Novel Matrix Hit and Run for Sampling Polytopes and Its GPU Implementation

Mario Vazquez Corte[a], Ph. D. Luis V. Montiel[b]

[a]*Department of Computer Science,*
Instituto Tecnológico Autónomo de México - ITAM
Río Hondo 1, CDMX, México.
Email uumami@sonder.art
[b]*Faculty of Engineering,*
Universidad Nacional Auntónoma de México - UNAM
Ciudad Universitaria, CDMX, 04510, México
Email lvmontiel@utexas.edu

Abstract

We propose and analyze a new Markov Chain Monte Carlo algorithm that generates a uniform sample over full and non-full-dimensional polytopes. This algorithm, termed "Matrix Hit and Run" (MHAR), is a modification of the Hit and Run framework. For a polytope in $\mathbb{R}^n$ defined by m linear constraints, the regime $n^{1+\frac{1}{3}} \ll m$ has a lower asymptotic cost per sample in terms of soft-O notation ($\mathcal{O}^*$) than do existing sampling algorithms after a *warm start*. MHAR is designed to take advantage of matrix multiplication routines that require less computational and memory resources. Our tests show this implementation to be substantially faster than the *hitandrun* R package, especially for higher dimensions. Finally, we provide a python library based on PyTorch and a Colab notebook with the implementation ready for deployment in architectures with GPU or just CPU.

Keywords: Vector Simulation, Random Walk, MCMC, PyTorch, Parallel Processing, Graphics Processing Unit.

1. Introduction

A random sampling of convex bodies is employed in disciplines such as operations research, statistics, probability, and physics. Among random-sampling approaches, Markov Chain Monte Carlo (MCMC) is the fastest, most accurate, and most accessible to use (Chen et al., 2018). MCMC is often implemented using polytope sampling algorithms, which are used in volume estimation (see Lawrence, 1991; Lee and Vempala, 2018; Emiris and Fisikopoulos, 2014), convex optimization (Vempala and Bertsimas, 2004; Ma et al., 2019), contingency tables (Kannan and Narayanan, 2013), mixed integer programming (Huang and Mehrotra, 2015), linear programming (Feldman et al., 2005), hard-disk modeling (Kapfer and Krauth, 2013), and decision analysis (Tervonen et al., 2013; Montiel and Bickel, 2013a,b).

Sampling methods start by defining a Markov chain whose stationary distribution converges to a desired target distribution. In particular, Robert L. Smith proved in Smith (1984) that the Hit and Run algorithm converges to the uniform distribution on the relative interior of the polytope. Then, our concern is to efficiently draw many samples quickly, knowing that the samples will converge in the long run. Current sampling methods have two sources of computational complexity: *mixing-time*, which is the number of samples needed to lose the "dependency" between each draw, and *cost per sample*, which is the number of computational operations required to obtain a single sample.

We consider this work to be fundamental given the numerous applications in operations research that require exploratory polytope methods to complement diverse optimization models. For example, when the characterization of a joint probability distribution is not unique (Montiel and Bickel, 2013b), we require a method to generate a set of consistent joint distributions that help us to solve the problem for each possible description of the uncertainties. These models have applications on real options valuation (Montiel and Bickel, 2012) and oil field exploration and development (Montiel and Bickel, 2013a). Another application comes from the incomplete specification of a multi-attribute utility function in decision analysis, where the problem is understanding the decision maker's range of preferences to provide recommendations (Tervonen et al., 2013; Montiel and Bickel, 2014). Finally, in cooperative game theory, (Cid and Montiel, 2019) polytope exploratory methods can create an approximate objective function to optimize the negotiation strategy for a coalition of players.

This work presents an algorithm we call Matrix Hit and Run (MHAR)[1] for sampling

[1]The prefix "M" in reality stands for mentat, a type of human in Frank Herbert's Dune series who

full and non-full-dimensional polytopes. MHAR enhances the Hit-and-Run (HAR) algorithm described in Montiel and Bickel (2013a). We use the standard definition of a generic polytope $\Delta := \{x \in \mathbb{R}^n | Ax \leq b\}$, where $(A, b) \in \mathbb{R}^{m \times n} \times \mathbb{R}^{m \times 1}$, n is the number of elements of x, $m = m_E + m_I$ is the number of restrictions, m_E is the number of equality constraints, and m_I is the number of inequality constraints. We will use the term full-dimensional for polytopes with $m_E = 0$ and non-full-dimensional for polytopes with $m_E > 0$, assuming the system is non-degenerate.

