Report - Solving high dimensional PDEs with tensor networks

Antoine Debouchage, Thomas Lemercier

April 19, 2024

Abstract

From physics and biology to finance and economics, partial differential equations (PDEs) are pervasive, serving as foundational tools in various mathematically oriented fields. Despite the diverse array of PDEs introduced to address these fields, they share a common challenge: the numerical resolution becomes increasingly difficult as the number of system variables grows. Real-world problems often manifest in high-dimensional settings, necessitating an exponential increase in parameters and data to achieve reliable solutions, a phenomenon referred to as the curse of dimensionality.

This challenge arose in quantum physics in the 1990s when quantum physicists sought efficient methods to simulate quantum systems without storing an impractical number of parameters. For instance, representing a system composed of 100 2-spin particles requires approximately 10^{30} parameters. One solution involved employing clever low-rank decompositions of systems, leveraging the assumption that most systems exhibit low entanglement and only occupy a small portion of their ambient Hilbert space. This approach gave rise to Matrix Product States (also known as Tensor Trains in the context of computer science), which later extended to more intricate structures as Tensor Networks.

Tensor networks, particularly tensor trains, hold promise for addressing high-dimensional partial differential equations. This report focuses on employing tensor trains to resolve parabolic PDEs.

Contents

1 Introduction

In this report, we present our findings regarding the implementation of a solver for parabolic Partial Differential Equations (PDEs) in high dimensions, utilizing tensor networks to approximate the underlying function. Our implementation is based on the methodology outlined in the paper by Lorenz Richter, Leon Sallandt, and Nikolas Nusken, titled "Solving high-dimensional parabolic PDEs using the tensor train format" [18], which employs tensor trains instead of classical neural networks [9] [17] to obtain an approximation at each time step of the PDE solution.

Tensor trains, also known as Matrix Product State, represent a class of tensor networks functioning as a chain of low-dimensional tensors that are contracted sequentially to reconstruct a higher-dimensional tensor. They offer efficient mathematical constructs and algorithms for representing high-dimensional tensors using low ranks as information-passing controllers. However, employing them entails challenging optimization processes, necessitating the use of specialized tensor-train-based algorithms that sequentially and iteratively optimize the tensor cores.

Our work was structured into several distinct phases:

- Development of a dedicated tensor train library: Utilizing tensor trains can be challenging due to the lack of standardized libraries, with existing options often lacking in features and possibly containing bugs due to their independent development and lack of external review and testing. To address this limitation and facilitate our algorithm implementation, we created our own library built on Numpy arrays, enabling easy manipulation of tensors and tensor networks.

- Implementation of Alternating Least Squares (ALS) algorithms: These optimization algorithms were employed in learning the tensor train at each time step of the PDE discretization process. Various variants of ALS were developed, including standard ALS, Modified Alternating Least Squares (MALS), and Stable Alternating Least Square Approximation (SALSA) [8] [6] [7] [3], tailored to the characteristics of tensor trains, enabling fast and efficient optimization.

- Establishment of the complete training pipeline: This encompassed explicit and implicit discretization schemes and the integration of Backward-Forward Stochastic Differential Equation (SDE) interfaces in order to solve PDEs.

- Benchmarking of all implementations, ranging from the ALS algorithms to the various pipelines, across different settings.

The report commences with an overview of Backward-Forward Stochastic Differential Equations, which serve to equivalently transform a parabolic PDE into a set of Stochastic Differential Equations through a generalized form of

the Feynman-Kac formula. Subsequently, we provide an introduction to tensor networks and their formalism, focusing particularly on tensor trains, elucidating their key properties relevant to our solver. With tensor trains as our foundation, we delve into a detailed description of the Alternating Least Squares algorithm, accompanied by its pseudo-code and inherent advantages.

Following this, we explained the integration of various components, starting from the PDE formulation to the discretization of the equivalent BFSDE, and the time-evolution optimization algorithms utilizing tensor trains. Finally, we present our diverse results, encompassing the benchmarking of ALS and SALSA discrepancies, alongside the outcomes of the entire pipeline.

2 Backward-Forward Stochastic Differential Equations (BFSDE)

In introduction to pricing theory and derivatives, a linear framework is derived from the Black-Scholes models and different limiting hypothesis are made (costless shorting, frictionless model with one single interest rate for lending and borrowing...). Obviously, these contrast with real market and there is a need to develop non linear pricing theory. In this section, we introduce the Backward-Forward Stochastic Differential Equations (BFSDE) [16] [4] that serve as an equivalence from the classical world of PDEs into the one of Stochastic Differential Equations.

Throughout this section, we consider a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and a d-dimensional $(\mathcal{F}_t)_{t\in[0,T]}$-Brownian motion $(W_t)_{t\in[0,T]} = (W_t^1, \cdots, W_t^d)_{t\in[0,T]}$ on it.

Let $n \in \mathbb{N}$ and consider,

$$f : [0,T] \times \Omega \times \mathbb{R}^n \times \mathbb{R}^{n\times d} \longrightarrow \mathbb{R}$$

The motivation is to find a generalization of the Feynman-Kac formula to represent semilinear parabolic PDEs of the form :

$$\partial_t u(t,x) + \mathcal{L}u(t,x) + f(t,x,u(t,x),Du(t,x)) = 0 \tag{1}$$

where $\mathcal{L} = \frac{1}{2}\sum_{i,j=1}^{d} \left(\sigma\sigma^\top\right)_{ij}(t,x)\partial_{x_i}\partial_{x_j} + \sum_{i=1}^{d} b_i(t,x)\partial_{x_i}$ with functions $b : [0,T] \times \mathbb{R}^d \longrightarrow \mathbb{R}^d$ and $\sigma : [0,T] \times \mathbb{R}^d \longrightarrow \mathbb{R}^{d\times d}$.

Additionally, let's define a stochastic process X as the solution of the following Stochastic Differential Equation (SDE) :

$$\mathrm{d}X_t = b(X_t)\mathrm{d}t + \sigma(X_t)\mathrm{d}W_t \tag{2}$$

2.1 Kolmogorov Forward and Backward Equations

We defined above $\mathcal{L}$ as the action of the infinitesimal generator of X on $\mathcal{C}^2$ functions. Denote by $\mathbb{P}_x$ the probability measure under which X satisfies the

dynamic of equation (2) with initial condition $X_0 = x$. Then, it is possible given a measure μ on $\mathbb{R}^d$ to define another measure $\mathbb{P}$ under which $X_0 \sim \mu$ and X satisfies (2). Such measure can be defined, for any measurable set A by,

$$\mathbb{P}(A) = \int_{\mathbb{R}^d} \mathbb{P}_x(A)\mu(\mathrm{d}x)$$

From Itô's lemma, we get the **Kolmogorov forward equation** also called the **Fokker-Planck equation**.

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbb{E}\left[\varphi(X_t)\right] = \mathbb{E}\left[\mathcal{L}\varphi(X_t)\right] \tag{3}$$

for any $\varphi \in \mathcal{C}_b^2$ ($\mathcal{C}^2$ bounded functions with their 1st and 2nd derivative bounded).

Moreover, assume that X_t has a density $p(t,\cdot)$ for any $t \geq 0$, then from 3

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\mathbb{R}^d} \varphi(x)p(t,x)\mathrm{d}x = \int_{\mathbb{R}^d} \mathcal{L}\varphi(x)p(t,x)\mathrm{d}x = \int_{\mathbb{R}^d} \varphi(x)\mathcal{L}^*p(t,x)\mathrm{d}x$$

where $\mathcal{L}^*$ is the L^2-adjoint of $\mathcal{L}$. This can be formally summarize as:

$$\partial_t p(t,x) = \mathcal{L}^* p(t,x), \quad p(0,x) = p_0(x) \tag{4}$$

which totally characterize the solution of a Stochastic Differential Equation by solving a PDE.

2.2 Backward Stochastic Differential Equations

Assuming a solution u exists to (1) whatever the initial condition is, if we can describe the dynamics of $Y_t = u(t, X_t)$, then we would have $u(t,x) = u(t, X_t^{t,x}) = Y_t^{t,x}$ (where $X_t^{t,x}$ is another dynamic of (2)).

Using Itô's lemma, we get :

$$\begin{aligned}
\mathrm{d}Y_t &= (\partial_t u(t,X_t) + \mathcal{L}u(t,X_t))\mathrm{d}t + D_x u(t,X_t)\sigma(X_t)\mathrm{d}W_t && (5)\\
&= -f(t,X_t,Y_t,D_x u(t,X_t))\mathrm{d}t + D_x u(t,X_t)\sigma(X_t)\mathrm{d}W_t && (6)\\
& && (7)
\end{aligned}$$

This suggests a slightly modified version of the previous equation:

$$\partial_d u(t,x) + \mathcal{L}u(t,x) + f(t,x,u(t,x),D_x u(t,X_t)\sigma(X_t)) = 0$$

Let $Y_t = u(t,X_t)$ and $Z_t = D_x u(t,X_t)\sigma(X_t)$. If u is a solution to the equation, we obtain:

$$\mathrm{d}Y_t = -f(t,X_t,Y_t,Z_t)\mathrm{d}t + Z_t\mathrm{d}W_t \tag{8}$$

2.3 Forward-Backward Stochastic Differential Equations

Using the same notation of above, we call a **Forward-Backward Stochastic Differential Equation (FBSDE)** the pair of equations with boundary conditions:

$$dX_s^{t,x} = b(s, X_s^{t,x})\mathrm{d}s + \sigma(s, X_s^{t,x})\mathrm{d}W_s,\ s \geq t \tag{9}$$

$$X_s^{t,x} = x,\ s \leq t \tag{10}$$

$$-\mathrm{d}Y_s^{t,x} = f(s, X_s^{t,x}, Y_s^{t,x}, Z_s^{t,x})\mathrm{d}s - Z_s^{t,x}\mathrm{d}W_s,\ 0 \leq s \leq T \tag{11}$$

$$Y_T = \Psi(X_T^{t,x}) \tag{12}$$

where $(t, x) \in \mathbb{R}_+ \times \mathbb{R}^d$, $X_s^{t,x} \in \mathbb{R}^d$, functions $b : \mathbb{R}_+ \times \mathbb{R}^d \longrightarrow \mathbb{R}^d, \sigma : \mathbb{R}_+ \times \mathbb{R}^d \longrightarrow \mathbb{R}^{d\times n}$ Borel measurable, $Y_s^{t,x} \in \mathbb{R}^m, Z_s^{t,x} \in \mathbb{R}^{m\times n}$.

Theorem 2.1 *Given standard parameters, the BSDE (12) has a unique solution* (Y, Z) *in* $H_T^2(\mathbb{R}^d) \times H_T^2(\mathbb{R}^{n\times d})$

3 Tensor Networks & Tensor trains

3.1 What are tensor networks?

In mathematics and physics, tensors are well-known objects that generalize matrices to higher dimensions. The field of multilinear algebra and the theory of tensor approximations play an essential role in computational mathematics and numerical analysis. Most problems arise in high dimension, where handling the size of the system becomes challenging due to the exponential growth in both memory requirements and operations as the dimension increases. For example, a d-dimensional problem involving n items results in a tensor of size n^d, quickly becoming incomprehensible.

