Annals of Mathematical Sciences and Applications
Volume 8, Number 3, 427–460, 2023

A fast multilevel dimension iteration algorithm for high dimensional numerical integration

Xiaobing Feng* and Huicong Zhong

Dedicated to the memory of Professor Roland Glowinski

In this paper we propose and study a fast multilevel dimension iteration (MDI) algorithm for computing arbitrary d-dimensional integrals based on the tensor product approximations. It reduces the computational complexity (in terms of the CPU time) of a tensor product method from the exponential order $O(N^d)$ to the polynomial order $O(d^3N^2)$ or better, where N stands for the number of quadrature points in each coordinate direction. As a result, the proposed MDI algorithm effectively circumvents the curse of the dimensionality of tensor product methods for high dimensional numerical integration. The main idea of the proposed MDI algorithm is to compute the function evaluations at all integration points in cluster and iteratively along each coordinate direction, so lots of computations for function evaluations can be reused in each iteration. This idea is also applicable to any quadrature rule whose integration points have a lattice-like structure.

AMS 2000 subject classifications: Primary 65D30, 65D40; secondary 65C05, 65N99.
Keywords and phrases: Multilevel dimension iteration (MDI), high dimensional integration, numerical quadrature rules, tensor product methods, Monte Carlo methods.

1. Introduction

Numerical integration is one of the most fundamental building blocks in computational mathematics and in computational science at large. Many numerical methods (or quadrature rules) had been well developed as documented in numerical analysis textbooks (cf. [2, 16] and the references therein). They

arXiv: 2210.13658
*The work of this author was partially supported by the NSF grants: DMS-2012414 and DMS-2309626.

are tensor-product-type methods and work very well for computing integration in low dimensions. However, they all become prohibitively expensive in the high-dimensional case because the number of function evaluations grows as $O(N^d)$ (hence, the CPU time for computing them is expected to grow even faster), where d and N denote respectively the dimension of the integration domain and the number of the quadrature points in each coordinate direction. This exponential growth of the computational complexity is known as "the curse of the dimensionality" (cf. [1, 5]).

To circumvent or lessen "the curse of the dimensionality", various improved numerical integration methods have been developed in the literature. Among them are the Monte Carlo (MC) method and its variants including quasi- and multilevel Monte Carlo (QMC) methods [3, 5, 10, 11, 14, 5, 12, 17], and sparse grid (SG) methods [1, 8, 6], and deep neural network-based methods [7, 9, 13, 15, 18]. Although those improved methods are fundamentally different in their design ideas and mechanisms, they share a common strategy that is to reduce the number function evaluations compared to the tensor-product methods while maintaining the reasonable degree of accuracy. As expected, such a strategy works to some extent for medium dimensions (i.e., $d \approx 100$), but may not work for very high dimensions (i.e., $d \approx 1000$) because the required number of function evaluations still grows very rapidly for large d. The situation is even direr if one wants to solve partial differential equations (PDEs) in high dimensions. Consequently, developing faster and efficient numerical integration methods remains critical for tackling more challenging problems arising from mathematical finance, image processing, economics and data science.

The primary goal of this paper is to develop a fast algorithm, called *multilevel dimension iteration (MDI)*, for high dimensional numerical integration. Unlike the MC, QMC, SG and deep neural network (DNN) methods, the proposed MDI algorithm is not aiming to providing a new numerical integration method per se, instead, it is an acceleration algorithm for an efficient implementation of any tensor-product-type method. Thus, the MDI is not a "discretization" method but a "solver" (borrowing the numerical PDE terminologies). A well suited analogy would be high order polynomial evaluations, that is, to compute $p_0 := p(x_0) = a_k x_0^k + a_{k-1} x_0^{k-1} + \cdots + a_1 x_0 + a_0$ for a given real number input x_0. It is well known that such a high order polynomial evaluation on a computer is notoriously unstable, inaccurate (due to roundoff errors) and expensive, however, those difficulties can be easily overcome by a simple nested iteration (or Horner's algorithm. cf. [2, 16]), namely, set $p_0 := a_k$ and for $j = k, k-1, \ldots, 1$, compute $p_0 := p_0 x_0 + a_{j-1}$.

From the cost saving and efficiency point view, the reason for the nested iteration to be efficient and fast is that it reuses many multiplications involving x_0 compared to the direct evaluations of each term in $p(x_0)$. Conceptually, this is exactly the approach adopted by the proposed MDI algorithm, i.e., to reuse computations of the function evaluations in a tensor product method as much as possible to save the computation cost and hence to make it efficient and fast. A key observation is that the function evaluations of every tensor product method (including SG) involve a lot of computation in each coordinate direction which can be shared because each coordinate ξ_j of every integration point $\xi = (\xi_1, \xi_2, \ldots, \xi_d) \in \mathbb{R}^d$ is shared by many other integration points due to their tensor product structure. This observation motivates us to compute the required function evaluations in cluster and iteratively in each coordinate direction instead of computing them at the integration points independently, which is exactly the key idea of the proposed MDI algorithm. In other words, our MDI algorithm is based on a numerical Fubini's approach to compute the summation (and function evaluations) defined by a given tensor product method.

The remainder of the paper is organized as follows. In Section 2 we introduce our MDI algorithm first in two and three dimensions to explain the main ideas of the algorithm, and then generalize them to arbitrary dimensions. In Section 3 we present various numerical experiments to test the performance of the proposed MDI algorithm and to do various performance comparisons with the classical MC method. It shows that the MDI algorithm (regardless the choice of the underlying tensor product method) is faster than the classical MC method in low and medium dimensions (i.e., $d \approx 100$), much faster in very high dimensions (i.e., $d \approx 1000$), and succeeds even when the MC method fails. In Section 4 we provide numerical experiments to gauge the influence of parameters in the proposed MDI algorithm, including the dependence on choices of the underlying tensor product method and the iteration step size. In Section 5, we use the computation techniques to find out the computational complexity of the MDI algorithm. This is done by discovering the relationship between CPU time and dimension by the standard regression technique. It shows that the CPU time grows in the polynomial order $O(d^3N^2)$ at most. Furthermore, numerical experiments are designed to test the limit of the proposed MDI algorithm, it can compute integrals on standard desktop computers in medium dimensions easily and in very high dimensions quickly. Finally, we complete the paper with a few concluding remarks given in Section 6.

2. Methodology: the MDI algorithm

Let $\Omega \subset \mathbb{R}^d$ $(d \geq 1)$ be a bounded rectangular domain. Without loss of the generality, unless mentioned otherwise, we assume $\Omega = [0,1]^d$. Let $g : \overline{\Omega} \to \mathbb{R}$ denote a generic continuous function on $\overline{\Omega}$ (hence, g has a pointwise value at every $x = (x_1, x_2, \ldots, x_d) \in \overline{\Omega}$). Then the central issue to be addressed in this paper is to evaluate

$$I(g,\Omega) := \int_\Omega g(x)\,dx \tag{1}$$

accurately and efficiently for $d \gg 1$.

We like to note that numerically evaluating integral (1) is imperative in many applications. For example, when we solve random/stochastic PDEs, we must compute quantities of stochastic interests (QoSIs) such as mean, variance and high moments. All these QoSIs involve numerical integration, which is must be done in high dimensions if the sample space is high-dimensional. Moreover, when we solve high-dimensional linear PDEs using Green's function method or using the probability method, both approaches require computing high-dimensional integrals to obtain the approximate solutions. Finally, in the financial mathematics, when modeling an option of a basket of securities, a high-dimensional Black-Scholes model must be solved, which also requires one way or another to compute high-dimensional integrals.

2.1. Tensor product methods

In this subsection, we briefly recall the formulation of tensor product methods for approximating (1) and their well-known properties.

Let N be a positive integer and $h = \frac{1}{N}$. Let $\mathcal{T}_h = \{K\}$ denote the uniform rectangular mesh of Ω with mesh size h. Note that each element $K \in \mathcal{T}_h$ is a d-rectangle (in fact, a d-square of side h when Ω is a d-square domain) and the total number of elements in $\mathcal{T}_h$ is N^d. Define for $K \in \mathcal{T}_h$

$$I(g,K) := \int_K g(x)\,dx. \tag{2}$$

By the summation property of integrals, we have

$$I(g,\Omega) = \sum_{K\in\mathcal{T}_h} I(g,K). \tag{3}$$

Now, approximating every $I(g, K)$ by a local numerical quadrature rule

$$J(g, K) \approx I(g, K) \qquad \forall K \in \mathcal{T}_h, \tag{4}$$

it then leads to the following global approximation

$$J(g, \Omega) := \sum_{K \in \mathcal{T}_h} J(g, K) \approx \sum_{K \in \mathcal{T}_h} I(g, K) = I(g, \Omega), \tag{5}$$

which is known as the composite method of the chosen local quadrature rule.

When the local rule (4) is constructed by repeatedly using the same one-dimensional quadrature rule in each coordinate direction, then the resulting global rule (5) is called a tensor product quadrature rule for approximating $I(g, \Omega)$.

In this paper we only focus on the following four popular tensor product quadrature rules: (i) the trapezoidal rule; (ii) the Simpson's rule; (iii) the two-point Gaussian rule; (iv) the midpoint rule. It is well known [2, 16] that these four rules have the following error estimates:

$$\text{(i) } I(g, \Omega) - J_{\text{trap}}(g, \Omega) = O(N^{-2}), \qquad \text{(ii) } I(g, \Omega) - J_{\text{simp}}(g, \Omega) = O(N^{-4}),$$
$$\text{(iii) } I(g, \Omega) - J_{\text{gauss}}(g, \Omega) = O(N^{-4}), \qquad \text{(iv) } I(g, \Omega) - J_{\text{midp}}(g, \Omega) = O(N^{-2}).$$

Clearly, these error estimates hold only when g is sufficiently smooth. Also, the trapezoidal rule is lower order compared to the Simpson and two-point Gaussian rules, however, the midpoint, trapezoidal and Simpson rules are easier to implement and flexible on choosing the integration points when the step size h is not fixed. This feature is advantageous in the case when explicit formula of the integrand g is not known. On the other hand, the two-point Gaussian rule has higher accuracy at the expense of computing the Gaussian points (which is costly when $d \gg 1$). In Section 4 we shall use numerical experiments to further elaborate this point in the context of the proposed MDI algorithm.

