Marzouk, (2019), "A Continuous Analogue of the Tensor-Train Decomposition", *Computer Methods in Applied Mechanics and Engineering*, 347, pp. 59–84.

[12] Kruger, D., and F. Kubler, (2004), "Computing Equilibrium in OLG Models with Stochastic Production", *Journal of Economic Dynamics and Control*, 28, pp. 1411–1436.

[13] Marcet, A., and R. Marimon, (2019), "Recursive Contracts", *Econometrica*, 87, 5, pp. 1589–1631.

[14] Miao, J., (2014), *Economic Dynamics in Discrete Time*, The MIT Press, Cambridge, Massachusetts.

[15] Novak, E., and K. Ritter, (1998), "The Curse of Dimension and a Universal Method for Numerical Integration", in *Multivariate Approximation and Splines*, G. Nürnberger, W. Schmidt, and G. Waltz (eds), pp. 177–188.

[16] Oseledets, I., (2009), "A New Tensor Decomposition", *Soviet Mathematics, Doklady*, 80, 1, pp. 495–496.

[17] Oseledets, I., (2011), "Tensor-Train Decomposition", *SIAM Journal of Scientific Computing*, 33, 5, pp. 2295–2317.

[18] Oseledets, I., and E. Tyrtyshnikov, (2010), "TT-cross Approximation for Multidimensional Arrays", *Linear Algebra and its Applications*, 432, pp. 70–88.

[19] Rouwenhorst, K, (1995), "Asset Pricing Implications of Equilibrium Business Cycle Models," In: Cooley, T. (Ed.), Frontiers of Business Cycle Research, Princeton University Press, Princeton, NJ, pp. 294–330.

[20] Rotemberg, J., (1982), "Sticky Prices in the United States", *Journal of Political Economy*, 90, 6, pp. 1187–1211.

[21] Savostyanov, D., and I. Oseledets, (2011), “Fast Adaptive Interpolation of Multi-Dimensional Arrays in Tensor Train Format”, *7'th International Workshop on Multidimensional (nD) Systems*, Poitiers, France.

[22] Shetty, S., Lembono, T., Löw, T., and S. Calinon, (2024), “Tensor Train for Global Optimization Problems in Robotics”, *The International Journal of Robotics Research*, 46, 6, pp. 811–839.

[23] Shetty, S., Xue, T., and S. Calinon, (2024), “Generalized Policy Iteration using Tensor Approximation for Hybrid Control”, *The 12'th International Conference on Learning Representations*, Vienna, Austria.

[24] Smolyak, S., (1963), “Quadrature and Interpolation Formulas for Tensor Products of Certain Classes of Functions”, *Soviet Mathematics, Doklady*, 4, pp. 240–243.

[25] Stokey, N., and R. Lucas, (1989), *Recursive Methods in Economic Dynamics*, Harvard University Press.

[26] Sutton, R., and A. Barto, (2018), *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, Massachusetts.

[27] Tauchen, G., (1986), “Finite State Markov-Chain Approximations to Univariate and Vector Autoregressions,” *Economics Letters*, 20, 2, pp. 177–181.