The primary objective of tensor approximations is to discover efficient ways to represent such tensors using as few parameters as possible while maintaining a high level of accuracy. Tensor networks emerge as mathematical constructs that correspond to factorizations of very large tensors into networks of smaller ones. This involves identifying smaller tensors that, when contracted together, can reconstruct the original tensor with a given approximation error.

Tensor networks were originally devised in response to the curse of dimensionality in the study of many-body quantum systems during the 1990s. As such, they constitute a class of variational wave functions. Over the years, various types of tensor networks and algorithms have been developed, including Matrix Product States (MPS) (also called Tensor Trains (TT)) [5] [15] [12] [10] [19] [11] [1], Tree Tensor Networks (TTN) [20], Multi-scale Entanglement Renormalization Ansatz (MERA) [2], Projected Entangled Pair States (PEPS) [22] [23], among others.

3.2 Tensor diagrams

Tensor networks are accompanied by an intuitive graphical language that facilitates the visualization of complex operations between tensors: tensor (Penrose) diagrams [1] [14]. Introduced by Roger Penrose in the 1970s, these diagrams are invaluable for avoiding cluttered and heavy notations by representing tensors as nodes and contractions between them as edges. For readers familiar with quantum computing, it's worth noting that quantum circuits are essentially tensor diagrams with specific conventions for representing common operators such as measurements and Pauli gates.

Before jumping to the diagrams, let's discuss Einstein notation. Einstein notation is a notational convention aimed at abbreviating equations when dealing with tensors. The basic concept involves eliminating the summation symbol ($\sum$): when two indices are the same in an expression, it implies a summation. Additionally, tensors are expressed solely by their elements without explicitly denoting their indices. This streamlined notation simplifies equations involving tensors, enhancing readability and comprehension.

Another point is to express a tensor only by its element for any indices (i.e v_j for a tensor, M_{ij} for a matrix...).

For example, let $A \in \mathcal{M}_{n\times p}(\mathbb{R})$, we know that you can write the matrix A as the set of its elements $(A_{ij})_{i\in\{1,\cdots,n\},j\in\{1,\ldots,p\}}$. Using Einstein notation, when refering to the matrix A, you just write A_{ij} (assuming it is for any $i \in \{1, \cdots, n\}$ and $j \in \{1, \ldots, p\}$), $A \longrightarrow A_{ij}$

	Standard Notation	Einstein Notation
Vector	$v = (v_i)_{i\in\{1,\cdots,n\}}$	v_i
Matrix	$M = (M_{ij})_{ij}$	M_{ij}
Tensor	$T = \left(T_{ij}^{ab}\right)_{ab,ij}$	T_{ij}^{ab}

	Standard Notation	Einstein Notation
Matrix x Vector	$w_i = \sum_j M_{ij} v_j$	$w_i = M_{ij} v_j$
Matrix product	$C_{ik} = \sum_j A_{ij} B_{jk}$	$C_{ik} = A_{ij} B_{jk}$

Figure 1: Simple Einstein notation

In a tensor diagram, a tensor corresponds to a (labelled) shape such as a circle, rectangle, diamond with one or more output legs. Those legs represent the indices of the tensor. For instance, a vector that only has one index will be represented as a circle with one leg (potentially labelled with the name of the index). A matrix has two indices (rows and columns) and thus will be a circle with two output legs.

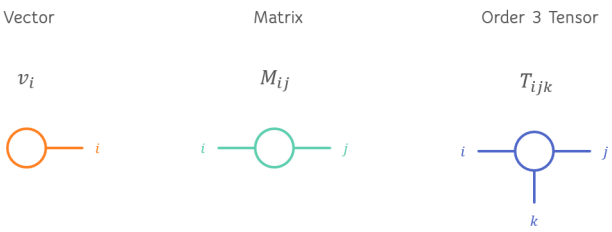


Figure 2: Tensor diagram for the base elements (vector, matrix, tensor)

The generalisation of matrix multiplication to tensors is called tensor contractions where two axes are being summed over $T^{ef}_{ab} = U^{cd}_{ab} V^{ef}_{cd} = \sum_{c,d} U^{cd}_{ab} V^{ef}_{cd}$ where we did not use the Einstein notation in the last equality.

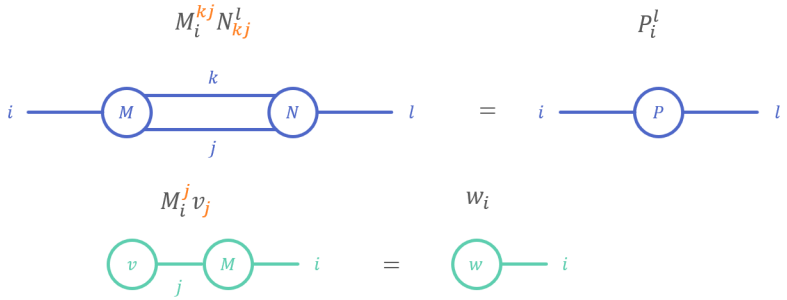


Figure 3: Tensor contractions

3.2.1 Tensor trains

A tensor train [12] is a class of tensor networks that is both efficient and easy to implement. It corresponds to a chain of subtensor of small rank controlled by an inner rank parameter r (or multiple ranks $r_1, \cdots, r_d$) such that a large tensor $T_{m_1,\cdots,m_d}$ is decomposed into $U_1{}^{r_1}_{r_0,m_1} U_2{}^{r_2}_{r_1,m_2} \cdots U_d{}^{r_d}_{r_{d-1},m_d}$ with each $U_i \in \mathbb{R}^{r_{i-1} \times m_i \times r_i}$ a three-order tensor called the i-th core of the tensor train.

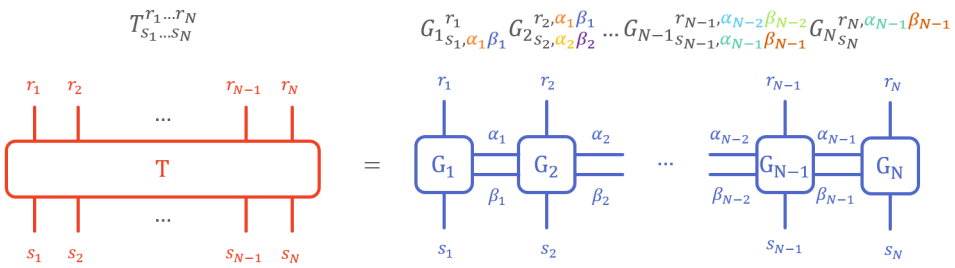


Figure 4: More difficult example with T be a $2N$-order tensor

The inner dimension r controls how much information is needed to represent the full tensor. The main assumption comes from quantum physics where a quantum state lies in a huge Hilbert space but only a very small portion of it is actually necessary to represent the given state. Indeed, in simple low-intricated systems only particules that are close to each other interacts with long distance interaction being negligible.

Using this paradigm, we can represent up to an approximation error leveraged by r a tensor of $O(n^d)$ parameters using only $O(dr^2n)$ parameters (d cores of $O(r^2n)$ parameters). This allows to reduce the number of parameters from a number exponential to the dimension of the problem to only polynomial with the rank contributing to this tradeoff of information. For consistency, the first and last cores are kept as three-order tensor with dummy indices of shape 1 ($r_0 = r_d = 1$).

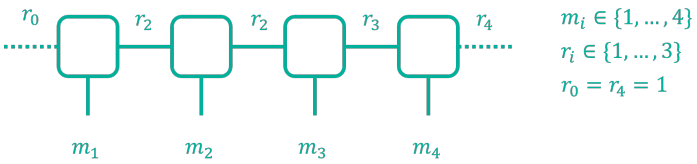


Figure 5: Tensor train with 4 cores representing a tensor of shape $(4, 4, 4, 4)$ i.e 256 parameters with only $12 + 36 + 36 + 12 = 96$ parameters.

Many algorithms have been developed to use tensor train (also called Matrix Product States in the litterature linked to quantum physics and quantum computing). The most important ones are:

- **TT-SVD**: to convert a high-dimensional tensor into a tensor train with given approximation by performing a series of SVD [12]
- **TT-rounding**: to reduce the inner rank r of each core in a tensor train [12]
- **Product** and **addition** of two tensor trains [12]

- **Solving linear systems**: Density Matrix Renormalization Group (DMRG) etc. [19]
- **Time evolution algorithm**: Trotter Gate Time Evolution (TEBD), MPO W^{II} time evolution, local and global Krilov time evolution, Time-Dependent Variational Principle (TDVP) [13]
- **Optimisation algorithm (detailed in this report)**: Alternating Least Squares and its variants [8] [7] [6].

3.2.2 Unfolding

Tensor unfolding smoothly converts a tensor into a matrix while keeping its properties and relationships intact. By organizing the tensor's elements along its mode dimensions, we reshape it into a matrix where each row or column represents a set of unique tensor indices. This simpler format keeps the essence of the data and makes complex tensor calculations easier by using standard matrix operations. In our case, using tensor unfolding with tensor train method is especially helpful during optimization steps. Since tensor trains mostly deal with three-dimensional tensors, we call the unfolding steps left and right unfolding, shown in Figure 6, corresponding to the different mode of unfolding of the tensor (there is an additional mode with the ranks being merge together into a matrix $r_{i-1}r_i \times m - i$ but it is not useful in our algorithm).

Thus converting by reshape operation a tensor in $\mathbb{R}^{r_{i-1}\times m_i \times r_i}$ into a matrix $\mathbb{R}^{r_{i-1}m_i \times r_i}$ for a left-unfolding and $\mathbb{R}^{r_{i-1}\times m_i \times r_i}$ into a matrix $\mathbb{R}^{r_{i-1}\times m_i r_i}$ for a right-unfolding. This will prove usefull when performing QR and SVD decomposition of tensor cores as most numercial computation library can only deal with matrix rather than tensor for such decomposition.

In the following section, we denote by $\mathcal{L}$ and $\mathcal{R}$ the left and right unfolding operators i.e $\mathcal{L}(U_i) \in \mathbb{R}^{r_{i-1}m_i \times r_i}$, $\mathcal{R}(U_i) = \mathbb{R}^{r_{i-1}\times m_i r_i}$.

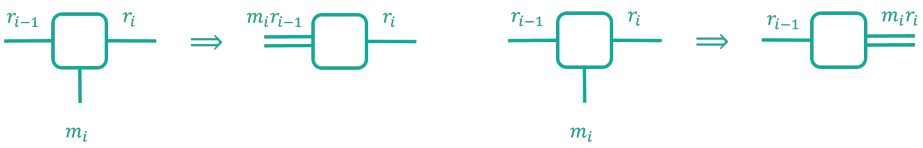


Figure 6: Left and Right unfolding

3.2.3 Orthonormalization & Canonical form

An important result of tensor trains is that they can be converted into a **canonical form** that will give interesting orthonormal properties while being strictly isometric to its original representation.

This canonical form can be left or right orthonormal. The cores of a tensor train that are partial isometries respect the properties that:

- $\mathcal{L}(U_i)^\intercal \mathcal{L}(U_i) = I_{r_i}$ (if left-orthonormal)

- $\mathcal{R}(U_i)\mathcal{R}(U_i)^\intercal = I_{r_{i-1}}$ (if right-orthonormal)

With this property in mind, in algorithms involving quadratic forms that utilize a tensor train, contractions with the same core often occur. In a canonical form, many operations can be eliminated due to the orthonormalization property. This property simplifies computations by reducing redundant operations, ultimately streamlining the algorithm and enhancing efficiency.