2.2. Formulation of the MDI algorithm in two dimensions

To better understand and to present the idea of the MDI algorithm, we first consider the simple two dimensional case (i.e., $d = 2$ and $\Omega = [0, 1]^2$). In the two-dimensional case, by Fubini's Theorem we have

$$I\big(g, [0, 1]^2\big) := \int_{[0,1]^2} g(x)\, dx = \int_0^1 \left(\int_0^1 g(x)\, dx_1 \right) dx_2. \tag{6}$$

It should be noted that the exact evaluation of the above integral (assuming it is doable) by hand is often done using Fubini's theorem in calculus. Conceptually, this trivial fact will play an important role in conceiving the idea of our MDI algorithm.

Let $N \geq 1$ be an integer. Suppose that we have (or choose) the following generic one-dimensional quadrature rule:

$$\int_0^1 \phi(s)\,ds \approx \sum_{i=1}^{N} w_i \phi(\xi_i) =: J(g,[0,1]), \tag{7}$$

where $\{\xi_i\}$ and $\{w_i\}$ denote respectively the nodes and weights of the quadrature rule. ϕ is a generic (continuous) function on $[0,1]$.

Then, the corresponding two-dimensional tensor product rule takes the form

$$J\big(g,[0,1]^2\big) = \sum_{i,j=1}^{N} w_i w_j g(\xi_{ij}) \tag{8}$$

where the two-dimensional nodes $\xi_{ij} := (\xi_i, \xi_j)$. Obviously, the computational complexity of the above quadrature rule is $O(N^2)$

Motivated by (and mimicking) the Fubini's formula (6), we rewrite the tensor product rule (8) as

$$J\big(g,[0,1]^2\big) = \sum_{j=1}^{N} w_j \bigg(\sum_{i=1}^{N} w_i g(\xi_{ij}) \bigg) = \sum_{j=1}^{N} w_j g_1(\xi_j), \tag{9}$$

where

$$g_1(s) := \sum_{i=1}^{N} w_i g\big((\xi_i, s)\big). \tag{10}$$

We note that the evaluation of $g_1(\xi_j)$ is amount to applying the 1-d formula (7) to approximate the integral $\int_0^1 g\big((x_1,\xi_j)\big)\,dx_1$. However, the values of $\{g_1(\xi_j)\}$ will not be computed by the 1-d quadrature rule in our MDI algorithm, instead, g_1 is formed as a symbolic function, so the 1-d quadrature rule can be called again.

Evidently, (9) is a discrete analogue of the Fubini's formula (7), hence, we refer (9) as a *discrete Fubini's formula* in the rest of this paper. This simple formula has a significant computational benefit because it does all computations which involve the first (i.e., x_1) components of all two-dimensional

integration nodes/points first and save them in terms of the symbolic function g_1 defined by (10), then the final function evaluations in the tensor product formula are done by evaluating g_1 at the second (i.e., x_2) component of all two-dimensional integration nodes/points.

Let W and X denote the weight and node vectors of a selected 1-d numerical quadrature rule on $[0,1]$ and we use a parameter r to indicate the quadrature rule. The following algorithm implements the above discrete Fubini's formula.

Algorithm 1 2d-MDI(g, Ω, N, r)

Inputs: g, Ω, N, r.
Output: $J = J(g, \Omega)$.

1: Initialize $g_1 = 0$, $J = 0$.
2: **for** $i = 1 : N$ **do**
3: $\quad g_1 = g_1 + W(i)g((X(i), \cdot))$.
4: **end for**
5: **for** $j = 1 : N$ **do**
6: $\quad J = J + W(j)g_1(X(j))$.
7: **end for**
8: **return** J.

We note that the first do-loop forms the symbolic function g_1 which encodes all computations involving the x_1-components of all integration points. The second do-loop evaluates the 1-d quadrature rule for the function g_1. As mentioned above, in this paper we only focus on the four well-known 1-d quadrature rules: (i) the trapezoidal rule; (ii) the Simpson's rule; (iii) the two-point Gaussian rule; (iv) the midpoint rule. They will be represented respectively by $r = 1, 2, 3, 4$.

Below we use a concrete 2-d example to explain the mechanism of above 2d-MDI algorithm. It is clear that to directly compute the Tensor Product (TP) sum

$$\sum_{i_1=1}^{N} \sum_{i_2=1}^{N} \omega_{i_1} \omega_{i_2} g(\xi_{i_1}, \xi_{i_2}) \tag{11}$$

it is necessary to compute the function values of $g(x_1, x_2)$ at $n_1 n_2$ points, which are often done independently. On the other hand, the 2d-MDI algorithm is based on rewriting the sum as

$$\sum_{i_1=1}^{N} \sum_{i_2=1}^{N} \omega_{i_1} \omega_{i_2} g(\xi_{i_1}, \xi_{i_2}) = \sum_{i_1=1}^{N} \omega_{i_1} g_1(\xi_{i_1}), \tag{12}$$

where $g_1(x_1) = \sum_{i_2}^{N} \omega_{i_2} g(x_1, \xi_{i_2})$ denotes the symbolic function obtained in the first do-loop. Hence, the algorithm performs two separate do-loops. In the first d-loop, symbolic computations are performed to obtain the symbolic function $g_1(x_1)$ which is saved. In the second do-loop, the single sum $\sum_{i_1=1}^{N} g_1(\xi_{i_1})$ is done. When computing the symbolic function $g_1(x_1)$, a lot of computations have been reused for computing the coefficients in $g_1(x_1)$, and those coefficients are constants in the second do-loop. Efficiently generating the symbolic function and using it to compute the TP sum are the main reasons of saving computation and computer memory.

Take $g(x_1, x_2) = x_1^2 + x_1x_2 + x_2^2$ as an example. The direct computation the TP sum in (11) requires to compute the function value $g(\xi_1, \xi_2) = \xi_1^2 + \xi_1\xi_2 + \xi_2^2$ at each node (ξ_1, ξ_2), this in turn requires three multiplications and two additions. With a total of N^2 nodes, then computing the sum requires *a total of* $5N^2$ *multiplications and* $4N^2 - 1$ *additions.* On the other hand, when using the 2d-MDI algorithm to compute the same sum, in the first do-loop, we compute the symbolic function $g_1(x_1) = \sum_{i_2=1}^{N} \omega_{i_2} g(x_1, \xi_{i_2})$ which requires N "symbolic multiplications" of $\xi_{i_2} x_1$ (no real multiplication is needed because of its linear dependence on ξ_{u_2}) and N multiplications of $\xi_{i_2}^2$, as well as $3N - 1$ additions. In the second do-loop, computing the sum $\sum_{i_1=1}^{N} \omega_{i_1} g_1(\xi_{i_1})$ requires N multiplications of $\xi_{i_1}^2$ and N multiplications of $\xi_{i_1}\xi_{i_2}$, as well as $3N - 1$ additions. Thus, the 2d-MDI algorithm requires *a total of* $8N$ *multiplications and* $6N - 4$ *additions.* Therefore, the 2d-MDI algorithm computes the TP sum much cheaper than the standard implementation, and this advantage will become more significant in high dimensions. See Sections 2.4 and 3 for details.

2.3. Formulation of the MDI algorithm in three dimensions

In the subsection we shall formulate the MDI algorithm in the 3-d case. Since the main idea is similar to that of the 2-d case, we shall only highlight its main steps.

Applying the 1-d quadrature rule (7) in each of three coordinate directions, we readily obtain the following 3-d tensor product rule for approximating integral $I(g, [0,1]^3)$:

$$J\big(g, [0,1]^3\big) = \sum_{i,j,k=1}^{N} w_i w_j w_k g(\xi_{ijk}) \tag{13}$$

where the three-dimensional nodes $\xi_{ijk} := (\xi_i, \xi_j, \xi_k)$. Obviously, the computational complexity of the above formula is $O(N^3)$

Again, by Fubini's Theorem we have

$$I\big(g,[0,1]^3\big)=\int_{[0,1]^3} g(x)\,dx=\int_{[0,1]^2}\left(\int_0^1 g(x)\,dx_1\right)dx', \tag{14}$$

where $x'=(x_2,x_3)$. Mimicking the above Fubini's formula, we rewrite (13) as

$$J\big(g,[0,1]^3\big)=\sum_{j,k=1}^{N} w_j w_k\left(\sum_{i=1}^{N} w_i g(\xi_{ijk})\right)=\sum_{j,k=1}^{N} w_j w_k g_2(\xi_j,\xi_k), \tag{15}$$

where

$$g_2(s,t):=\sum_{i=1}^{N} w_i g((\xi_i,s,t)). \tag{16}$$

Once again, it should be noted that g_2 will be formed as a symbolic function in our MDI algorithm and the right-hand side of (15) is viewed as a 2-d tensor product formula for g_2, it can be computed either directly or recursively by using **Algorithm 1**. Below we present our MDI algorithm for implementing the recursive strategy.

Algorithm 2 3d-MDI(g, Ω, N, r)

Inputs: g, Ω, N, r.
Output: $J = J(g, \Omega)$.

1: Initialize $g_2 = 0$, $J = 0$.
2: **for** $i = 1 : N$ **do**
3: $\quad g_2 = g_2 + W(i)g((X(i), \cdot, \cdot))$.
4: **end for**
5: $\Omega_2 = P_3^2\Omega$.
6: J =2d-MDI(g_2, Ω_2, N, r).
7: **return** J.

where P_3^2 denotes the orthogonal projection (or natural embedding): $x = (x_1, x_2, x_3) \to x' = (x_2, x_3)$, W and X stand for the weight and node vectors of the selected 1-d quadrature rule.

From **Algorithm 2** we already can see the procedure of the MDI algorithm. It is based on the two main ideas: (i) to use the discrete Fubini's formula to reduce the computation of the tensor product sum into the computation of a lower dimensional tensor product sums, which allow us to call

recursively a lower dimensional MDI algorithm; (ii) the function evaluations are done in cluster in each coordinate direction during the dimension iteration/reduction, which is the main reason for a significant computational saving due to reusing lots of computations, compared to the standard pointwise function evaluations which treat all the integration points independently and do not reuse any computation.

2.4. Formulation of the MDI algorithm in arbitrary d-dimensions

The goal of this subsection is to extend the 2- and 3-d MDI algorithms to arbitrary d-dimensions. To the end, we first recall a more general version of Fubini's Theorem stated as follows:

$$I(g,\Omega)=\int_{\Omega} g(x)\,dx=\int_{\Omega_{d-m}}\left(\int_{\Omega_m} g(x)\,dx''\right)dx', \tag{17}$$

where $1\le m<d$, $\Omega=[0,1]^d, \Omega_m=Q_d^m\Omega=[0,1]^m$ and $\Omega_{d-m}=P_d^{d-m}\Omega=[0,1]^{d-m}$ in which Q_d^m and P_d^{d-m} denote respectively the orthogonal projections (or natural embeddings): $x=(x_1,x_2,\dots,x_d)\to x''=(x_1,x_2,\dots,x_m)$ and $x=(x_1,x_2,\dots,x_d)\to x'=(x_{m+1},x_{m+2},\dots,x_d)$. The integer $1\le m\le 3$ is the dimension reduction step length in our algorithm. In Section 4, we shall demonstrate using numerical tests the optimal choice of step length m.