For clarity and simplicity, HAR will refer to the algorithm presented in Montiel and Bickel (2013a), which extends Smith (1996) for non-full-dimensional polytopes. For ease of comparison, we use "soft-O" notation $\mathcal{O}^*$, which suppresses $\log(n)$ factors and other parameters like error bounds (Chen et al., 2018; Lee and Vempala, 2018; Lovász, 1999). To allow comparison with different algorithms, we assume that the polytope sampled by HAR and MHAR has received proper pre-processing, which means the polytope is in near isotropic position (Chen et al., 2018; Lee and Vempala, 2018; Tervonen et al., 2013). Although the procedure to set the polytope in a near isotropic position is beyond the scope of this article, we can briefly mention that the procedure involves applying an affine transformation to reshape the polytope into a more rounded body with less sharp corners, which in turn makes it easier for the random walk to explore the complete polytope. Additionally, all algorithms are compared from a *warm start*. We use $f \ll g$ notation to define a relation where $f \in \mathcal{O}(g)$. We also use the concept of *cost per iteration*, which refers to the number of computational operations needed to run the entire algorithm once. In the case of the HAR the *cost per sample* and the *cost per iteration* are the same since every iteration of the algorithm yields one sample (point). Meanwhile, for the MHAR the *cost per sample* and the *cost per iteration* differ since a single iteration could generate more than one sample. Sampling algorithms aim for efficient mixing-times, so that they can produce independent samples without dropping (also called "burning") too many of them and a low cost per iteration to

could simultaneously see the multiple probable paths the future may take.

draw samples fast (Geyer and Charles, 1992). Generating sample points from (close to) the desired target distribution, the MCMC algorithms need a *warm start* or *burning* period, which allows the algorithms to lose "dependency" from their starting point or initial conditions. Intuitively the *warm start* is intended to give the algorithms time to reach its equilibrium distribution, especially if it has started from a lousy starting point. Finally, we assume the existence of a random stream of bits that allows us to generate a random number in $\mathcal{O}(1)$.

The contribution of this work is six-fold:

- First, we introduce Matrix Hit-and-Run (**MHAR**).
- Second, we show that the cost per sample of MHAR is $\mathcal{O}^*\big(\min(m_I^{\omega-2}n^4, m_I n^{\omega+1})\big)$ for the full-dimensional scenario, and of $\mathcal{O}^*\big(\min(n^{\omega+2}, m_I n^{\omega+1})\big)$ for the non-full-dimensional one, where m, n are as described in the definition of Δ, ω represents a matrix multiplication coefficient as described in Table 1, and z is an expansion (or padding) hyper-parameter specified by the user.
- Third, we demonstrate that MHAR has a lower cost per sample than HAR if the hyper-parameter z is bigger than $\max(n, m)$.
- Fourth, we show that after proper pre-processing and a warm start, MHAR has a lower asymptotic cost per sample for the regime $n^{1+\frac{1}{3}} \ll m$ than do any of the published sampling algorithms (Chen et al., 2018).
- Fifth, we provide code for MHAR as a python library. It is ready for use in CPU or CPU-GPU architectures. The code is available in `https://github.com/uumami/mhar_pytorch`. The python package can be installed with the python package manager **pip install mhar**, the official site of the package is `https://github.com/uumami/mhar`
- Sixth, we present experimental results to assess the performance of MHAR against the *hitandrun* package used in Tervonen et al. (2013). MHAR was substantially faster in almost all scenarios, especially in high dimensions.

The remainder of this paper is organized as follows. Section 2 revises the definitions and some basic matrix-to-matrix operations. Section 3 revisits the cost per iteration and cost per sample of HAR. Section 4 provides a computational complexity analysis of MHAR. Section 5 compares MHAR against other algorithms developed for full-dimensional scenarios. Section 6 contains a back-to-back comparison of our implementation against the *"hitandrun"* library used in Tervonen et al. (2013) and a numerical analysis of the expansion hyper-parameter z. Section 7 identifies future work, and Section 8 presents our conclusions.

In the Appendix section, Appendix A presents additional proofs for lemmas and theorems. Appendix B presents additional optimal expansion experiments, and Appendix C presents additional performance experiments.

2. Preliminaries

This section formalizes the notation and briefly overviews computational complexity in matrix-to-matrix operations.