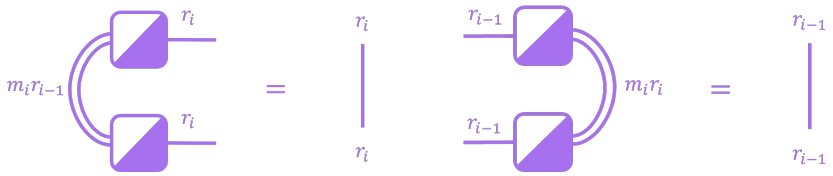


Figure 7: Left and right orthonormal cores

The most efficient way to convert a tensor train into a left or right canonical form is to perform a series of SVD or QR decomposition from one end to the other and updating the next core at each iteration with the non-orthonormal part of the decomposition. Only the last core will not be orthonormal and will gather all information about the norm of the tensor train.

Algorithm 1 Left-orthonormalisation of a TT

Require: A tensor train with cores $(U_1, \cdots, U_d)$
1: **for** $i = 1, 2, \ldots, d-1$ **do**
2: Perform a QR decomposition $Q_i R_i = \mathcal{L}(U_i)$.
3: Set $U_i \leftarrow Q_i$ (with appropriate reshape).
4: Set $U_{i+1} \leftarrow R_i U_{i+1}$ (with appropriate reshape)
5: **end for**

Algorithm 2 Right-orthonormalisation of a TT

Require: A tensor train with cores $(U_1, \cdots, U_d)$
1: **for** $i = d-1, \ldots, 1$ **do**
2: Perform a QR decomposition $Q_i R_i = \mathcal{R}(U_i)^\intercal$.
3: Set $U_{i-1} \leftarrow U_{i-1} R_i^\intercal$ (with appropriate reshape)
4: Set $U_i \leftarrow Q_i^\intercal$ (with appropriate reshape).
5: **end for**

3.2.4 Derivatives, Gradient & Hessian of tensor trains

Performing calculus on tensors can be tedious especially for functions of multiple tensors such as tensor networks where subtensors are densely contracted. However, tensor diagrams provide an efficient way to perform algebra on such mathematical objects. Differentiating a tensor train by one of its core simply

boils down to removing it as in figure 8 because each core are independent of the others and all interactions are purely linear.

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}U_i} U_{m_1,\cdots,m_d} &= \frac{\mathrm{d}}{\mathrm{d}U_i} \sum_{r_1,\cdots,r_{d-1}} {U_1}^{r_1}_{r_0,m_1} {U_2}^{r_2}_{r_1,m_2} \cdots {U_d}^{r_d}_{r_{d-1},m_d} \\
&= \sum_{r_1,\cdots,r_{d-1}} \frac{\mathrm{d}U_i}{\mathrm{d}U_1}^{r_1}_{r_0,m_1} {U_2}^{r_2}_{r_1,m_2} \cdots {U_d}^{r_d}_{r_{d-1},m_d} \\
&= \sum_{r_1,\cdots,r_{d-1}} {U_1}^{r_1}_{r_0,m_1} {U_2}^{r_2}_{r_1,m_2} \cdots \frac{\mathrm{d}}{\mathrm{d}U_i}\left({U_i}^{r_i}_{r_{i-1},m_i}\right) \cdots {U_d}^{r_d}_{r_{d-1},m_d} \\
&= \sum_{r_1,\cdots,r_{d-1}} {U_1}^{r_1}_{r_0,m_1} {U_2}^{r_2}_{r_1,m_2} \cdots I_{m_i} \cdots {U_d}^{r_d}_{r_{d-1},m_d}
\end{aligned}$$

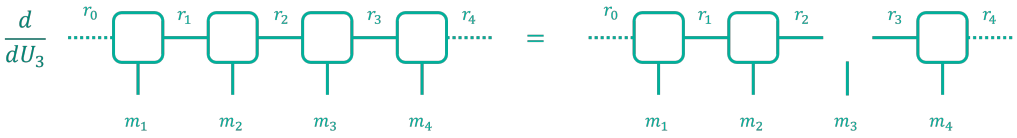


Figure 8: Core derivative

Following this pattern the Gradient of a tensor train is a list of tensor trains with a hole on the corresponding derived core (see figure 9).

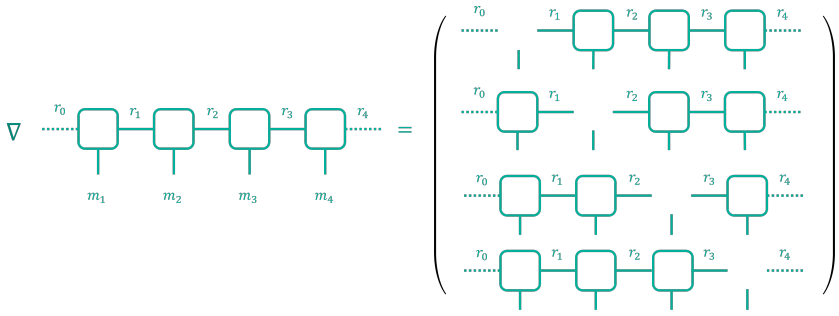


Figure 9: Gradient of a tensor train

The hessian of tensor train is, thus, computed the same way by removing the cores i and j for the element H_{ij} with diagonal element being 0 as the tensor train does not have second partial derivatives on the same core because of its linearity.

4 Algorithms

4.1 Tensor train Optimisation

Alternating Least Squares [21] [3] is a well-used algorithm that have been developed by decomposing a matrix factorisation problem into several sequential micro-optimisation ones. In the context of tensor trains, which represent tensors in a specific decomposition form, ALS can be extended to tensor networks, leading to various variants and extensions tailored for tensor trains. In this study, we present descriptions of the standard ALS generalization, a modified approach (MALS), and a regularized version (SALSA), which incorporate dynamical rank adaptation.

These algorithms are particularly effective for solving linear equations and are suitable for minimizing Mean Squared Error problems. Resolving high-dimensional functions like Partial Differential Equations (PDEs) poses significant numerical challenges due to the curse of dimensionality. Direct resolution of such functions is often impractical, hence approximations using techniques like tensor networks offer powerful tools by leveraging the inherent linear properties of tensor spaces.

In the tensor train format, a tensor U is represented as a multi-linear combination of small component tensors U_i. In order to easily optimize such object, the ALS-like algorithms depend on a relaxation techniques : only one or two components are optimised at a time keeping the other fixed. However, convergence to a minimum is not guaranteed for this kind of relaxation.

4.1.1 Micro-optimization

Every ALS-like algorithm rely upon a series of micro-optimisation that only focuses on a small part of the system (here tensor train) that we would like to minimize.

Given a functional $\mathcal{J} : \mathbb{R}^{n_1 \times \cdots \times n_d} \longrightarrow \mathbb{R}$. We want to solve the problem $AU = B$ with $U, B \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ and A a symmetric positive define operator from $\mathbb{R}^{n_1 \times \cdots \times n_d}$ to $\mathbb{R}^{n_1 \times \cdots \times n_d}$. However, in this work, A need not be defined, as the goal is to fit a tensor train U given a batch of features $\varphi(X)$ as an input such that $U_{m_1,\cdots,m_d}\varphi(X_i)^{m_1,\cdots,m_d} = b_i$ for all $i \in \{1, \cdots, N\}$ and $\mathbf{b} = (b_i)_i$ a vector of right hand side results. Here X_i corresponds to a vector of assets $(x_i^{(1)}, \cdots, x_i^{(d)})$ that are mapped to a feature space of dimension m_i through φ. For instance, a polynomial feature space such that $x_i^{(p)} \longmapsto \left(1, x_i^{(p)}, {x_i^{(p)}}^2, \cdots, {x_i^{(p)}}^k\right)$.

The functional considered is thus

$$\mathcal{J} : V, \Phi \in \mathbb{R}^{n_1 \times \cdots \times n_d} \times \mathbb{R}^{n_1 \times \cdots \times n_d} \longrightarrow \frac{1}{2}\langle V \circ \Phi, V \circ \Phi\rangle - \langle b, U \circ \Phi\rangle$$

Now, to only select a part of the tensor train to optimise a given core at a time, one need a **retraction operator**. This is an operator that maps a core

to an full tensor train via

$$P_i : U \in \mathbb{R}^{r_{i-1} \times m_i \times r_i} \longrightarrow U_1{}^{r_1}_{r_0,m_1} U_2{}^{r_2}_{r_1,m_2} \cdots U^{r_i}_{r_{i-1},m_i} \cdots U_d{}^{r_d}_{r_{d-1},m_d} \in \bigoplus_{i=1}^{d} \mathbb{R}^{m_i}$$

Our formalism slightly diverges from what has been introduced in the original paper because of the featured space. Hence, our actual retraction operator is a contraction between the original and the feature space tensor $\Phi(X)$ and can be represented as a simple tensor of shape $\mathbb{R}^{r_{i-1} \times m_i \times r_i}$ (see figure 10).

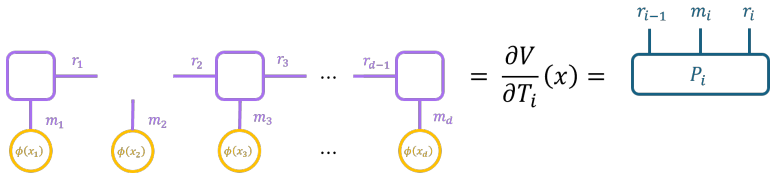


Figure 10: First order retraction operator

We then perform a quadratic optimisation to find the core that minimises $\mathcal{J} \circ (P_i, I)$ as show with the diagram of figure 11.

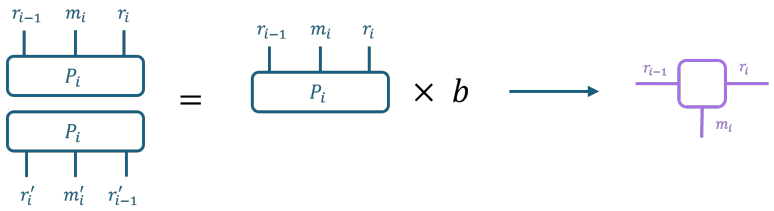


Figure 11: Quadratic minimisation to obtain an optimised core

Additionnaly, regularisation terms can be added as in the case of the SALSA algorithm described below.

4.2 Alternating Least Squares (ALS)

The initial algorithm we discuss is a direct extension of the standard ALS algorithm [21]. Beginning with a right-orthonormal tensor train up to component U_2, as previously describe in the tensor trains section, we undergo a series of Singular Value Decompositions (SVD) or QR decompositions for each core sequentially, starting from U_d to orthonormalize core i at step i, while updating core U_{i-1} with the remainder (R in QR, diag(S)V in SVD) up to core U_2.

Subsequently, the initial micro-optimization is executed on the first core U_1, resulting in a new core V_1. Similar to the orthonormalization process, we transfer the non-orthonormal part of V_1 to U_2 via a QR decomposition.