We also recall that the tensor product quadrature rule for $I(g,\Omega)$ is defined as

$$J(g,\Omega)=\sum_{i_1,i_2,\dots,i_d=1}^{N} w_{i_1}w_{i_2}\cdots w_{i_d}g(\xi_{i_1,\dots,x_{i_d}}). \tag{18}$$

Where $\{\xi_j\}_{j=1}^N$ and $\{w_j\}_{j=1}^N$ are the nodes and weights of the given 1-d quadrature rule (7), and $\xi_{i_1,\dots,x_{i_d}}=(\xi_{i_1},\xi_{i_1},\dots,\xi_{i_d})$. Clearly, the computational complexity of the above formula is $O(N^d)$.

Rewrite (18) as

$$\begin{aligned} J(g,\Omega)&=\sum_{i_{m+1},\dots,i_d=1}^{N} w_{i_{m+1}}w_{i_2}\cdots w_{i_d}\left(\sum_{i_1,\dots,i_m=1}^{N} w_{i_1}w_{i_2}\cdots w_{i_m}g(\xi_{i_1,\dots,x_{i_d}})\right)\\ &=\sum_{i_{m+1},\dots,i_d=1}^{N} w_{i_{m+1}}w_{i_2}\cdots w_{i_d}\,g_{d-m}\big(\xi_{i_{m+1}},\dots,\xi_{i_d}\big), \end{aligned} \tag{19}$$

where

(20)

$$g_{d-m}(s_1,\ldots,s_{d-m}) = \sum_{i_1,\ldots,i_m=1}^{N} w_{i_1} w_{i_2} \cdots w_{i_m}\, g\big((\xi_1,\ldots,\xi_m,s_1,\ldots,s_{d-m})\big).$$

We note that in our MDI algorithm g_{d-m} is formed as a symbolic function using (20) and the right-hand side of (19) is a $(d-m)$-order multi-summation, which itself can be evaluated by employing the above dimension reduction strategy. The reduction can be iterated $\ell := [\frac{d}{m}]$ times until $d - \ell m \le m$. Since $m \le 3$, the final sum can be evaluated by calling **Algorithm 1** or **2**. To realize this procedure, we introduce the following conventions.

- If $k = 1$, set $\mathrm{MDI}(k, g_k, \Omega_k, N, m, r) := J(g_k, \Omega_k)$, which is computed by using the one-dimensional quadrature rule (7).
- If $k = 2$, set $\mathrm{MDI}(k, g_k, \Omega_k, N, m, r) := \text{2d-MDI}(g_k, \Omega_k, N, r)$.
- If $k = 3$, set $\mathrm{MDI}(k, g_k, \Omega_k, N, m, r) := \text{3d-MDI}(g_k, \Omega_k, N, r)$.

We note that when $k = 1, 2, 3$, the parameter m becomes a dummy variable and can be given any value.

Let P_k^{k-m} denote the natural embedding from $\mathbb{R}^k$ to $\mathbb{R}^{k-m}$ by deleting the first m components of vectors in $\mathbb{R}^k$. Then the tensor product quadrature approximation $J(g, \Omega)$ with $\Omega = [0,1]^d$ can be computed efficiently as follows.

Algorithm 3 $\mathrm{MDI}(d, g, \Omega, N, m, r)$

Inputs: $d(\ge 4), g, \Omega, N, m(= 1, 2, 3), r$.
Output: $J = J(g, \Omega)$.

1: $\Omega_d = \Omega$, $g_d = g$, $\ell = [\frac{d}{m}]$.
2: **for** $k = d : -m : d - \ell m$ (the index is decreased by m at each iteration) **do**
3: $\quad \Omega_{d-m} = P_k^{k-m} \Omega_k$.
4: $\quad$ Construct symbolic function g_{k-m} by (21) below).
5: $\quad \mathrm{MDI}(k, g_k, \Omega_k, N, m, r) := \mathrm{MDI}(k-m, g_{k-m}, \Omega_{k-m}, N, m, r)$.
6: **end for**
7: $J = \mathrm{MDI}(d - \ell m, g_{d-\ell m}, \Omega_{d-\ell m}, N, m, r)$.
8: **return** J.

Where

(21)

$$g_{k-m}(s_1,\ldots,s_{k-m}) = \sum_{i_1,\ldots,i_m=1}^{N} w_{i_1} w_{i_2} \cdots w_{i_m}\, g_k\big((\xi_1,\ldots,\xi_m,s_1,\ldots,s_{k-m})\big).$$

Remark 2.1. **Algorithm 3** recursively generates a sequence of symbolic functions $\{g_d, g_{d-m}, g_{d-2m}, \ldots, g_{d-\ell m}\}$, each function has m fewer arguments than its predecessor. As already mentioned above, our MDI algorithm explores the lattice structure of the tensor product integration points, instead of evaluating function values at all integration points independently, the MDI evaluates them in cluster and iteratively along m-coordinate directions, the function evaluation at any integration point is not completed until the last step of the algorithm is executed. So many computations are reused in each iteration, which is the main reason for the computation saving and to achieve a faster algorithm. Clearly, this idea can be applied to other quadrature rules, including sparse grid methods, whose integration points have a lattice-like structure.

3. Numerical performance tests

In this section, we shall present extensive and purposely designed numerical experiments to gauge the performance of the proposed MDI algorithm and to compare it with the standard tensor product (STP) method and the classical Monte Carlo (MC) method for computing high dimensional integrals. All the numerical tests show that the MDI outperforms both TP and MC methods in low and medium dimensions (i.e., $d \approx 100$), and significantly outperforms them in very high dimensions (i.e., $d \approx 1000$), and succeeds even when the other two methods fail. We shall evaluate the influence of the choice of the 1-d base quadrature rule (indicated by the parameter r) and step length of the dimension iteration (indicated by the parameter m).

All our numerical experiments are done in Matlab 9.4.0.813654(R2018a) on a desktop PC with Intel(R) Xeon(R) Gold 6226R CPU 2.90 GHz and 32 GB RAM.

3.1. Two and three-dimensional tests

We first test our MDI on simple 2- and 3-d examples and to compare its performance (in terms of the CPU time) with the STP and MC methods. A word of warning is that due to small size of the problems and good accuracy of all the methods, the performance differences between of these methods may not be significant when the integrand g is very "nice". This is the reason that we shall use oscillatory or rapidly growing integrands which often require to use a large number of integration points to achieve high accuracy.

Test 1. Let $\Omega = [0,2]^2$ and consider the following 2-d integrands:

$$g(x) := \exp\big(5x_1^2 + 5x_2^2\big); \qquad \widehat{g}(x) := \sin\big(2\pi + 10x_1^2 + 5x_2^2\big). \tag{22}$$

Let $h \in (0,1)$ denote the grid size of the tensor product grid. Then the number of integration points in each coordinate direction is $N = \frac{2}{h} + 1$. The base 1-d quadrature rule is chosen to be the Simpson's rule, hence, $r = 2$. Its composite quadrature rule is denoted by STP-S which stands for the standard tensor product-Simpson rule.

Tables 1 and 2 present the computational results (errors and CPU times) of the STP-S and MDI methods for approximating $I(g,\Omega)$ and $I(\widehat{g},\Omega)$, respectively.

Table 1: Relative errors and CPU times of STP-S and MDI simulations with $m = 1$ for approximating $I(g,\Omega)$

		STP-S		MDI	
Mesh size (h)	Total nodes	Relative error	CPU time	Relative error	CPU time
0.1	441	1.2146×10^{-1}	0.0380032	1.2146×10^{-1}	0.1371068
0.05	1681	1.0222×10^{-2}	0.0438104	1.0222×10^{-2}	0.1857806
0.025	6561	7.0238×10^{-4}	0.0545541	7.0238×10^{-4}	0.3617802
0.0125	25921	4.5031×10^{-5}	0.0633071	4.5031×10^{-5}	0.5514163
0.0100	40401	1.8502×10^{-5}	0.0659092	1.8502×10^{-5}	0.6151638
0.00625	103041	2.8328×10^{-6}	0.0720637	2.8328×10^{-6}	0.8968891

Table 2: Relative errors and CPU times of STP-S and MDI simulations with $m = 1$ for approximating $I(\widehat{g},\Omega)$

		STP-S		MDI	
Mesh size (h)	Total nodes	Relative error	CPU time	Relative error	CPU time
0.1	441	8.4038×10^{-1}	0.0413903	8.4038×10^{-1}	0.1381629
0.05	1681	1.2825×10^{-2}	0.0477657	1.2825×10^{-2}	0.1841843
0.025	6561	5.1928×10^{-4}	0.0579602	5.1928×10^{-4}	0.2845160
0.0125	25921	2.9642×10^{-5}	0.0579613	2.9642×10^{-5}	0.4957854
0.0100	40401	1.2014×10^{-5}	0.0617214	1.2014×10^{-5}	0.6276175
0.00625	103041	1.8123×10^{-6}	0.0708075	1.8123×10^{-6}	0.9682539
0.003125	410881	1.1213×10^{-7}	0.0946567	1.1213×10^{-7}	2.1355674

From Table 1 and 2, we observe that the CPU times used by these two methods are very small although that of the STP-S method in both simulations are slightly less. However, we like to note that both methods are very

efficient and the difference is almost negligible in the 2-d case. The main reason for this is that the advantages of the MDI algorithm are not fully utilized when calculating low-dimensional integrals. This is because that the STP-S method employs floating-point calculations in MATLAB, whereas MDI utilizes symbolic calculations. When the dimension is low and the number of integration points is small, the STP-S method will yield faster calculations of those function values. However, as the dimension and the number of integration points increase, evaluating function values at so many integration points becomes very expensive and the advantages of the MDI algorithm become more apparent. See **Test 2** and **Test 3** below for detailed comparisons.