2.1. Polytopes

We define a polytope, the n-dimensional generalization of a polyhedron, as the intersection of half-spaces. Formally, a polytope is characterized by a set of m_E linear equality constraints and m_I linear inequality constraints in a Euclidean space ($\mathbb{R}^n$):

$$\begin{aligned}
\Delta^I &= \{x \in \mathbb{R}^n \mid A^I x \leq b^I,\ A^I \in \mathbb{R}^{m_I \times n},\ b^I \in \mathbb{R}^{m_I}\}, && (1)\\
\Delta^E &= \{x \in \mathbb{R}^n \mid A^E x = b^E,\ A^E \in \mathbb{R}^{m_E \times n},\ b^E \in \mathbb{R}^{m_E}\}, && (2)\\
\Delta &= \Delta^I \cap \Delta^D, && (3)
\end{aligned}$$

where Equations (1) and (2) are defined by the inequalities and equalities, respectively. The third equation defines the polytope of interest, the intersection of the two previous sets. Since Δ is the intersection of convex sets, then by construction, it is also convex.

For simplicity, we assume all polytopes to be bounded, non-empty, and characterized with no redundant constraints.

2.2. Matrix multiplication

We adopt the standard notation used in matrix multiplication. ω represents the matrix multiplication coefficient - which characterizes the number of operations required to multiply two $n \times n$ matrices. The complexity for such multiplication is of the order $\mathcal{O}(n^{\omega})$. Table 1 shows the theoretical bounds for many well-known multiplication algorithms (Lai et al., 2013; Le Gall, 2014).

Table 1: Asymptotic complexity of matrix multiplication algorithms

Matrix Multiplication Algorithms	
Algorithm	**Complexity**
naive	$\mathcal{O}(n^3)$
Strassen-Schoenhag	$\mathcal{O}(n^{2.807})$
Coppersmith-Winograd	$\mathcal{O}(n^{2.376})$
Le Gall	$\mathcal{O}(n^{2.373})$

In general, Knight (1995) showed that the number of operations needed to multiply two matrices with dimensions $m \times n$ and $n \times p$ is of $\mathcal{O}(d_1 d_2 d_3^{\omega-2})$, where $d_3 = \min\{m, n, p\}$ and $\{d_1, d_2\} = \{m, n, p\} \setminus \{d_3\}$. The special case of matrix-vector multiplication $d_3 = 1$ yields a $\mathcal{O}(mn)$ bound. Le Gall (2014) published an ω of 2.373, the smallest known until today.

It is possible to define a function μ that represents the matrix multiplication order of complexity for matrices $A \in \mathbb{R}^{n_1 \times n_2}$ and $B \in \mathbb{R}^{n_2 \times n_3}$ as

$$\mu_{A,B} = \begin{cases} n_1^{\omega-2} n_2 n_3 & \text{if } \min\{n_1, n_2, n_3\} = n_1, \\ n_1 n_2^{\omega-2} n_3 & \text{if } \min\{n_1, n_2, n_3\} = n_2, \\ n_1 n_2 n_3^{\omega-2} & \text{if } \min\{n_1, n_2, n_3\} = n_3. \end{cases} \tag{4}$$

Thus, we can express the complexity of the operation AB as $\mathcal{O}(\mu_{A,B})$.

In practice, only the naive and Strassen-Schoenhag algorithms are used because the constants hidden in the Big O notation are usually significantly big for large enough matrices to take advantage of. Moreover, many multiplication algorithms are impractical due to numerical instabilities (Chen et al., 2018). Fortunately, there are fast and numerically stable implementations of the Strassen-Schoenhag algorithm using GPUs available for open use (Li et al., 2011; Press et al., 2007; Huang et al., 1993).

2.3. Parallelism

Intuitively, in the single-worker paradigm, an operation is executed every time unit (also known as "tick" or "clock cycle"). But in the parallel paradigm, if we have V workers (also known as threads, processes, or cores), they can execute at most V operations in one unit of time. Yet the algorithm's total number of operations executed is not decreased in terms of $\mathcal{O}$ notation, as described in Theorem 2.1.

Theorem 2.1. *Let $\mathcal{O}(f(n))$ be the complexity of an algorithm and $V \in \mathbb{N}$ the number of workers in a parallel architecture. Then, $\mathcal{O}(\frac{1}{V}f(n)) = \mathcal{O}(f(n))$.*

Proof. This follows from the fact that $\frac{1}{V}$ is a constant. □

Theorem 2.1 implies that the complexity of executing any algorithm is unaffected by parallel architectures. This will become relevant in comparing HAR and MHAR complexities. The MHAR will execute z simultaneous walks that are tracked in a matrix regardless of the setting we are using (parallel or single-worker architectures). The fact that every algorithm step can be executed as a matrix operation makes the MHAR efficient in parallel architectures that use GPU, as introduced in the following subsection.