In more detail, V_1 assumes the shape of $\mathbb{R}^{r_0 \times m_1 \times r_1}$. We then unfold it to the left into a matrix with dimensions $\mathbb{R}^{(r_0 \times m_1) \times r_1}$ prior to the QR decomposition: $[V_1]r_0, {m_1}^{r_1} = [Q]r_0, {m_1}^{r_1}[R]{r_1}^{r_1}$. Subsequently, core U_1 is set to Q, and we update core U_2 by transferring the non-orthonormal part R as $[U^{'}2]{r_1}^{m_2,r_2} = [R]{r_1}^{r_1}[U_2]_{r_1}^{m_2,r_2}$.

This process is then iteratively applied from left to right up to U_{d-}. Such computations from left to right are referred to as a "half-sweep". We then conduct another half-sweep in the opposite direction, from right to left, by replicating the same operations (after the left half-sweep, the tensor train will indeed be left orthonormalized). This culmination constitutes a full iteration sweep of the ALS.

Algorithm 3 ALS

for $iteration = 1, 2, \ldots, N$ **do**
  for $i = 1, 2, \ldots, d-1$ **do**
    Get the retraction operator $P_i = \prod_{j \neq i} U_j$.
    $V_i = \arg\min \mathcal{J} \circ P_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$.
    Perform a QR decomposition on the optimisation result $Q_i R_i = V_i$.
    Set $U_i \leftarrow Q_i$.
    Set $U_{i+1} \leftarrow R_i U_{i+1}$.
  end for
  for $i = d-1, d-2, \ldots, 1$ **do**
    Get the retraction operator $P_i = \prod_{j \neq j} U_j$.
    $V_i = \arg\min \mathcal{J} \circ P_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$.
    Perform a QR decomposition on the optimisation result $Q_i R_i = V_i^{\intercal}$.
    Set $U_{i-1} \leftarrow U_{i-1} R_i^{\intercal}$.
    Set $U_i \leftarrow Q_i^{\intercal}$.
  end for
end for

4.3 Modified Alternating Least Squares (MALS)

Although the aforementioned ALS version yields generally good solutions, it presents two drawbacks:

1. Orthonormalization is required at each micro-step using a QR decomposition.
2. Adapting the ranks to achieve the desired accuracy is not straightforward.
3. It may lead to unstable solution for too complex problems.

To address these issues, the Modified Alternating Least Squares (MALS) [6] algorithm employs a slightly more costly greedy strategy tailored specifically to tensor trains. The key concept involves focusing on optimizing two cores simultaneously instead of just one, thereby reducing the necessity to adjust the

ranks through a truncated Singular Value Decomposition (SVD) to separate both cores after the micro-optimization.

At step i, we consider cores U_i and U_{i+1}, which can be contracted to form a tensor $W = U_i U_{i+1}$ with dimensions $\mathbb{R}^{r_{i-1} \times m_i \times m_{i+1} \times r_{i+1}}$, contracted along the r_i axis. This tensor is then optimized using a routine similar to what has been explained in the micro-optimisation section with a second order retraction operator that will removes two cores from the initial tensor train. Subsequently, its unfolding is split into two new cores using a truncated SVD, as follows:

$$[W]_{r_{i-1},m_i}^{m_{i+1},r_{i+1}} \approx [U]_{r_{i-1},m_i}^{r_i} [S]_{r_i}^{r_i} [V]_{r_i}^{m_{i+1},r_{i+1}} = [U_i]_{r_{i-1},m_i}^{r_i} [U_{i+1}]_{r_i}^{m_{i+1},r_{i+1}}$$

$$\text{where } [U_{i+1}]_{r_i}^{m_{i+1},r_{i+1}} = [S]_{r_i}^{r_i} [V]_{r_i}^{m_{i+1},r_{i+1}} \text{ and } [U_i]_{r_{i-1},m_i}^{r_i} = [U]_{r_{i-1},m_i}^{r_i}$$

Algorithm 4 MALS

for *iteration* $= 1, 2, \ldots, N$ **do**
  for $i = 1, 2, \ldots, d-2$ **do**
    Get the retraction operator $P_i^{(2)} = \prod_{j \notin \{i,i+1\}} U_j$.
    $W_{i,i+1} = \arg\min \mathcal{J} \circ P_i^{(2)} \in \mathbb{R}^{r_{i-1} \times n_i \times n_{i+1} \times r_{i+1}}$.
    Perform an SVD decomposition $W_i = U_i S_i V_i$.
    Rank adaptation: Truncate the singular values up to an error ε.
    Set $U_i \leftarrow U_i^{(trunc)}$.
    Set $U_{i+1} \leftarrow S_i^{(trunc)} V_i^{(trunc)}$.
  end for
  for $i = d-2, d-3, \ldots, 1$ **do**
    Get the retraction operator $P_i^{(2)} = \prod_{j \notin \{i,i+1\}} U_j$.
    $W_{i,i+1} = \arg\min \mathcal{J} \circ P_i^{(2)} \in \mathbb{R}^{r_{i-1} \times n_i \times n_{i+1} \times r_{i+1}}$.
    Perform an SVD decomposition $W_i^\intercal = U_i S_i V_i$.
    Rank adaptation: Truncate the singular values up to an error ε.
    Set $U_{i+1} \leftarrow U_i^{(trunc)\intercal}$.
    Set $U_i \leftarrow \left(S_i^{(trunc)} V_i^{(trunc)} \right)^\intercal$.
  end for
end for

4.4 Stable Alternating Least Squares Approximation (SALSA)

The Alternating Least Squares (ALS) algorithm is indeed a potent tool well-suited for optimizing tensor trains, thanks to the interconnected nature of sub-tensors resembling a chain. However, its standard version often struggles when confronted with tensor trains containing numerous cores. This is primarily due to its instability and the accumulation of slight precision errors, which eventually lead to divergence.

To address this challenge, a novel algorithm was introduced in 2019: the Stable Alternating Least Squares Approximation (SALSA) algorithm [7]. SALSA

is designed to ensure high stability even for high-dimensional problems, eliminating the necessity to scale the number of ALS iterations.

Algorithm 5 SALSA (solve $AX = b$)

Require: η (regularisation coefficient)
Require: τ (singular value threshold)
Initialize a random tensor train with cores $(U_1, \cdots, U_d)$
for *iteration* $= 1, 2, \ldots, N$ **do**
    Right-orthonormalize the tensor train
    for $i = 1, 2, \ldots, d$ **do**
        if $i \neq 1$ **then**
            $USV = \mathrm{SVD}(U_{i-1})$
            Set $\Gamma = (\max(s_j, \tau))_j$ (s_j j-th singular values of S)
            if $S[-1] > \tau$ **then**
                Expand rank of U, S, VU_i
            end if
            Set $U_{i-1} \leftarrow U$
            Set $U_i \leftarrow SVU_i$
        end if
        if $i \neq d$ **then**
            $USV = \mathrm{SVD}(U_i)$
            Set $\Theta = (\max(s_j, \tau))_j$ (s_j j-th singular values of S)
            Set $U_i \leftarrow US$
            Set $U_{i+1} \leftarrow VU_{i+1}$
        end if
        Get the retraction operator $P_i = \prod_{j \neq i} U_j$.
        $V_i = \arg\min (\mathcal{J} \circ P_i + \Omega(\Gamma, \Theta)) \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$.
        Set $U_i \leftarrow V_i$.
    end for
    Compute the residual $R = P_d U_d$
    $\eta, \eta_{\mathrm{tmp}} = ||R - b||^2, \eta$
    $\mathrm{rel}_\eta = \frac{\eta_{\mathrm{tmp}} - \eta}{\eta_{\mathrm{tmp}}}$
    $\omega = \max\left(\min\left(\omega / f_\omega, \max\left(\eta, \sqrt{\eta}\right)\right), \mathrm{rel}_\eta\right)$
    $\tau = (\omega/5, \mathrm{rel}_\eta/5)_+$
end for

During the micro-optimisation step, we keep the exact algorithm as in the standard ALS process. However, regularised terms based on the inverse of the singular values of the previous and current cores are added to the operator. These regularisation terms are computed as in figure (12).

$$
\begin{aligned}
A^{r'_{i-1},m'_i,r'_i}_{r_{i-1},m_i,r_i} &= P_{r_{i-1},m_i,r_i} P_{r'_{i-1},m'_i,r'_i} \\
&+ \eta^2 \Gamma^{-1\,a}_{\;\;r_{i-1}} \Gamma^{-1\,r'_{i-1}}_{\;\;a} \mathrm{Id}^{m'_i}_{m_i} \mathrm{Id}^{r'_i}_{r_i} \\
&+ \eta \mathrm{Id}^{r'_{i-1}}_{r_{i-1}} \mathrm{Id}^{m'_i}_{m_i} \mathrm{Id}^{r'_i}_{r_i}
\end{aligned}
$$

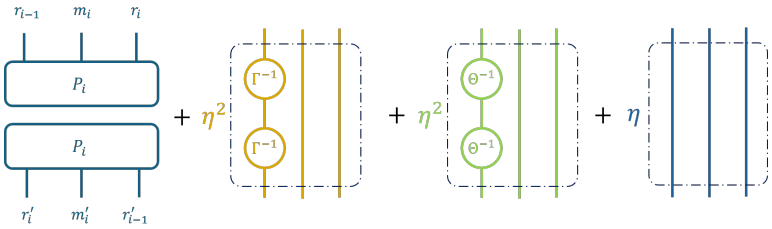


Figure 12: Regularised operator

5 Solving PDEs with Tensor Networks

5.1 Numerical approximation of BSDEs

We have seen in section 2 that a parabolic PDE can be transposed into a couple of stochastic equations as a general extension of the Feynman-Kac formula.

This leads to the following BFSDE [16]:

$$
\begin{aligned}
dX_t &= b(t, X_t)\mathrm{d}t + \sigma(t, X_t)\mathrm{d}W_t,\ t \geq T && (13)\\
X_0 &= x && (14)\\
\mathrm{d}Y_t &= -f(t, X_t, Y_t, Z_t)\mathrm{d}t + Z_t \mathrm{d}W_t,\ 0 \leq t \leq T && (15)\\
Y_T &= \Psi(X_T) && (16)
\end{aligned}
$$

with $Y_t = u(t, X_t), Z_t = D_x u(t, X_t)\sigma(X_t)$

In order to solve this formulation, it needs to be discretize opening the door for Monte Carlo algorithms and tensor train optimisation. Following the work of [18], let's discretize the coupled equations:

Let $0 = t_0 < t_1 < \cdots < t_n = T$ a decomposition of the time into an equal subdivision. The dynamic of the system can be simply discretized in a forward manner as

$$
X_{n+1} = X_n + b(t_n, X_n)\Delta t + \sigma(t_n, X_n)\xi_{n+1}\sqrt{\Delta t}
$$

with $\Delta t = t_{n+1} - t_n$ and $n \in \{0, \cdots, N-1\}$. The variables ξ_i, $i \in \{1, \cdots, N\}$ corresponds to random normal variables $\mathcal{N}(0, \mathrm{Id}_{d\times d})$.

As for the backward process $\mathrm{d}Y$, two versions can be considered that will play an important role in how stable our implementation ought to be.