Test 2. Let $\Omega = [0,2]^3$ and we consider the following 3-d integrands:

$$(23)\quad g(x) = \exp\big(5x_1^2 + 5x_2^2 + 5x_3^2\big), \qquad \widehat{g}(x) = \sin\big(2\pi + 10x_1^2 + 5x_2^2 + 20x_3^2\big).$$

We compute integrals of these two functions over Ω by using the STP-S and MDI methods. Again, let h denote the grid size, $N = \frac{2}{h} + 1$, $r = 2$ and $m = 1$.

Tables 3 and 4 display the computational results (errors and CPU times), we observe that when the number of integration points is small (i.e., the grid size h is relatively large), the STP-S method requires less CPU times in both simulations. However, when the number of integration points increases, the advantage shifts to the MDI method and becomes significant when the number of integration points become large. This is because, by the computational complexity analysis to be given in the next section, the CPU time required by the MDI method grows in (d^3N^2) order while that of the STP-S method increases in exponential order $O(N^d)$. As a result, it is expected that when

Table 3: Relative errors and CPU times of STP-S and MDI simulations with $m = 1$ for approximating $I(g, \Omega)$

		STP-S		MDI	
Mesh size (h)	Total nodes	Relative error	CPU time(s)	Relative error	CPU time(s)
0.1	9261	1.8762×10^{-1}	0.0594529	1.8762×10^{-1}	0.1678806
0.05	68921	1.0222×10^{-2}	0.0830445	1.0222×10^{-2}	0.2330631
0.025	531441	7.0238×10^{-4}	0.1748109	7.0238×10^{-4}	0.4138331
0.0125	4173281	4.5031×10^{-5}	0.4316260	4.5031×10^{-5}	0.8359030
0.0100	8120601	1.8502×10^{-5}	0.7565165	1.8502×10^{-5}	1.0155151
0.00625	33076161	2.8328×10^{-6}	2.7365100	2.8328×10^{-6}	1.8655724
0.003125	263374721	2.6601×10^{-7}	56.872493	2.6601×10^{-7}	7.6742217

Table 4: Relative errors and CPU times of STP-S and MDI simulations with $m = 1$ for approximating $I(\widehat{g}, \Omega)$

		STP-S		MDI	
Mesh size (h)	Total nodes	Relative error	CPU time(s)	Relative error	CPU time(s)
0.1	9261	2.5789×10^{-1}	0.042875	2.5789×10^{-1}	0.173581
0.05	68921	3.0493×10^{-1}	0.0729843	3.0493×10^{-1}	0.247261
0.025	531441	1.2800×10^{-2}	0.1942965	1.2800×10^{-2}	0.4377125
0.0125	4173281	4.9563×10^{-4}	0.9138938	4.9563×10^{-4}	0.8930069
0.0100	8120601	1.9345×10^{-4}	1.7594922	1.9345×10^{-4}	1.1012937
0.00625	33076161	2.8065×10^{-5}	7.0588477	2.8065×10^{-5}	2.3235149
0.003125	263374721	1.7128×10^{-6}	56.322503	1.7128×10^{-6}	9.7523139

$d = 3$ and N is large, the advantage of the MDI method over any standard tensor product method becomes significant, and it will be even more pronouncing when both d and N become large as shown by the tests to be given in the next subsection.

3.2. High-dimensional tests

Since the MDI method is designed to computing high dimensional integration, it is important to test its performance and power for $d \gg 1$. In addition, we provide a performance comparison (in terms of the CPU time) of the MDI with standard tensor product methods as well as with the classical Monte Carlo (MC) method on computing high-dimensional integration.

The next test compares the performance of the MDI and STP-S methods on computing a well-known integral in dimensions $2 \le d \le 11$, respectively.

Test 3. Let $\Omega = [0,1]^d$ for $2 \le d \le 11$ and consider the following Gaussian integrand:

$$g(x) = \frac{1}{\sqrt{2\pi}} \exp\Big(-\frac{1}{2}|x|^2\Big), \tag{24}$$

where $|x|$ stands for the Euclidean norm of the vector $x \in \mathbb{R}^d$.

Once again, we approximate the integral $I(g, \Omega)$ by the MDI and STP-S (i.e., $r = 2$) methods as done in **Tests 1–2**. We also set $m = 1$ in the MDI method and perform the simulations with two grid sizes $h = 0.1, 0.05$ (or $N = 11, 21$) respectively.

Table 5 presents the relative errors and CPU times of both MDI and STP-S methods using the grid size $h = 0.1$ (or $N = 11$) and Table 6 gives the

Table 5: Relative errors and CPU times of STP-S and MDI simulations with $m = 1$ for approximating $I(g, \Omega)$ when $N = 11$

	STP-S		MDI	
Dimension (d)	Relative error	CPU time(s)	Relative error	CPU time(s)
2	1.5809×10^{-6}	0.0015542	1.5809×10^{-6}	0.0853444
4	3.1618×10^{-6}	0.0091310	3.1618×10^{-6}	0.1348654
6	4.7427×10^{-6}	0.4403814	4.7427×10^{-6}	0.5389767
8	6.3237×10^{-6}	56.1856842	6.3237×10^{-6}	1.4880431
10	7.9046×10^{-6}	7341.3815698	7.9046×10^{-6}	3.7304532
11	8.6951×10^{-6}	80322.5805531	8.6951×10^{-6}	5.2628807

Table 6: Relative errors and CPU times of STP-S and MDI simulations with $m = 1$ for approximating $I(g, \Omega)$ when $N = 21$

	STP-S		MDI	
Dimension (d)	Relative error	CPU time(s)	Relative error	CPU time(s)
2	9.8542×10^{-8}	0.0068182	9.8542×10^{-8}	0.1122873
4	1.9708×10^{-7}	0.0715035	1.9708×10^{-7}	0.6049681
6	2.9564×10^{-7}	20.7002115	2.9564×10^{-7}	6.4742624
8	3.9149×10^{-7}	9622.1118103	3.9149×10^{-7}	19.8850829
9	4.4344×10^{-7}	215136.0654597	4.4344×10^{-7}	28.6823906
10	4.9271×10^{-7}	failed	4.9271×10^{-7}	38.9745044

corresponding results for $h = 0.05$ (or $N = 21$). We observe from both tables that the errors of both methods are the same as they should be (because they compute the same multi-summation in each simulation), but their CPU times are significantly different. The STP-S method is more efficient when the dimension $d \leq 6$ when $N = 11$ and $d \leq 5$ when $N = 21$, but the MDI method excels when the dimension $d > 6$ and the winning margin becomes significant as d and N increase (also see Figure 1). For example, when $d = 11$ and $N = 11$, the CPU time required by the STP-S method is about 80323 seconds, which is about 22 hours, but the CPU time required by the MDI method is only less than 6 seconds! In addition, when $d = 10$ and $N = 21$, the STP-S method fails to compute the integral (because the computational cost is too large for the computer to handle), but it only takes the MDI method about 39 seconds to finish the computation! The reason for such a dramatic CPU time saving is, by the computational complexity analysis to be given in the next section, that the CPU time required by the MDI method grows in (d^3N^2) order while that of the STP-S method increases in exponential order $O(N^d)$.

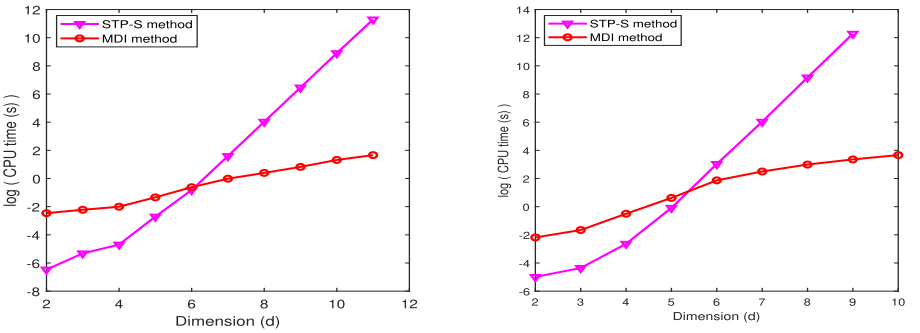

Figure 1: CPU time comparison of STP-S and MDI simulations: $N = 10$ (left), $N = 20$ (right).

Arguably, the classical Monte Carlo (MC) method (or its variant) is the method of choice for computing high dimensional integration. However, due to its low order of convergence and intrinsic need for using large amount of samples, it is only capable of simulating low and medium dimensional integration in practice due to the large number of function evaluations at randomly sampled integration points, which also grows quickly as the dimension d increases (due to the rapid growth of the variance). In the next test, we compare the performance of the MDI (with parameters $r = 2$, $N = 11$, $m = 1$) and the classical MC method.

Test 4. Let $\Omega = [0,1]^d$ and choose the following integrands:

$$g(x) = \prod_{i=0}^{d} \frac{1}{0.9^2 + (x_i - 0.6)^2}, \qquad \widehat{g}(x) = \frac{1}{\sqrt{2\pi}} \exp\Big(-\frac{1}{2}|x|^2\Big). \tag{25}$$

First, we use the relative error as the metric to guage the performance, namely, we use sufficient number of the random sampling points for the MC method so it produces a comparable relative error to that of the MDI method. The computational results for approximating $I(g,\Omega)$ and $I(\widehat{g},\Omega)$ are presented respectively in Tables 7 and 8 below.

From Tables 7 and 8, we clearly see that the CPU times of the MDI and MC methods are significantly different, the discrepancy becomes so dramatic when $d \geq 40$ because the MDI method only takes a few seconds/minutes to finish the computation of approximating $I(g,\Omega)$ and $I(\widehat{g},\Omega)$, while the MC method fails to produce a result on the computer! This is because, in order for the MC method to obtain an approximate value with the relative error

Table 7: CPU times of the MDI and MC simulations with comparable relative errors for approximating $I(g, \Omega)$

	MC		MDI	
Dimension (d)	Relative error	CPU time(s)	Relative error	CPU time(s)
5	2.6251×10^{-5}	9.2872030	2.6251×10^{-5}	0.1755604
10	5.2504×10^{-5}	17.9837795	5.2504×10^{-5}	0.2510754
20	1.0501×10^{-4}	66.6797401	1.0501×10^{-4}	0.6229516
30	1.5752×10^{-4}	4302.1801130	1.5752×10^{-4}	0.8786786
35	1.8377×10^{-4}	11055.6351555	1.8377×10^{-4}	1.0470143
40	2.1003×10^{-4}	failed	2.1003×10^{-4}	1.2647430
80	4.2011×10^{-4}		4.2011×10^{-4}	2.8572520
100	5.2516×10^{-4}		5.2516×10^{-4}	3.2840268

Table 8: CPU times of the MDI and MC simulations with comparable relative errors for approximating $I(\widehat{g}, \Omega)$

	MC		MDI	
Dimension (d)	Relative error	CPU time(s)	Relative error	CPU time(s)
5	3.9523×10^{-6}	33.3246187	3.9523×10^{-6}	0.2620139
10	7.9046×10^{-6}	346.0491007	7.9046×10^{-6}	3.7304532
20	1.5809×10^{-5}	1210.092329	1.5809×10^{-5}	35.9158247
30	2.3714×10^{-5}	3416.6898735	2.3714×10^{-5}	108.7032555
35	2.7666×10^{-5}	12664.1000000	2.7666×10^{-5}	154.1259392
40	3.1618×10^{-5}	failed	3.1618×10^{-5}	240.1254540
80	6.3238×10^{-5}		6.3238×10^{-5}	1678.4531292
100	7.9049×10^{-5}		7.9049×10^{-5}	3207.3510623

of order 10^{-5}, it requires about 10^{10} randomly sampled integration points at which function values must be computed independently, a task that is too big to be handled by Matlab on the computer. We note that g is an oscillatory function and $\widehat{g}$ is an exponentially growth function, both functions are tough for the MC method to handle, because a very large number of sampling points must be used to resolve those functions with a reasonable resolution and the function values must be computed independently at those points in the MC method. On the other hand, although the MDI method must use a comparable large amount (if not larger) of integration points (because the underlying tensor product method does), due to its efficient way of computing those function evaluations in cluster and iteratively along each coordinate direction, the MDI can handle the computation of the multi-summation in the blink of an eye.

Table 9: CPU times of the MDI and MC simulations using the same number of integration points for approximating $I(g, \Omega)$

		MC		MDI	
Dimension (d)	Total nodes	Relative error	CPU time(s)	Relative error	CPU time(s)
5	11^5	5.8241×10^{-4}	0.3056289	2.6251×10^{-5}	0.1755604
11	11^{11}	4.3309×10^{-7}	9827.3992235	5.7754×10^{-5}	0.2815804
15	11^{15}	failed		7.8757×10^{-5}	0.4811588
30	11^{30}			1.5752×10^{-4}	0.8786786
40	11^{40}			2.1003×10^{-4}	1.2647430
50	11^{50}			2.6254×10^{-4}	1.6818385

Next, we compute the same test problems as above but use a different metric to gauge the performance of both methods. We now specify the number of integration points, instead of the relative errors, then let both methods compute their respective multi-summations using the same number of points (so the same number of function evaluations are required by both methods to approximate the integrals). Tables 9 and 10 present the simulation results. Although the numbers are slightly different but the message is the same, that is, the MC method fails to produce a result when the dimension $d \geq 15$ while the MDI can finish the simulation in the blink of an eye for the integrand g and in a few minutes for the integrand $\widehat{g}$ even when $d = 50$.

Table 10: CPU times of the MDI and MC simulations using the same number of integration points for approximating $I(\widehat{g}, \Omega)$

		MC		MDI	
Dimension (d)	Total nodes	Relative error	CPU time(s)	Relative error	CPU time(s)
5	11^5	1.0664×10^{-3}	0.0080803	3.9523×10^{-6}	0.2620139
11	11^{11}	1.1947×10^{-6}	10195.085484	8.6951×10^{-6}	6.9215447
15	11^{15}	failed		1.1856×10^{-5}	15.316679
30	11^{30}			2.3714×10^{-5}	108.703255
40	11^{40}			3.1618×10^{-5}	240.125454
50	11^{50}			3.9523×10^{-5}	413.607179

One natural question is how high the dimension d which the MDI can handle. First, we note that the answer is machine-dependent as expected. Second, we perform the next test to seek an answer to this question using the computer at our disposal as described at the beginning of this section.

Test 5. Let $\Omega = [0,1]^d$ and consider the following integrands:

$$\begin{aligned} &\widetilde{g}(x) = \exp\Big(\prod_{i=1}^{d} x_i\Big), \qquad g(x) = \exp\Big(\sum_{i=1}^{d} (-1)^{i+1} x_i\Big), \\ &\widehat{g}(x) = \prod_{i=0}^{d} \frac{1}{0.9^2 + (x_i - 0.6)^2}. \end{aligned} \tag{26}$$

We first approximate $I(\widetilde{g}, \Omega)$ using the MDI algorithm with parameters $r = 3$ (three-point Gaussian rule), $m = 1$, $N = 3$ and an increasing sequence of d. The computed results are presented in Table 11. We note that the composite three-point Gaussian rule is very accurate for evaluating this integral.

We then approximate $I(g, \Omega)$ and $I(\widehat{g}, \Omega)$ using the MDI algorithm with parameters $r = 2$ (composite Simpson's rule), $m = 1$, $N = 7$ and an increasing sequence of d. The computed results are presented in Tables 12 and 13. The simulation is stopped at $d = 1000$ because it is already in the very high dimension regime and $N = 7$ is chosen to minimize the computation and because it is sufficient to produce reasonable relative errors. This test demonstrates the promise and capability of the MDI method for efficiently computing high dimensional integrals.

Table 11: Computed results for approximating $I(\widetilde{g}, \Omega)$ by the MDI algorithm

Dimension (d)	Total nodes	Approximation	Relative error	CPU time(s)
10	3^{10}	1.000985193399077	1.4085×10^{-13}	0.133632
20	3^{20}	1.000000953817867	9.5919×10^{-16}	0.3871058
30	3^{30}	1.000000000931325	1.4387×10^{-16}	1.6115572
40	3^{40}	1.000000000000909	8.8817×10^{-16}	5.2981164
50	3^{50}	1.000000000000001	2.3979×10^{-17}	16.5415132
60	3^{60}	1.000000000000000	2.8775×10^{-17}	32.007989
70	3^{70}	1.000000000000000	3.3571×10^{-17}	59.8037414
80	3^{80}	1.000000000000000	3.8367×10^{-17}	107.1823183
90	3^{90}	1.000000000000000	4.3163×10^{-17}	170.3656174
100	3^{100}	1.000000000000000	4.2734×10^{-17}	249.1619032

Table 12: Computed results for approximating $I(g,\Omega)$ by the MDI algorithm

Dimension (d)	Total nodes	Relative error	CPU time(s)
10	7^{10}	4.2726×10^{-5}	0.2459398
100	7^{100}	4.2734×10^{-4}	74.6973942
200	7^{200}	8.5487×10^{-4}	503.3034372
300	7^{300}	1.4386×10^{-4}	1560.0488058
400	7^{400}	1.7097×10^{-3}	3546.4398972
500	7^{500}	2.1371×10^{-3}	6772.0225935
600	7^{600}	2.8772×10^{-3}	11954.1886240
700	7^{700}	3.3566×10^{-3}	19355.4847153
800	7^{800}	3.5194×10^{-3}	28273.5752793
900	7^{900}	3.8467×10^{-3}	42427.2391457
1000	7^{1000}	4.2742×10^{-3}	62445.0882189

Table 13: Computed results for approximating $I(\widehat{g},\Omega)$ by the MDI algorithm

Dimension (d)	Total nodes	Relative error	CPU time(s)
10	7^{10}	4.0743×10^{-4}	0.2059168
100	7^{100}	4.0818×10^{-3}	2.0993900
200	7^{200}	8.1803×10^{-3}	4.5213100
300	7^{300}	1.2295×10^{-2}	7.2644682
400	7^{400}	1.6427×10^{-2}	10.1062101
500	7^{500}	2.0576×10^{-2}	13.5705851
600	7^{600}	2.4742×10^{-2}	17.8284828
700	7^{700}	2.8925×10^{-2}	21.6876065
800	7^{800}	3.3125×10^{-2}	25.9204244
900	7^{900}	3.7342×10^{-2}	31.3307727
1000	7^{1000}	4.1576×10^{-2}	35.7704489

4. Influence of parameters

Besides the dimension d, there are three other input parameters in the MDI algorithm, they are r, m and N. The parameter r indicates the choice of 1-d base numerical quadrature rule. As mentioned earlier, here we only consider four such choices, hence, r takes integer values $\{1,2,3,4\}$ and they represent respectively the (composite) trapezoidal rule, Simpson's rule, two-point Gaussian rule, and midpoint rule. Their efficiency will be tested in this section.

Recall that m represents the step length in the multi-dimension iteration, namely, it indicates how many dimensions to reduce at each iteration. Practically, $1 \le m \le 3$, hence, it takes integer values $\{1, 2, 3\}$. The performance of each of these choices will be compared in this section. It should be noted that after $\ell := [\frac{d}{m}]$ iterations, the residual dimension satisfies $d - \ell m \le m$. Then in case $m = 2$ or 3, one has two options to choose to complete the algorithm. On one hand, one just continues the dimension reduction by calling 3d-MDI or 2d-MDI as explained in the definition of **Algorithm 3**. On the other hand, it is also possible to compute the remaining 2- or 3-d integration directly using the underlying 2- or 3-d tensor product formula without further dimension reduction. The effect of these two choices will be tested in this section.

It is clear that the larger N, the more expensive the computation. The dependence of the efficiency of the MDI algorithm on the parameter N will also be tested.

4.1. Influence of parameter r

We first examine the effect of the choices $r = 1, 2, 3, 4$ in the MDI algorithm. They will be done on the same grid (i.e., N fixed) and with the same step length $m = 1$.

Test 6. Let $\Omega = [0, 1]^d$ and the integrand g be given by (24).

Table 14 presents the simulation results of **Test 6**. We note that since the composite two-point Gaussian rule ($r = 3$) is too expensive to compute this integral when the dimension is larger than 10, so it is not included in this test. It shows that Simpson's and trapezoidal rules have the same efficiency, but Simpson's rule has much better accuracy. The midpoint and trapezoidal rules have the same accuracy, but the midpoint rule is three times more efficient than the trapezoidal rule in this test.