- explicit discretization :

$$Y_{n+1} = Y_n - f(t_{n+1}, X_{n+1}, Y_{n+1}, Z_{n+1})\Delta t + Z_n \cdot \xi_{n+1}\sqrt{\Delta t}$$

- implicit discretization :

$$Y_{n+1} = Y_n - f(t_n, X_n, Y_n, Z_n)\Delta t + Z_n \cdot \xi_{n+1}\sqrt{\Delta t}$$

with both the terminal condition $Y_N = g(X_N)$.
Then, we can write for the explicit discretization:

$$\begin{aligned} Y_n &= Y_{n+1} + f(t_{n+1}, X_{n+1}, Y_{n+1}, Z_{n+1})\Delta t - Z_n \cdot \xi_{n+1}\sqrt{\Delta t} \\ &= \mathbb{E}\left[Y_{n+1} + f(t_{n+1}, X_{n+1}, Y_{n+1}, Z_{n+1})\Delta t | \mathcal{F}_n\right] \end{aligned}$$

By taking the conditional expectations w.r.t. to the σ-algebra generated by the discrete Brownian motion at time step n, denoted $\mathcal{F}_n$.

Then, by utilizing the fact that under the right assumptions, the conditional expectation is the orthogonal projection in L2 satisfying, $\forall B \in \mathrm{L}^2$:

$$\mathbb{E}\left[B|\mathcal{F}_n\right] = \underset{Y \in \mathrm{L}^2, \mathcal{F}_n - measurable}{\arg\min} \mathbb{E}\left[|Y - B|^2\right]$$

we get:

$$Y_n = \underset{Y \in \mathrm{L}^2, \mathcal{F}_n - measurable}{\arg\min} \mathbb{E}\left[|Y - Y_{n+1} - f(t_{n+1}, X_{n+1}, Y_{n+1}, Z_{n+1})\Delta t|^2\right]$$

By denoting $\hat{V}$, $\hat{X}$ and $\hat{Y}$, our approximation of V, X and Y respectively, and postulate that $\hat{V}_n(\hat{X}_n) \approx \hat{Y}_n \approx V(\hat{X}_n, t)$, we are are essentially trying to minimize the following quantity:

$$\mathbb{E}\left[|\hat{V}_n(\hat{X}_n) - \hat{V}_{n+1}(\hat{X}_{n+1}) - f(t_{n+1}, \hat{X}_{n+1}, \hat{Y}_{n+1}, \hat{Z}_{n+1})\Delta t|^2\right]$$

and in the same way for the implicit discretization:

$$\mathbb{E}\left[|\hat{V}_n(\hat{X}_n) - \hat{V}_{n+1}(\hat{X}_{n+1}) - f(t_n, \hat{X}_n, \hat{Y}_n, \hat{Z}_n)\Delta t + \sigma^T(\hat{X}_n, t_n)\nabla\hat{V}_n(\hat{X}_n)|^2\right]$$

5.2 Solving BSDEs with tensor trains

Building upon the previous results on the discretisation of the BFSDE, we use a tensor train at each time step to approximate the function V_n. The same way a neural network tries to learn a function from a set of inputs and outputs, here with a dataset (for the explicit scheme here) comprised of $\left(X_n^{(i)}, \hat{V}_{n+1}(\hat{X}_{n+1}^{(i)}) + f(t_{n+1}, \hat{X}_{n+1}^{(i)}, \hat{Y}_{n+1}^{(i)}, \hat{Z}_{n+1}^{(i)})\Delta t\right)_{i \in \{1, \cdots, N\}}$ where the outputs are fully computed from the previous time steps.

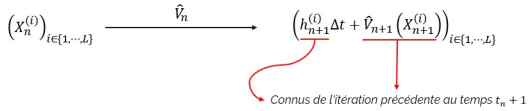


Figure 13: Dataset input to output

However, a critical step remains: the tensor train operates (multi)linearly while the function that we aim at approximating may not be linear. In this endeavor, we opt for using a feature map that will extend our vector of assets at time t_n, X_n, to a high-dimensional space generated by a set of basis functions $\{\phi_1, \cdots \phi_m\}$. Formally, $\hat{V} : \mathbb{R}^d \longrightarrow \mathbb{R}$ is expressed as:

$$\hat{V}(x_1, \cdots, x_d) = \sum_{i_1=1}^{m} \cdots \sum_{i_d=1}^{m} c_{i_1,\cdots i_d} \phi_{i_1}(x_1) \cdots \phi_{i_d}(x_d)$$

Throughout the paper, we refer to the number of feature components as the degree of our tensor train.

However, this formulation necessitates handling a tensor $c \in \mathbb{R}^{m\times m\times\cdots\times m} = \mathbb{R}^{m^d}$, wherein the number of parameters grows exponentially with dimensionality. To circumvent this challenge, we leverage the tensor train approach, effectively restructuring c as $u_1 \circ u_2 \circ \cdots \circ u_d$, where each u_i is a tensor of dimensionality three. Specifically, $u_i \in \mathbb{R}^{r_{i-1}\times m\times r_i}$, strategically containing significantly fewer parameters.

Using this tensor train and the formulation of the solver as an expectation minimization we can now leverage the previously seen algorithm for tensor train optimization by sampling samples of the discretized SDE and thus getting a PDE approximation. The simple pseudocode is presented in Algo 6.

Algorithm 6 PDE approximation

Require: Initial parametric choice for the functions V_n for $n \in \{0, \ldots, N-1\}$
Ensure: Approximation of $V(\cdot, t_n) \approx V_n$ along the trajectories for $n \in \{0, \ldots, N-1\}$

1: Simulate K samples of the discretized SDE.
2: Choose $V_N = g$.
3: **for** $n = N-1$ to 0 **do**
4: Approximate either the explicit or implicit expectation (both depending on V_{n+1}) using Monte Carlo
5: Minimize this quantity (explicitly or by iterative schemes)
6: Set V_n to be the minimizer
7: **end for**

For reference, the implicit iterative scheme is given in Algo 7.

Algorithm 7 Implicit iteration scheme

Require: Initial parametric choice for the functions V_n (typically equal to V_{n+1}), X_n, V_{n+1}
Ensure: Approximation of $V(\cdot, t_n) \approx V_n$
1: $V_n^k = V_n$
2: $Y_n^k = V_n^k(X_n)$
3: **for** $k = 0$ to $M - 1$ **do**
4: Compute $f(t_n, X_n, Y_n^k, Z_n^k)\Delta t + \sigma^T(X_n, t_n)\nabla V_n(X_n)$ using V_n^k and Y_n^k
5: Minimize for V_n:

$$\mathbb{E}\left[|V_n(X_n) - V_{n+1}(X_{n+1}) - f(t_n, X_n, Y_n^k, Z_n^k)\Delta t + \sigma^T(X_n, t_n)\nabla V_n(X_n)|^2\right]$$

6: Set V_n^k to be the minimizer
7: $Y_n^k = V_n^k(X_n)$
8: **end for**
9: $V_n = V_n^k$

6 Results

6.1 Tensor Network Library

The primary focus of the project involved the development of our own tensor network library, designed to streamline the management of tensor cores, creation of tensor networks, and optimized contraction of various tensor cores and networks.

One of the challenges encountered with tensors is their multidimensional nature, often involving axes that exceed the number of letters in the alphabet. This complexity can hinder the performance of dot products along the axes. While the einsum operand in most libraries is useful, it has limitations in terms of naming the axes, potentially resulting in code that is unclear and difficult to understand. To address this, we designed our own interface that extends numpy arrays, assigning a unique name to each axis. This approach enables easy transposition and efficient merging of axes, facilitating clarity and ease of understanding.

Furthermore, we replaced the conventional einsum function with a custom contract function. This function automatically sums over axes with the same name across tensor cores, eliminating the need for cluttered notations and streamlining the contraction process.

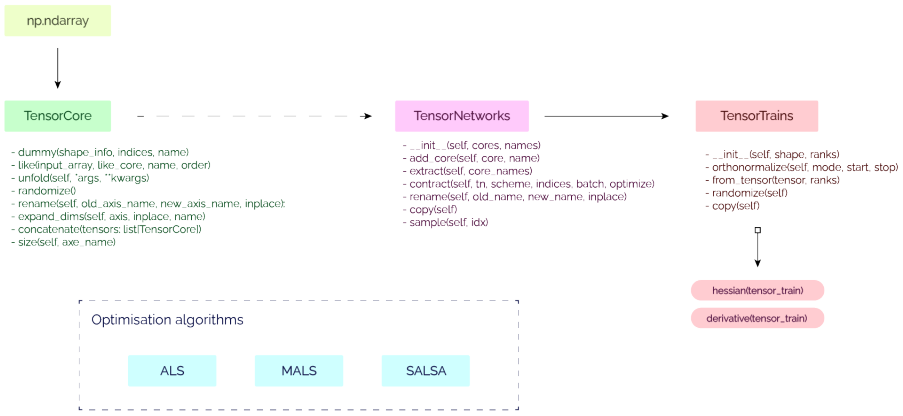


Figure 14: Simple UML of the tensor network library

These advancements enabled us to effortlessly implement the reduced density matrices, as explained earlier, along with various ALS algorithms and seamless pipelines. Importantly, we mitigated common errors associated with axis contraction that often arise when dealing with high-order tensors. This ensured robustness and accuracy throughout our implementations, facilitating smooth execution of complex tensor operations.

6.2 Degree importance & Reduced density matrix

In quantum physics and quantum information theory, a density matrix serves as a pivotal tool for describing the quantum state of a physical system. It facilitates the computation of probabilities associated with various measurement outcomes performed on the system, employing the Born rule. Conceptually, it is defined as the outer product of the wavefunction and its conjugate.

In the case of tensor trains, the density matrix corresponds to the outer product of the tensor train with itself, as depicted in Figure 15.

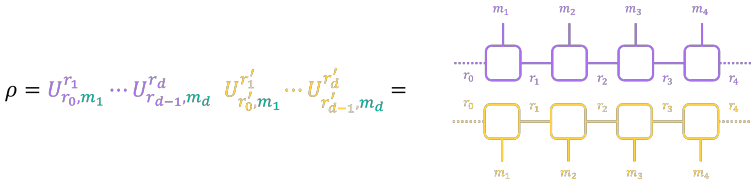


Figure 15: Density Matrix

This yields a tensor representation of the system with dimensions $\mathbb{R}^{m_1 \times \cdots \times m_d} \times \mathbb{R}^{m_1 \times \cdots \times m_d}$. Although this representation may exceed memory limitations,

we can compute a **reduced density matrix** via the partial trace operation, thereby selecting only a portion of the tensor train.

To discern the significance of individual cores, one can utilize the reduced density matrix corresponding to a specific core, as illustrated in Figure 16.

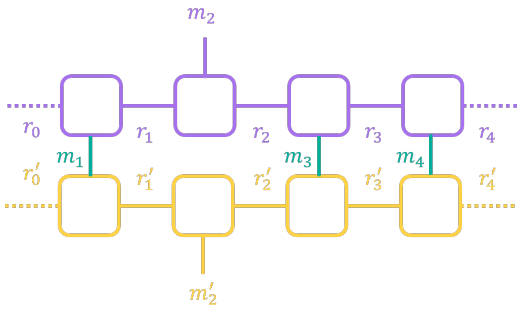


Figure 16: Reduced density matrix on core 2

This will result in a matrix of shape $m_2 \times m_2$ with diagonal element being the importance of each feature in the tensor train.