Test 7. Let $\Omega = [0, 1]^d$ and choose the integrand g as

$$g(x) = \exp\Big(\sum_{i=1}^{d}(-1)^{i+1}x_i\Big), \qquad \widehat{g}(x) = \prod_{i=0}^{d}\frac{1}{0.9^2 + (x_i - 0.6)^2}. \tag{27}$$

Table 15 presents the simulation results of **Test 7** for approximating integral $I(g, \Omega)$. Again, choosing different types of the 1-d base quadrature rule has a significant impact on the accuracy and efficiency of the MDI algorithm. In terms of accuracy, the trapezoidal ($r = 1$) and midpoint ($r = 4$)

Table 14: Efficiency comparison of the MDI algorithm with $m = 1$ and $r = 1, 2, 4$

Parameter (r)	Dimension (d)	Points (N)	Relative error	CPU time(s)
$r = 1$	10	11	5.8935×10^{-3}	3.8294275
	30	11	1.7576×10^{-2}	97.3001349
	50	11	2.9122×10^{-2}	419.1940599
	70	11	4.0532×10^{-2}	1056.501204
	90	11	5.1808×10^{-2}	2199.34273
	100	11	5.7396×10^{-2}	3255.997642
$r = 2$	10	11	7.9046×10^{-6}	3.7304532
	30	11	2.3714×10^{-5}	108.703255
	50	11	3.9523×10^{-5}	413.607179
	70	11	5.5333×10^{-5}	1147.446169
	90	11	7.1144×10^{-5}	2388.382073
	100	11	7.9049×10^{-5}	3207.351062
$r = 4$	10	10	2.9593×10^{-3}	1.6020058
	30	10	8.9042×10^{-3}	50.2901972
	50	10	1.4884×10^{-2}	186.0984739
	70	10	2.0899×10^{-2}	436.3740376
	90	10	2.6951×10^{-2}	855.0901709
	100	10	2.9990×10^{-2}	1062.3387568

rules are comparable, but the midpoint rule is more efficient (in terms of the CPU time) as the dimension d increases. Similarly, Simpson's ($r = 2$) and two-point Gaussian ($r = 3$) rules are comparable in accuracy, but Simpson's rule is significantly more efficient even the Gaussian rule uses fewer integration points. Moreover, Simpson's rule is much more accurate than the trapezoidal and midpoint rules with comparable efficiency because all three quadrature rules use the same number of integration points. The comparison shows that Simpson's rule is a clear winner among these four rules when they are used as the building block in the MDI algorithm for high dimension integration. We note that the reason that the two-point Gaussian rule requires a lot more CPU time is because it is costly to generate the Gauss points on fly and to do their function evaluations.

Table 16 shows the simulation results of **Test 7** for approximating integral $I(\widehat{g}, \Omega)$. Due to the nicer behavior of the integrand $\widehat{g}$, the MDI algorithm is very fast with all four base quadrature rules for computing this integral although the same observations as above can be made. Once again, the Simpson's rule excels.

Table 15: Efficiency comparison of the MDI algorithm with $m = 1$ and $r = 1, 2, 3, 4$

Parameter (r)	Dimension (d)	Points (N)	Relative error	CPU time(s)
$r = 1$	10	11	8.3632×10^{-3}	0.3793491
	30	11	2.5300×10^{-2}	3.9242551
	50	11	4.2521×10^{-2}	20.7387841
	70	11	6.0032×10^{-2}	76.7165061
	90	11	7.7837×10^{-2}	170.2491139
	100	11	8.6851×10^{-2}	234.4891902
$r = 2$	10	11	5.5489×10^{-6}	0.3435032
	30	11	1.6646×10^{-5}	4.0590394
	50	11	2.7745×10^{-5}	20.6479181
	70	11	3.8843×10^{-5}	69.2068795
	90	11	4.9941×10^{-5}	162.1716159
	100	11	5.5491×10^{-5}	209.2587748
$r = 3$	10	6	2.8477×10^{-5}	0.8652789
	30	6	8.5428×10^{-5}	219.3882758
	50	6	1.4237×10^{-4}	5281.6063020
	60	6	2.8775×10^{-5}	16366.6127593
	70	6		failed
$r = 4$	10	10	4.1576×10^{-3}	0.3168370
	30	10	1.2421×10^{-2}	2.3652108
	50	10	2.0616×10^{-2}	13.5012784
	70	10	2.8743×10^{-2}	45.0645030
	90	10	3.6802×10^{-2}	112.2156751
	100	10	4.0807×10^{-2}	153.9030288

4.2. Influence of parameter m

Recall that m stands for the step length in the MDI algorithm, it represents how many dimensions are reduced at each iteration. The intuition is the more reduction the better. However, that is not true because at each iteration, many m-dimensional tensor product sums must be evaluated. Hence, practically we have $1 \leq m \leq 3$. The next test presents a performance comparison of the MDI algorithm using $m = 1, 2, 3$.

Test 8. Let $\Omega = [0, 1]^d$, g and $\widehat{g}$ be the same as in (25).

We compute these integrals using the MDI algorithm with $r = 2$ (composite Simpson's rule) and $N = 11$. Tables 17 and 18 present respectively the computed results for these two integrals. We observe that the MDI algorithm with different parameters m has the same accuracy which is expected.

Table 16: Efficiency comparison of the MDI algorithm with $m = 1$ and $r = 1, 2, 3, 4$

Parameter (r)	Dimension (d)	Points (N)	Relative error	CPU time(s)
$r = 1$	10	11	1.2809×10^{-2}	0.2997928
	30	11	3.7939×10^{-2}	0.8831823
	50	11	6.2429×10^{-2}	1.5778940
	70	11	8.6296×10^{-2}	2.2930658
	90	11	1.0955×10^{-1}	3.1440537
	100	11	1.2096×10^{-1}	3.5166433
$r = 2$	10	11	5.2504×10^{-5}	0.2510754
	30	11	1.5752×10^{-4}	0.8786786
	50	11	2.6254×10^{-4}	1.6818385
	70	11	3.6758×10^{-4}	2.2283636
	90	11	4.7263×10^{-4}	3.1751084
	100	11	5.2516×10^{-4}	3.2840268
$r = 3$	10	10	3.5014×10^{-5}	0.3039294
	30	10	1.0503×10^{-4}	1.3949243
	50	10	1.7505×10^{-4}	4.2661412
	70	10	2.4507×10^{-4}	9.8203358
	90	10	3.1508×10^{-4}	19.3491396
	100	10	3.5008×10^{-4}	25.3835608
$r = 4$	10	10	6.4658×10^{-3}	0.2454473
	30	10	1.9523×10^{-2}	0.7190225
	50	10	3.2750×10^{-2}	1.3526353
	70	10	4.6148×10^{-2}	1.8832556
	90	10	5.9720×10^{-2}	2.4435807
	100	10	6.6572×10^{-2}	2.8005838

Table 17: Efficiency comparison of the MDI algorithm with $r = 2$ and $m = 1, 2, 3$

		$m = 1$	$m = 2$	$m = 3$
Dimension (d)	Relative error	CPU time(s)	CPU time(s)	CPU time(s)
10	5.2504×10^{-5}	0.2510754	1.7061606	10.8277897
30	1.5752×10^{-4}	0.8786786	5.6306318	49.1959065
50	2.6254×10^{-4}	1.6818385	10.1213779	79.0436736
70	3.6758×10^{-4}	2.2283636	14.4966060	127.4604677
90	4.7263×10^{-4}	3.1751084	19.3119927	171.5185236
100	5.2516×10^{-4}	3.2840268	21.8707585	196.2232037

Table 18: Efficiency comparison of the MDI algorithm with $r = 2$ and $m = 1, 2, 3$

Dimension (d)	Relative error	$m = 1$ CPU time(s)	$m = 2$ CPU time(s)	$m = 3$ CPU time(s)
10	7.9046×10^{-6}	3.7304532	17.4647858	31.7445041
30	2.3714×10^{-5}	108.703255	548.4820562	3762.781656
50	3.9523×10^{-5}	413.607179	2208.4417154	15367.76577
70	5.5333×10^{-5}	1147.446169	4332.8074455	35433.0000001
90	7.1144×10^{-5}	2388.382073	12397.0363557	75097.6376617
100	7.9049×10^{-5}	3207.351062	16428.9713811	102930.139707

However, the choice of m do affect the efficiency of the algorithm. It shows that the algorithm is most efficient when $m = 1$. The explanation for this observation is that when using larger m, the number of nested loops increases despite the number of iterations decreases. When $m = 1$, there is only one loop per iteration, so the MDI algorithm becomes faster.

4.3. Influence of the parameter N

In this section, we test the influence of the number of integration points N in each coordinate direction on the MDI algorithm. We set $m = 1$ and $r = 2$ (Simpson) in the test.

Test 9. Let $\Omega = [0, 1]^d$ and choose the following integrands:

$$g(x) = \exp\Big(\sum_{i=1}^{d}(-1)^{i+1}x_i\Big), \qquad \widehat{g}(x) = \cos\Big(2\pi + 2\sum_{i=1}^{d} x_i\Big),$$

$$\widetilde{g}(x) = \prod_{i=0}^{d} \frac{1}{0.9^2 + (x_i - 0.6)^2}.$$

Tables 19, 20 and 21 present respectively the computed results of **Test 9** with $d = 5, 10$ and $N = 11, 21, 41, 81, 161, 321$. It should be noted that the quality of the approximation also depends on the behavior of the integrand. For very oscillatory and fast growth functions, more integration points must be used to achieve good accuracy. In the next section, we shall examine using the regression technique the relationship between the CPU time and the parameter N and the dimension d.

Table 19: Performance comparison of the MDI algorithm with $d = 5, 10$ and $N = 11, 21, 41, 81, 161, 321$ for approximating $I(g, \Omega)$

		$d = 5$		$d = 10$	
Mesh size (h)	Points N	Relative error	CPU time(s)	Relative error	CPU time(s)
0.1	11	2.7744×10^{-6}	0.1756857	5.5489×10^{-6}	0.4397261
0.05	21	1.7355×10^{-7}	0.3406698	3.4711×10^{-7}	1.0226371
0.025	41	1.0849×10^{-8}	0.7929361	2.1699×10^{-8}	4.0287093
0.0125	81	6.7815×10^{-10}	2.1783682	1.3563×10^{-9}	31.1292788
0.00625	161	4.2384×10^{-11}	31.7971237	8.4768×10^{-11}	141.9877281
0.003125	321	2.6479×10^{-12}	136.4977085	5.2993×10^{-12}	550.772326

Table 20: Performance comparison of the MDI algorithm with $d = 5, 10$ and $N = 11, 21, 41, 81, 161, 321$ for approximating $I(\widehat{g}, \Omega)$

		$d = 5$		$d = 10$	
Mesh size (h)	Points N	Relative error	CPU time(s)	Relative error	CPU time(s)
0.1	11	4.4657×10^{-5}	0.1832662	8.9317×10^{-5}	0.4046452
0.05	21	2.7810×10^{-6}	0.3478922	5.5621×10^{-6}	1.0245456
0.025	41	1.7366×10^{-7}	0.7666344	3.4732×10^{-7}	4.5341739
0.0125	81	1.0851×10^{-8}	2.8629733	2.1703×10^{-8}	35.0469255
0.00625	161	6.7818×10^{-10}	26.3679753	1.3563×10^{-9}	150.875945
0.003125	321	4.2383×10^{-11}	146.441442	8.4768×10^{-11}	568.693914

Table 21: Performance comparison of the MDI algorithm with $d = 5, 10$ and $N = 11, 21, 41, 81, 161, 321$ for approximating $I(\widetilde{g}, \Omega)$

		$d = 5$		$d = 10$	
Mesh size (h)	Points N	Relative error	CPU time(s)	Relative error	CPU time(s)
0.1	11	2.6251×10^{-5}	0.1669653	5.2504×10^{-5}	0.273292
0.05	21	1.6339×10^{-6}	0.2529685	3.2679×10^{-6}	0.4857094
0.025	41	1.0200×10^{-7}	0.4683767	2.0401×10^{-7}	0.8593056
0.0125	81	6.3736×10^{-9}	0.8346902	1.2747×10^{-8}	1.7529975
0.00625	161	3.9832×10^{-10}	1.6505776	7.9664×10^{-10}	3.6539963
0.003125	321	2.4894×10^{-11}	3.6465664	4.9788×10^{-11}	9.5368530

5. Computational complexity

5.1. The relationship between the CPU time and N

In this subsection, we examine the relationship between the CPU time and parameter N using the regression technique based on the test data.