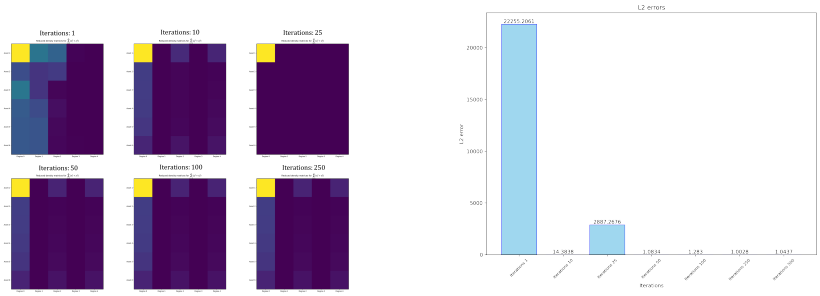


Figure 17: Feature importance with Reduced density matrix and L2 errors on ALS 6 assets for reconstruction of $\sum_i x_i + x_i^3$

Figure 17 demonstrates how features such as x^2 and x^4 contribute to the tensor train, while x and x^3 exhibit negligible contributions. Notably, the constant term consistently demonstrates significant activity, possibly serving as a residual term that catalyzes global information within the tensor, thereby nullifying each other. This is evident from the prominence of the top-left term, representing the constant term of the first core. In the ALS algorithm, upon completing an iteration, the tensor train assumes a left-orthonormal form, with the first core encapsulating residual information such as the norm, hence its elevated contribution.

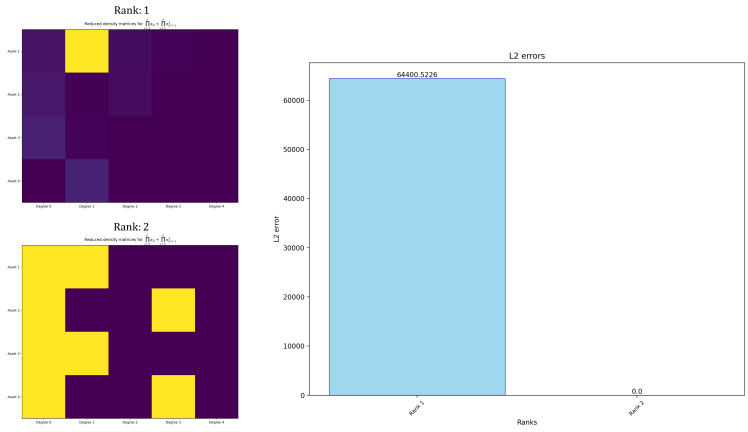


Figure 18: Feature importance for $\prod x_{2i} + \prod x_{2i+1}^3$ with intrication

Figure 18 unveils even more intriguing insights. The inner rank contributes to the mutual information transmitted from one core to another. Consequently, to facilitate interaction between multiple features (as observed in this scenario with products of different features from distinct cores), a tensor train with sufficient rank is imperative. This is evidenced by the inadequacy of a rank 1 tensor train to encompass all requisite information for optimization, whereas a rank 2 tensor train enables flawless reconstruction. Moreover, the features depicted reflect the relationship and computational utilization of features, with even terms exclusively associated with the x feature and odd terms solely attributed to the power of 3. A similar observation applies to the constant terms.

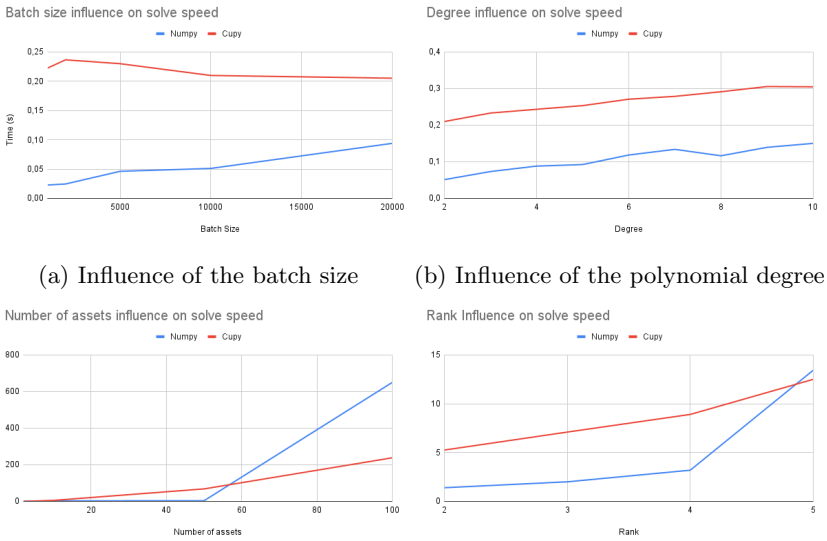
6.3 Comparison of a GPU implementation

One of the studies conducted in this project involved benchmarking the potential speed-up achieved by utilizing GPUs for various matrix operations. Transitioning from CPU to GPU implementation was relatively straightforward, facilitated by the compatibility between Numpy and Cupy, along with our own implementation of a tensor network library. To compare both implementations, we focused on benchmarking the ALS algorithm, primarily because it represents the main bottleneck in the entire pipeline. We opted not to benchmark SALSA due to its adaptive rank, which, while advantageous for performance, could introduce biased results as the network shape might inconsistently be modified during solving.

Consequently, we computed the average resolution time of the ALS algorithm for two different functions: the L2-norm of the features ($||X||_2^2$) and the log L2-norm of the features ($\log\big(0.5 + 0.5||X||_2^2\big)$) for varying sets of parameters.

The results depicted in Figure 19 illustrate the typical behavior of GPU implementation. The GPU-based implementation demonstrates superior scaling

compared to its CPU-based counterpart. However, despite this favorable scaling, in most tests conducted during this study, the CPU implementation outperformed the GPU, with the GPU only excelling in extremely high-dimensional setups. This discrepancy is further accentuated by the utilization of tensor train, as most matrix operations are exclusively applied on tensor cores, which are inherently of smaller dimensions. Moreover, both the pipeline and ALS algorithms are iterative processes, rendering parallelization impractical. Consequently, it becomes evident why GPU acceleration does not yield substantial speedup: the tensors are relatively small, and the operations are computed sequentially within the algorithms. As a result, the inherent parallelism of GPUs cannot be fully exploited in this context.



(a) Influence of the batch size

(b) Influence of the polynomial degree

(c) Influence of the number of assets (Dimension of the initial space)

(d) Influence of the rank of the tensor train

Figure 19: Benchmark: Cupy vs Numpy

6.4 Alternating Least Squares comparisons

The issue with the standard ALS algorithm is that it failed in being stable as the number of assets increases. This leads to an accumulation of errors that cannot be reduced. The SALSA variant, meanwhile, consists in regularisation terms and rank adaptation that act as a stabiliser.

In the setting of the reconstruction of a function such as the squared L2 norm (which is the terminal function of the Black-Scholes BFSDE), the ALS algorithm diverges as the number of assets augments while the SALSA is performing well as seen on the figure 20.

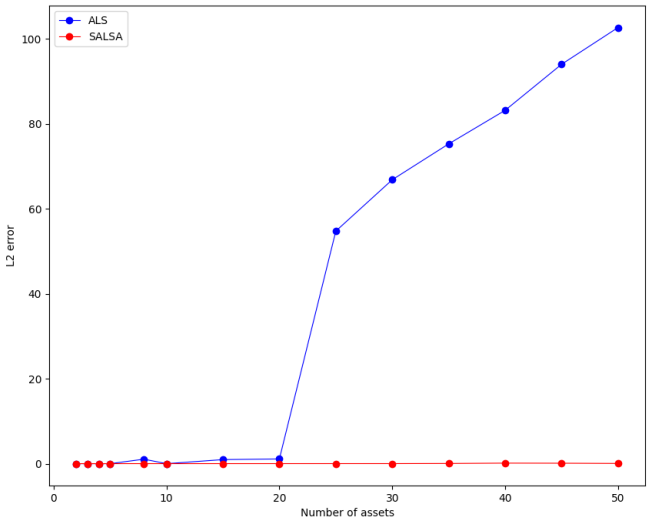


Figure 20: ALS vs SALSA errors in function of the number of assets

The previous figure was for a fixed number of ALS iterations (25). Now for a fixed number of assets, such as 25, if we vary the number of iterations we can see that the SALSA algorithm converges faster than the ALS one (see figure 21) while being more stable (the ALS algorithm somehow converges for 5 itertaions but diverges for 10 and 25).

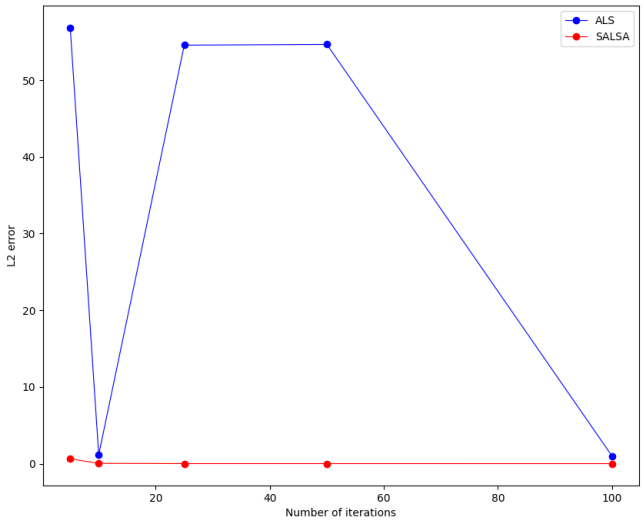


Figure 21: ALS vs SALSA errors in function of the number of iterations for 25 assets

As for the batch size, that is the number of samples that we want to fit, SALSA is more stable and can fit the function even with a very small number of samples while standard ALS fails to converge when there is not enough samples (figure 22).

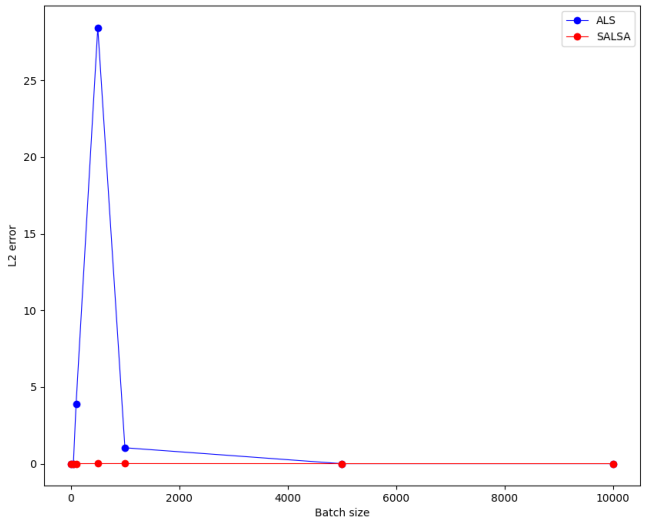


Figure 22: ALS vs SALSA errors in function of the number of samples for 20 assets

In these comparisons, regrettably, we were unable to incorporate the MALS algorithm. Divergences arose between the changes from the paper implementation and our specific use cases, preventing us from fully computing the given micro-optimization equations, which differed. Nonetheless, the results obtained from comparing ALS with SALSA underscore the superiority of employing an algorithm that adapts ranks and incorporates regularization based on singular values.

6.5 Explicit scheme: ALS vs SALSA

During our benchmarking process, it naturally became imperative to assess how both the ALS and SALSA algorithms would fare within the explicit pipeline, gauging whether the results aligned with those described in the previous subsection on simple function approximation.

In the next settings, we use the following BFSDE in the pipeline:

$$\begin{aligned} \mathrm{d}X_t &= \sigma \mathrm{diag}(X_t)\mathrm{d}W_t \\ X_0 &= \xi \\ \mathrm{d}Y_t &= r(Y_t - Z_t^\intercal X_t)\mathrm{d}t + \sigma Z_t^\intercal \mathrm{diag}(X_t)\mathrm{d}W_t \\ Y_T &= g(X_T) \end{aligned}$$

where $T = 1, \sigma = 0.4, r = 0.05, \xi = (1, 0.5, 1, 0.5, \cdots, 1, 0.5)$ and $g(x) = ||x||^2$. The couple of equations above correspond to the BFSDE of the following Black-Scholes-Barenblatt PDE:

$$u_t = -\frac{1}{2} Tr\left[\sigma^2 \text{diag}(X_t^2) D^2 u\right] + r(u - (Du)'x)$$

Which admits the explicit solution:

$$u(t, x) = \exp((r + \sigma^2)(T - t))g(x)$$

Utilizing this equation, with a batch size of 2000, 25 ALS iterations, and 15 time steps, Figures (23) and (24) exhibited outcomes akin to those observed in the preceding case, with the ALS algorithm's error escalating as the number of assets and degrees of features increased. However, it became apparent that instability set in earlier with smaller parameters compared to the simple approximation scenario. This phenomenon can be attributed to the complete pipeline comprising a sequence of ALS applications to each time step, with each subsequent step dependent on the outcome of the preceding ALS iteration. Consequently, errors accumulate more rapidly, leading to exacerbated errors for the ALS algorithm, whereas SALSA achieves stability through regularization and rank adaptation.

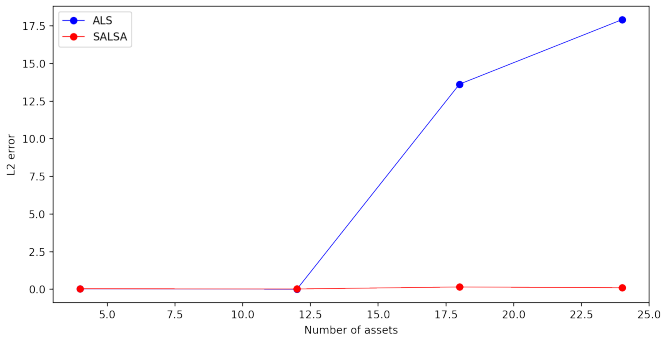


Figure 23: Influence of number of assets in the explicit pipeline for ALS & SALSA

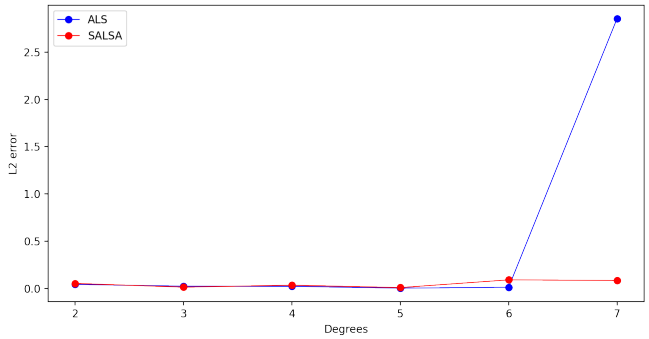


Figure 24: Influence of feature degrees in the explicit pipeline for ALS & SALSA

Additionally, in Figure 25, we present samples of trajectories across time steps for both training results obtained using ALS and SALSA, alongside their respective ground truth counterparts, which can be readily computed within the Black-Scholes framework. All trajectories originate from random points in space due to the inherent randomness of the Brownian motion dictated by the asset dynamics but converge to the same point corresponding to the PDE solution.

For both ALS and SALSA, the plots depict the offset between the ground truth dynamics and the approximation at each time step, for 4 and 16 assets. Notably, the plots are interpreted from right to left, as the evolution unfolds backward in time.

A direct inference drawn from these plots is that while the standard ALS algorithm adeptly captures the fluctuations of the curve, it exhibits a pronounced offset leading to instability that cannot be mitigated as the time steps decrease. Conversely, SALSA closely aligns with the curves, resulting in minimal errors on the yielded result at time step 0.

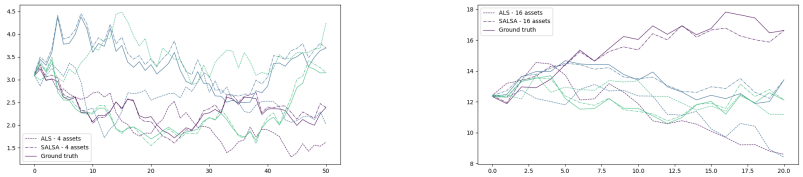


Figure 25: Sample of trajectories for 4 and 16 assets for the Black-Scholes PDE (ground truth, ALS, SALSA)

6.6 Comparing the explicit and implicit pipeline

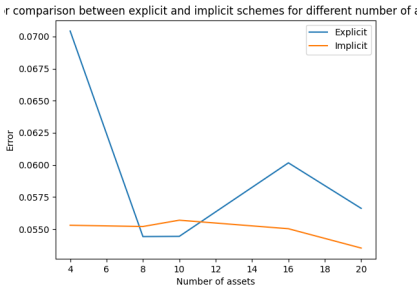
For the last experiment, we aimed to discern the disparities between implicit and explicit schemes. We reviewed the pipeline's performance under both schemes

while adjusting various parameters of our problem, including the number of assets, polynomial degree, batch size, and the number of iterations in the implicit solving scheme.

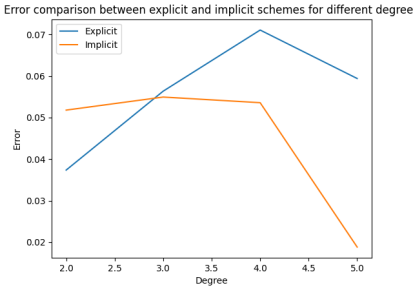
For reference, when parameters are not explicitly mentioned, the following default values were used: a batch size of 2000, a time horizon of 1, 4 assets, 50 iterations for SALSA, 50 iterations for implicit solving, a polynomial degree of 3, and an initial rank of 3.

Regarding the findings in Figure 26, particularly the initial two figures concerning the number of assets and polynomial degree, we observe a distinct advantage of the implicit scheme as the problem dimension increases. Notably, the graph illustrating the number of implicit iterations suggests that convergence is achieved with minimal iterations, typically around 10 iterations for this specific configuration. However, this graph also exposes a significant flaw in the explicit pipeline: its instability. Despite multiple runs with identical configurations, the explicit scheme demonstrates considerable disparity in results, contrary to the expected consistency.

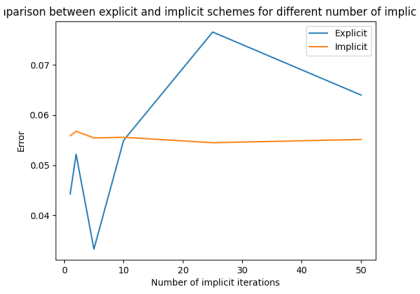
Lastly, the figure concerning batch size reveals little new insight, as both schemes exhibit similar behavior in response to this parameter adjustment.



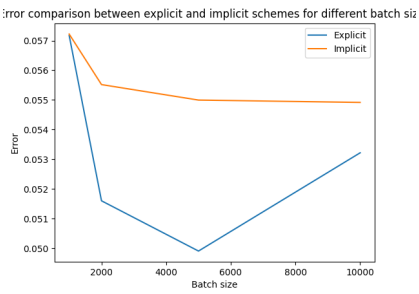
(a) Influence of the number of assets (Dimension of the initial space)



(b) Influence of the polynomial degree



(c) Influence of the number of implicit iterations



(d) Influence of the batch size

Figure 26: Benchmark: Explicit vs Implicit

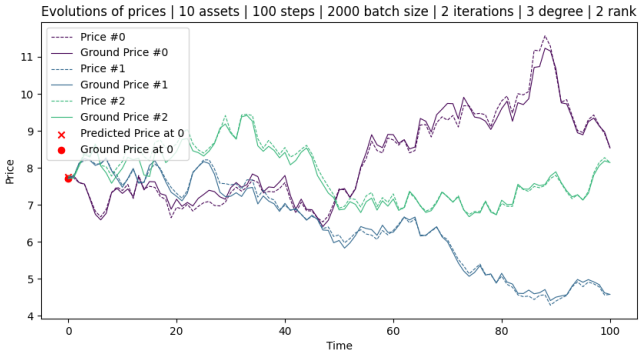


Figure 27: Example of solution on Black-Scholes

7 Conclusion

In conclusion, this project has been an extensive work encompassing both work and research. We successfully implemented numerous algorithms centered on tensor networks, thus enhancing our comprehension of tensor networks, PDEs, and their interconnection with BSDEs. Of particular significance is the development of a comprehensive tensor library. Without this crucial component, the implementation of various optimization algorithms would have been impractical, especially considering the limited availability of tensor framework libraries in Python.

This setup enabled us to conduct extensive benchmarking and testing of various optimization algorithms and configurations, facilitating a deeper exploration of the complexities inherent in utilizing tensor networks for PDE solutions. One noteworthy observation is the improved performance of most pipeline components as the dimensionality of the original space, particularly the number of assets, increases. This phenomenon, often referred to as the "blessing of dimensionality," has been previously recognized in literature. Although not fully comprehended, some insights from the theory of interacting particle systems suggest that the joint distribution of a large number of particles tends to approximately factorize as the number of particles increases, and that is plausible that similar approximate factorizations are relevant for highdimensional PDEs.

However, the current state of the project reveals some limitations, particularly in the meticulous tuning required for various hyperparameters. As with most machine learning pipelines, this tuning is crucial for ensuring convergence and avoiding potential divergence of solutions.

We would also like to highlight areas for improvement in our current pipeline and benchmarks:

- Implementing and comparing performance with other methods, such as neural networks, which are the primary comparison in the original paper, but seem to yield different results from paper to paper [17].

- Considering optimization of the code, although it was not the main focus of the work, we made efforts to maintain a clean and efficient codebase.
- Conducting extensive research to enhance the stability of the pipeline.
- Debugging the MALS algorithm and exploring other classical tensor network optimization algorithms.

A ALS subproblems derivation

Writing our tensor train based on its i-th core matrix notation ($V = L_i U_i R_i$), we aims at solving for each i, the following optimization least square problem:

$$\begin{aligned}(L_i U_i R_i - b)^2 &= 0 \\ &= L_i U_i U_i^T L_i^T - 2bL_i U_i R_i + b^2 = 0\end{aligned}$$

using the fact that our tensor train in right orthogonalised ($R_i R_i^T = I$).