Figures 2 and 3 show the CPU time as a function of N obtained by the least square method and the fitting functions are given in Table 22. All the results indicate that the CPU time grows at most quadratically in N.

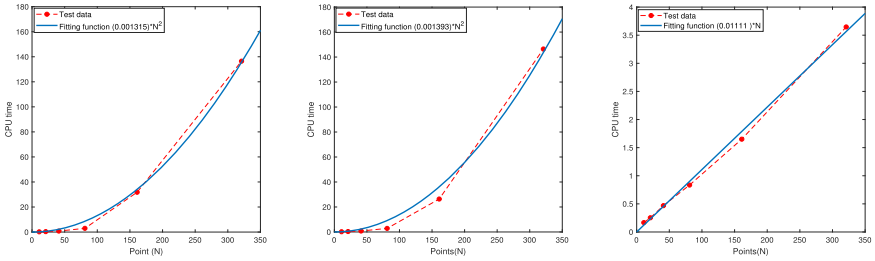

Figure 2: The relationship between the CPU time and parameter N when $d = 5$: $I(g, \Omega)$ (left), $I(\widehat{g}, \Omega)$ (middle), $I(\widetilde{g}, \Omega)$ (right).

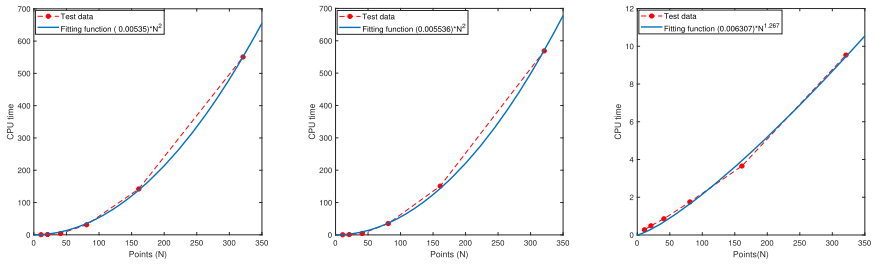

Figure 3: The relationship between the CPU time and parameter N when $d = 10$: $I(g, \Omega)$ (left), $I(\widehat{g}, \Omega)$ (middle), $I(\widetilde{g}, \Omega)$ (right).

Table 22: The relationship between the CPU time and parameter N

Integrand	r	m	d	Fitting function	R-square
$g(x)$	2	1	5	$h_1(N) = (0.001315) * N^2$	0.9973
$\widehat{g}(x)$	2	1	5	$h_2(N) = (0.001393) * N^2$	0.9914
$\widetilde{g}(x)$	2	1	5	$h_3(N) = (0.01111) * N$	0.9963
$g(x)$	2	1	10	$h_4(N) = (0.00535) * N^2$	0.9998
$\widehat{g}(x)$	2	1	10	$h_5(N) = (0.005536) * N^2$	0.9997
$\widetilde{g}(x)$	2	1	10	$h_6(N) = (0.006307) * N^{1.267}$	0.9971

5.2. The relationship between the CPU time and the dimension d

Recall that the computational complexity of tensor product methods is of the exponential order $O(N^d)$. The numerical tests presented above overwhelmingly and consistently show that the MDI algorithm has hidden capability to overcome the curse of dimensionality faced by tensor product methods. The goal of the next test is to find out the computational complexity (in terms of CPU time as a function of d) using the least square method based on numerical test data.

Test 10. Let $\Omega = [0,1]^d$, we consider the following five integrands:

$$g_1(x) = \exp\Big(\sum_{i=1}^{d}(-1)^{i+1}x_i\Big), \qquad g_2(x) = \prod_{i=0}^{d}\frac{1}{0.9^2 + (x_i - 0.6)^2},$$

$$g_3(x) = \frac{1}{\sqrt{2\pi}}\exp\Big(-\frac{1}{2}|x|^2\Big), \qquad g_4(x) = \cos\Big(2\pi + 2\sum_{i=1}^{d}x_i\Big),$$

$$g_5(x) = \exp\Big(5\sum_{i=1}^{d}(-1)^{i+1}x_i^2\Big), \qquad g_6(x) = \exp\Big(\prod_{i=1}^{d}x_i\Big).$$

Figure 4 displays the CPU time as functions of d obtained by the least square method whose analytical expressions are given in Table 23. We note that the parameters of the MDI algorithm only affect the coefficients of the fitting functions, but not the order.

We quantitatively characterize the performance of the fitted curve by the R-square in Matlab, which is defined as $R\text{-square} = 1 - \frac{\sum_i^n (y_i - \widehat{y}_i)^2}{\sum_i^n (y_i - \overline{y})^2}$. Where y_i represents the test data, $\widehat{y}_i$ refers to the predicted value, and $\overline{y}$ indicates the mean value of y_i. Table 23 also shows that the R-square of all fitting functions is very close to 1, which indicates the fitting function is quite accurate. These results indicate that the CPU time grows at most cubically in d. Combining the results of **Test 9** in Section 4.3 we conclude that the CPU time required by the proposed MDI algorithm grows at most in the polynomial order $O(d^3N^2)$.

6. Conclusions

In this paper we introduced a fast MDI (multilevel dimension iteration) algorithm (or solver) for efficiently implementing tensor product methods for

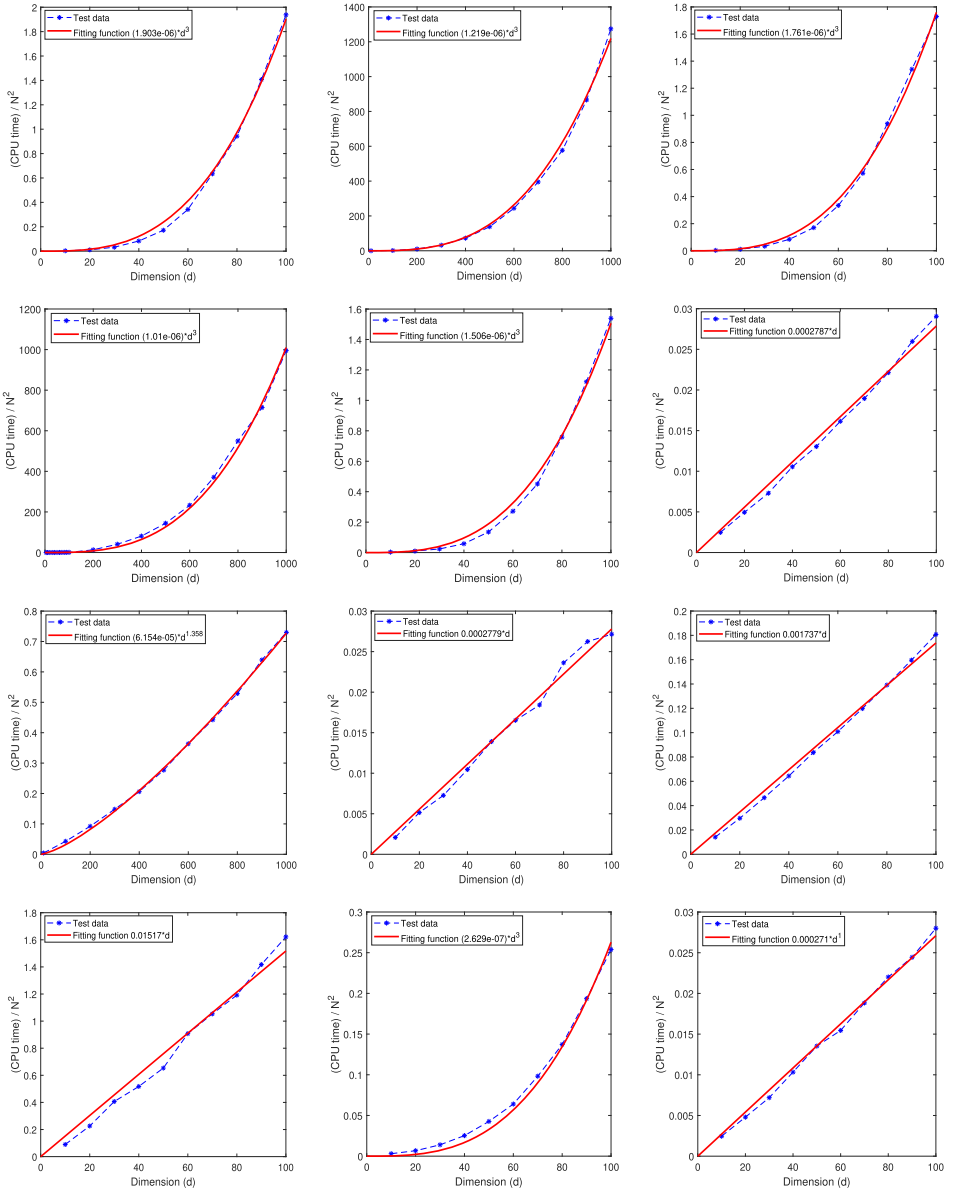

Figure 4: The relationship between the CPU time and dimension d.

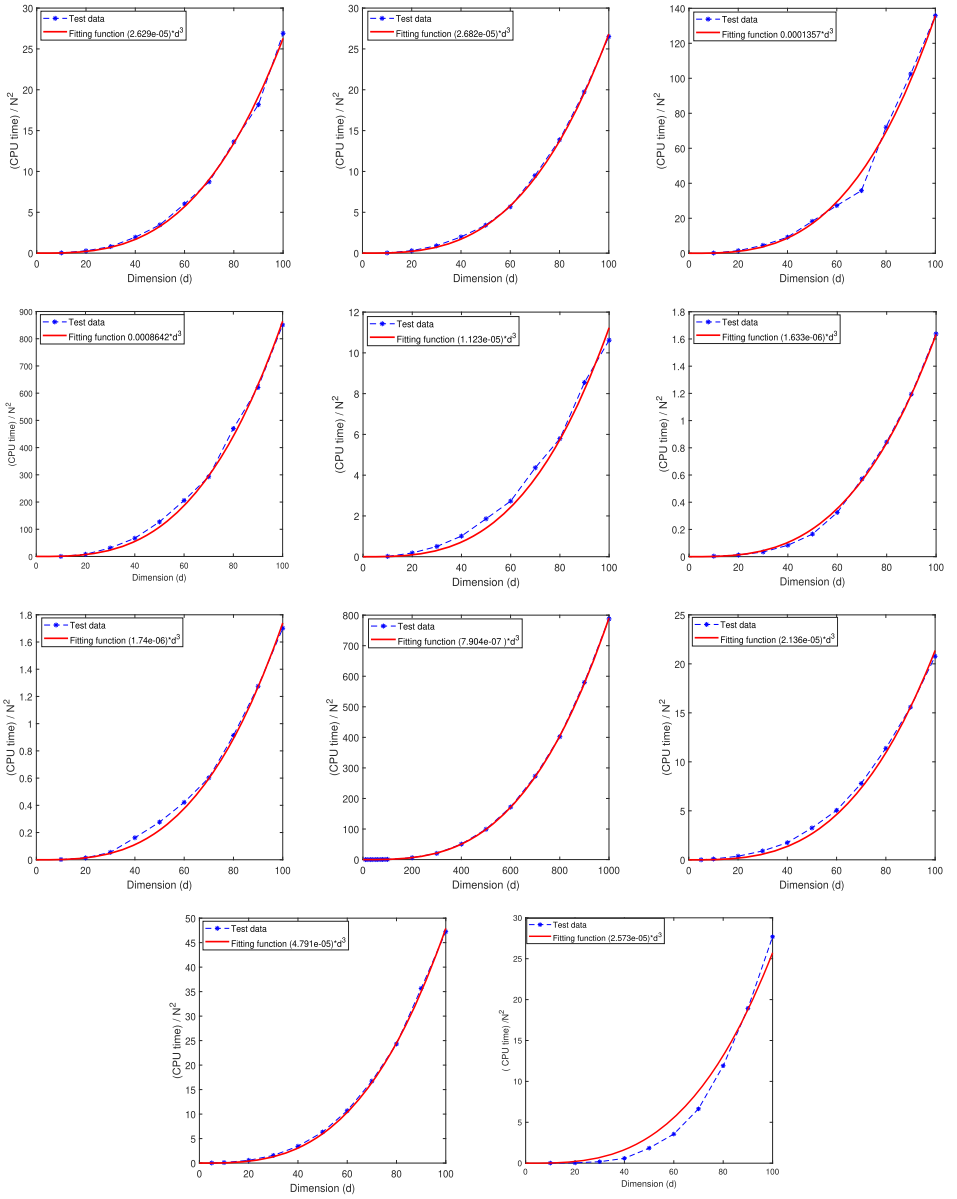

Figure 4: (*Continued.*)

Table 23: The relationship between CPU time and the integral dimension d

Integrand	r	m	N	Fitting function	R-square
g_1	1	1	11	$f_1 = (1.903e-06) * N^2 d^3$	0.9966
	2	1	7	$f_2 = (1.219e-06) * N^2 d^3$	0.9961
	2	1	11	$f_3 = (1.761e-06) * N^2 d^3$	0.9964
	3	1	3	$f_4 = (1.01e-06) * N^2 d^3$	0.9978
	4	1	10	$f_5 = (1.506e-06) * N^2 d^3$	0.9947
g_2	1	1	11	$f_6 = 0.0002787 * N^2 d^1$	0.9922
	2	1	7	$f_7 = (6.154e-05) * N^2 d^{1.358}$	0.9991
	2	1	11	$f_8 = 0.0002779 * N^2 d^1$	0.9898
	2	2	11	$f_9 = 0.001737 * N^2 d^1$	0.9937
	2	3	11	$f_{10} = 0.01517 * N^2 d^1$	0.9808
	3	1	10	$f_{11} = (2.629e-07) * N^2 d^3$	0.9932
	4	1	10	$f_{12} = 0.000271 * N^2 d^1$	0.9952
g_3	1	1	11	$f_{13} = (2.629e-05) * N^2 d^3$	0.9977
	2	1	11	$f_{14} = (2.682e-05) * N^2 d^3$	0.9995
	2	2	11	$f_{15} = 0.0001357 * N^2 d^3$	0.9929
	2	3	11	$f_{16} = 0.0008642 * N^2 d^3$	0.9974
	4	1	10	$f_{17} = (1.123e-05) * N^2 d^3$	0.9903
g_4	2	1	11	$f_{18} = (1.633e-06) * N^2 d^3$	0.9990
	2	1	21	$f_{19} = (1.74e-06) * N^2 d^3$	0.9966
	3	1	3	$f_{20} = (7.904e-07) * N^2 d^3$	1.0000
g_5	2	1	11	$f_{20} = (2.136e-05) * N^2 d^3$	0.9968
	2	1	21	$f_{22} = (4.791e-05) * N^2 d^3$	0.9994
g_6	3	1	3	$f_{23} = (2.573e-05) * N^2 d^3$	0.9786

high dimension numerical integration. It is based on the idea of computing the function evaluations at all integration points in cluster and iteratively along each coordinate direction, so many computations can be reused in each iteration. It was showed numerically based on the simulation data that the computational complexity (in terms of the CPU time) of the MDI algorithm grows at most cubically in the dimension d, and overall in the order $O(d^3N^2)$, which shows that the proposed MDI algorithm could effectively circumvent the curse of the dimensionality in high dimensional numerical integration, hence, makes tensor product methods not only become competitive but also can excel. Extensive numerical tests were provided to gauge the performance of the MDI algorithm and to do performance comparisons with the standard tensor product methods and especially with the Monte Carlo (MC) method. They demonstrated that the MDI algorithm (regardless the choice of the 1-d base quadrature rules) is faster than the MC method in low and medium dimensions (i.e., $d \approx 100$), much faster in very high dimensions (i.e., $d \approx 1000$), and succeeds even when the MC method fails. As the idea of the

MDI algorithm is applicable to any quadrature rule whose integration points have a lattice-like structure, this extension will be further investigated in the future. Another direction of continuing this research is to sharpen the dimension-iteration idea to develop even faster algorithms which can achieve the optimal computational complexity (in terms of the CPU time) of the order $O(Nd)$, we shall present those new results in a forthcoming work.

References

[1] H.-J. Bungartz and M. Griebel, *Sparse grids*, Acta Numer., 13:147–269, 2014. MR2249147

[2] R. L. Burden and J. D. Faires, *Numerical Analysis*, 10th edition, Cengage Learning, 2015.

[3] R. E. Caflisch, *Monte Carlo and quasi-Monte Carlo methods*, Acta Numer., 7:1–49, 1998. MR1689431

[4] J. Chen, R. Du, and K. Wu, *A comparison study of deep Galerkin method and deep Ritz method for elliptic problems with different boundary conditions*, Commun. Math. Res., 36:354–376, 2020. MR4199960

[5] J. Dick, F. Y. Kuo, and I. H. Sloan, *High-dimensional integration: the quasi-Monte Carlo way*, Acta Numer. 22:133–288, 2013. MR3038697

[6] J. Dos Santos Azevedo and S. Pomponet Oliveira, *A numerical comparison between quasi-Monte Carlo and sparse grid stochastic collocation methods*, Commun. Comput. Phys., 12:1051–1069, 2012. MR2913450

[7] W. E and B. Yu, *The deep Ritz method: A deep learning-based numerical algorithm for solving variational problems*, Commun. Math. Stat., 6:1–12, 2018. MR3767958

[8] M. Griebel and M. Holtz, *Dimension-wise integration of high-dimensional functions with applications to finance*, J. Complexity, 26:455–489, 2010. MR2719643

[9] J. Han, A. Jentzen, and W. E, *Solving high-dimensional partial differential equations using deep learning*, PNAS, 115:8505–8510, 2018. MR3847747

[10] F. J. Hickernell, T. Müller-Gronbach, B. Niu, and K. Ritter, *Multi-level Monte Carlo algorithms for infinite-dimensional integration on R^N*, J. Complexity 26:229–254, 2010. MR2657363

[11] F. Y. Kuo, C. Schwab, and I. H. Sloan, *Quasi-Monte Carlo methods for high-dimensional integration: the standard (weighted Hilbert space) setting and beyond*, ANZIAM J., 53:1–37, 2011. MR2928989

[12] J. Lu and L. Darmofal, *Higher-dimensional integration with Gaussian weight for applications in probabilistic design*, SIAM J. Sci. Comput., 26:613–624, 2004. MR2116364

[13] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, *DeepXDE: A deep learning library for solving differential equations*, SIAM Rev., 63:208–228, 2021. MR4209661

[14] Y. Ogata, *A Monte Carlo method for high dimensional integration*, Numer. Math. 55:137–157, 1989. MR0987382

[15] J. Sirignano and K. Spiliopoulos, *DGM: A deep learning algorithm for solving partial differential equations*, J. Comput. Phys. 375:1339–1364, 2018. MR3874585

[16] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Springer, New York, 1980. MR0557543

[17] A. Wipf, *High-Dimensional Integrals*, in *Statistical Approach to Quantum Field Theory*, Lecture Notes in Physics, 100:25–46, Springer, 2013. MR2985153

[18] J. Xu, *Finite neuron method and convergence analysis*, Commun. Comput. Phys, 28:1707–1745, 2020. MR4188519

Xiaobing Feng
Department of Mathematics
The University of Tennessee
Knoxville, TN 37996
U.S.A.
E-mail address: xfeng@utk.edu

Huicong Zhong
School of Mathematics and Statistics
Northwestern Polytechnical University
Xi'an, Shaanxi 710129
China
E-mail address: huicongzhong@mail.nwpu.edu.cn

Received May 22, 2023