Thus, from matrix calculus, cancelling out the derivative of this expression gives us:

$$\begin{array}{lll}\frac{\partial}{\partial U_i}\left(L_i U_i U_i^T L_i^T\right) & = 2U_i L_i L_i^T & = 2U_i \\ \frac{\partial}{\partial U_i}(2bL_i U_i R_i) & = 2bL_i^T R_i^T & \end{array}$$

So that,

$$\boxed{U_i = bL_i^T R_i^T}$$

B Option pricing and BFSDE

B.1 Basket call under the Bachelier model without drift

B.1.1 Bachelier model and derivation

Let d be the dimension of our asset, that is the number of assets considered in the option pricing. Under gaussian assumption, we have

$$\forall i \in \{1, \cdots, d\}, \quad \mathrm{d}S_t^i = \sigma^i \mathrm{d}W_t^i$$

where $\mathbf{W} = \begin{pmatrix} W_t^1 & \cdots & W_t^d \end{pmatrix}$ is a d-dimensional correlated Brownian motion with correlation :

$$\forall t > 0, \forall i, j \in \{1, \cdots, d\} \quad \mathrm{corr}\left(W_t^i, W_t^j\right) = \rho_{ij}$$

Introducing, an independant d-dimensional Brownian motion $\mathbf{B} = \begin{pmatrix} B_t^1 & \cdots & B_t^d \end{pmatrix}$ and the matrix of correlation $\mathbf{\Sigma} = (\rho_{ij})_{1 \geq i,j \leq d}$, we have $\mathbf{W}_t = \mathbf{L}\mathbf{B}_t$, with $\Sigma = LL^T$ using chelosky decomposition.

In matrix notation, with $\mathbf{S}_t = \begin{pmatrix} S_t^1 & \cdots & S_t^d \end{pmatrix}$, $\sigma = \begin{pmatrix} \sigma^1 & \cdots & \sigma^d \end{pmatrix}$,

$$\mathrm{d}\mathbf{S}_t = \sigma \cdot \mathrm{d}\mathbf{W}_t = (\sigma \cdot \mathbf{L})\, \mathrm{d}\mathbf{B}_t$$

The payoff for a Basket option is then for d assets :

$$H = (F_T - K)_+ \quad \text{with} \quad \forall t > 0,\, F_t = \frac{1}{d}\sum_{i=1}^{d} S_t^i$$

The mean of all assets at time t, F_t is a gaussian process with

$$\mathbb{E}[F_t] = 0$$

$$\mathbb{V}[F_t] = \frac{1}{d^2}\left(\sum_{i=1}^{d}\mathbb{V}[\sigma^i W_t^i] + \sum_{i,j=1\ i\neq j}^{d}\operatorname{corr}\left(\sigma^i W_t^i, \sigma^j W_t^j\right)\right)$$
$$= \frac{1}{d^2}\left(\sum_{i=1}^{d}{\sigma^i}^2 t + \sum_{i,j=1\ i\neq j}^{d}\sigma^i\sigma^j\rho_{ij}t\right)$$

Hence,

$$F_t \sim \mathcal{N}\left(0, \frac{\sqrt{t}}{d}\sqrt{\sigma^\top\Sigma\sigma}\right)$$

The problem reduced itself from a d-dimensional to a one dimensional one :

$$\mathrm{d}F_t = \pi \mathrm{d}W_t$$

with $\pi = \frac{1}{d}\sqrt{\sigma^\top\Sigma\sigma}$ and W a one dimensional Brownian motion.

With this knowledge, we can find the price of the option using the Bachelier formula in one dimension.

The actual payoff is for a European call $C_T = (F_T - K)_+$ for a maturity K leading to the Bachelier fundamental principle at $t = 0$, $C_0 = \mathbb{E}\left[(F_T - K)_+\right]$.

This boils down to :

$$\boxed{C_0 = (F_0 - K)\,\Phi\left(\frac{F_0 - K}{\sigma\sqrt{T}}\right) + \sigma\sqrt{T}\phi\left(\frac{F_0 - K}{\sigma\sqrt{T}}\right)}$$

where Φ is the cumulative distribution function of the standard gaussian $\mathcal{N}(0,1)$ and ϕ its probability density.

B.1.2 Partial differential equation and Backward SDE

From $\mathrm{d}\mathbf{S}_t = \sigma \cdot \mathrm{d}\mathbf{W}_t$, we aim at finding the associated differential formulation of $\mathbf{Y}_t = V(\mathbf{S}_t, t)$. Applying Itô's lemma gives the following :

$$\begin{aligned}
\mathrm{d}\mathbf{Y}_t &= \mathrm{d}V(\mathbf{S}_t, t) \\
&= \frac{\partial V}{\partial t}\mathrm{d}t + \sum_{i=1}^{d}\frac{\partial V}{\partial x_i}dS_t^i + \frac{1}{2}\sum_{i,j=1}^{d}\frac{\partial^2 V}{\partial x_i \partial x_j}\mathrm{d}\langle S^i, S^j\rangle_t \\
&= \frac{\partial V}{\partial t}\mathrm{d}t + \sum_{i=1}^{d}\sigma^i\frac{\partial V}{\partial x_i}dW_t^i + \frac{1}{2}\sum_{i,j=1}^{d}\sigma^i\sigma^j\rho_{ij}\frac{\partial^2 V}{\partial x_i \partial x_j}\mathrm{d}t \\
&= \left(\frac{\partial V}{\partial t} + \frac{1}{2}\operatorname{Tr}\left(\left(\sigma\cdot\mathbf{\Sigma}\right)^\top H_V\left(\sigma\cdot\mathbf{\Sigma}\right)\right)\right)\mathrm{d}t + \left(\boldsymbol{\nabla}V\right)^\top\left(\sigma\cdot\mathbf{\Sigma}\right)\mathrm{d}\mathbf{B}_t
\end{aligned}$$

where $H_V = \left(\frac{\partial^2 V}{\partial x_i \partial x_j}\right)_{1\leq i,j\leq d}$ is the Hessian of V.

Noting $\mathbf{\Pi} = (\sigma\cdot\mathbf{\Sigma})$, we have the following Forward-Backward SDE system (with $\mathbf{S} = \mathbf{X}$ for standard notation):

$$\begin{cases} \mathrm{d}\mathbf{X}_t &= \mathbf{\Pi}\mathrm{d}\mathbf{B}_t \\ \mathbf{X}_0 &= \mathbf{x}_0 \\ \mathrm{d}\mathbf{Y}_t &= \left(\frac{\partial V}{\partial t} + \frac{1}{2}\operatorname{Tr}\left(\mathbf{\Pi}^\top H_V \mathbf{\Pi}\right)\right)\mathrm{d}t + \left(\boldsymbol{\nabla} V\right)^\top \mathbf{\Pi}\mathrm{d}\mathbf{B}_t \\ \mathbf{Y}_T &= g\left(\mathbf{X}_T\right) \end{cases}$$

The associated parabolic PDE is given by :

$$\boxed{\frac{\partial V}{\partial t} + \frac{1}{2}\operatorname{Tr}\left(\mathbf{\Pi}^\top H_V \mathbf{\Pi}\right) = 0}$$

This is a Fokker-Planck equation with diffusion tensor $\mathbf{D} = \frac{1}{2}\Pi^\top \Pi$.

$$\boxed{\frac{\partial V}{\partial t} + \operatorname{Tr}\left(\mathbf{D} H_V\right) = 0}$$

References

[1] Jacob C Bridgeman and Christopher T Chubb. Hand-waving and interpretive dance: an introductory course on tensor networks. *Journal of physics A: Mathematical and theoretical*, 50(22):223001, 2017.

[2] Lukasz Cincio, Jacek Dziarmaga, and Marek M Rams. Multiscale entanglement renormalization ansatz in two dimensions: quantum ising model. *Physical review letters*, 100(24):240603, 2008.

[3] Pierre Comon, Xavier Luciani, and André LF De Almeida. Tensor decompositions, alternating least squares and other tales. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 23(7-8):393–405, 2009.

[4] Lawrence C Evans. *Partial differential equations*, volume 19. American Mathematical Society, 2022.

[5] Mark Fannes, Bruno Nachtergaele, and Reinhard F Werner. Finitely correlated states on quantum spin chains. *Communications in mathematical physics*, 144:443–490, 1992.

[6] Lars Grasedyck, Melanie Kluge, and Sebastian Krämer. Alternating least squares tensor completion in the tt-format. *arXiv preprint arXiv:1509.00311*, 2015.

[7] Lars Grasedyck and Sebastian Krämer. Stable als approximation in the tt-format for rank-adaptive tensor completion. *Numerische Mathematik*, 143(4):855–904, 2019.

[8] Sebastian Holtz, Thorsten Rohwedder, and Reinhold Schneider. The alternating linear scheme for tensor optimization in the tensor train format. *SIAM Journal on Scientific Computing*, 34(2):A683–A713, 2012.

[9] Côme Huré, Huyên Pham, and Xavier Warin. Deep backward schemes for high-dimensional nonlinear pdes. *Mathematics of Computation*, 89(324):1547–1579, 2020.

[10] Ian P McCulloch. From density-matrix renormalization group to matrix product states. *Journal of Statistical Mechanics: Theory and Experiment*, 2007(10):P10014, 2007.

[11] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of physics*, 349:117–158, 2014.

[12] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[13] Sebastian Paeckel, Thomas Köhler, Andreas Swoboda, Salvatore R Manmana, Ulrich Schollwöck, and Claudius Hubig. Time-evolution methods for matrix-product states. *Annals of Physics*, 411:167998, 2019.

[14] Roger Penrose et al. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications*, 1:221–244, 1971.

[15] David Perez-Garcia, Frank Verstraete, Michael M Wolf, and J Ignacio Cirac. Matrix product state representations. *arXiv preprint quant-ph/0608197*, 2006.

[16] Nicolas Perkowski. Backward stochastic differential equations: An introduction. *Available on semanticscholar. org*, 2011.

[17] Maziar Raissi. Forward–backward stochastic neural networks: deep learning of high-dimensional partial differential equations. In *Peter Carr Gedenkschrift: Research Advances in Mathematical Finance*, pages 637–655. World Scientific, 2024.

[18] Lorenz Richter, Leon Sallandt, and Nikolas Nüsken. Solving high-dimensional parabolic pdes using the tensor train format. In *International Conference on Machine Learning*, pages 8998–9009. PMLR, 2021.

[19] Ulrich Schollwöck. The density-matrix renormalization group in the age of matrix product states. *Annals of physics*, 326(1):96–192, 2011.

[20] Y-Y Shi, L-M Duan, and Guifre Vidal. Classical simulation of quantum many-body systems with a tree tensor network. *Physical review a*, 74(2):022320, 2006.

[21] Gábor Takács and Domonkos Tikk. Alternating least squares for personalized ranking. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 83–90, 2012.

[22] Frank Verstraete and J Ignacio Cirac. Renormalization algorithms for quantum-many body systems in two and higher dimensions. *arXiv preprint cond-mat/0407066*, 2004.

[23] Frank Verstraete and J Ignacio Cirac. Valence-bond states for quantum computation. *Physical Review A*, 70(6):060302, 2004.