2.4. High-Performance Hardware and Software

As described by Nikolić et al. (2022) and Kimm et al. (2021), the current trend of performing matrix operations on GPU and TPUs (Tensor Processing Units) greatly exceeds the performance on CPUs. Hence, our intention is to provide a state-of-the-art implementation that can exploit these architectures for the benefit of the research community. In particular, the PyTorch library helps us to achieve the following:

- MHAR aims to minimize CPU-GPU communication, which is crucial for high-performance computing as described by Sharkawi and Chochia (2020).
- MHAR preserves memory usage in the GPU by computing almost everything there.
- Allow the hardware and software states, including matrix properties, precision, CPUs, GPUs, RAM, cache, and other relevant factors, to guide the determination of the optimal strategy for performing matrix operations.

3. Hit and Run (HAR)

This section revisits the HAR algorithm and calculates its cost per iteration and mixing time for non-full-dimensional polytopes, as defined in Montiel and Bickel (2013a).

3.1. Overview

HAR can be described as follows. A *walk* is initialized in a strict inner point of the polytope. At any iteration, a random direction is generated via independent normal variates. The random direction, along with the current point, generates a line set L, and its intersection with the polytope generates a line segment. The sampler selects a random point in L and repeats the process. After a warm start, HAR for full-dimensional convex bodies has a cost per iteration $\mathcal{O}(m_I n)$ and a cost per sample of $\mathcal{O}^*(m_I n^4)$, (Chen et al., 2018).

In general, the non-deterministic mixing time of HAR is of $\mathcal{O}^*(n^2\gamma_\kappa)$, where γ_κ is defined as

$$\gamma_\kappa = \inf_{R_{in}, R_{out} > 0} \left\{ \frac{R_{out}}{R_{in}} \ s.t.\ \mathcal{B}(x, R_{in}) \subseteq \Delta \subseteq \mathcal{B}(y, R_{out}) \text{ for some } x, y \in \Delta \right\},$$

where R_{in} and R_{out} are the radii of an inscribed and circumscribed ball of the polytope Δ, respectively, and $\mathcal{B}(q, R)$ is the ball of radius R containing the point q. In essence, γ_κ is the coefficient associated with the biggest inscribed ball and the smallest circumscribed ball of the polytope. That the mixing time depends on these parameters means that elongated polytopes are harder to sample. Implementations of HAR for convex bodies are typically analyzed after pre-processing and invoking a warm start, meaning that the body in question is brought to a near isotropic position in $\mathcal{O}^*(\sqrt{n})$, allowing the mixing time to be expressed as $\mathcal{O}^*(n^3)$, (see Chen et al., 2018; Lee and Vempala, 2022, 2018; Lovász and Simonovits, 1993; Lovász, 1999; Tervonen et al., 2013). For ease of comparison with the literature, the remainder of the paper assumes that the polytope has received proper pre-processing.

A HAR sampler must compute the starting point and find the line segment L at each iteration. Additionally, a thinning factor (also called "burning rate") $\varphi(n)$ must be included to achieve a fair almost uniform distribution over the studied space (Tervonen et al., 2013). This means that after a warm start, the algorithm needs to drop $\varphi(n)$ sampled points for each approximately i.i.d. observation. This thinning factor is known as the mixing-time, which is $\mathcal{O}^*(n^3)$ in the case of polytopes (see Chen et al., 2018; Tervonen et al., 2013; Lovász, 1999).

The HAR pseudocode described in Montiel and Bickel (2013a) for full and non-full-dimensional polytopes is re-stated in Algorithm 1. It samples a collection $\mathcal{X}$ of T uncorrelated points inside Δ. Above, we described the complexity of HAR for full-dimensional polytopes. However, the cost per iteration and the cost per sample in non-full-dimensional polytopes require analyzing the complexity of calculating the projection matrix $P_{\Delta^E} = I - A'^E(A^E A'^E)^{-1}A^E$, where A^E is the matrix defined by the equality constraints. Hence, P_{Δ^E} allows any direction vector h to be projected to the null space of A^E. The projection operation $P_{\Delta^E}h = d$ yields $A^E d = 0$. Then, $A^E(x + d) = b^E$. (This step is omitted if $m_E = 0$.)

Lemma 3.1. *If $m_E < n$, then the complexity of calculating P_{Δ^E} is $\mathcal{O}(m_E^{\omega-2}n^2)$.*

Algorithm 1: HAR pseudocode

```
Result: $\mathcal{X}$
Initialization;
$t \leftarrow 0$ (Sample point counter);
$j \leftarrow 0$ (Iteration counter);
$\mathcal{X} = \emptyset$;
Set the total sample size $T$;
Set a thinning factor $\varphi(n)$;
Find a strictly inner point of the polytope $\Delta$ and label it $x_{t=0,j=0}$;
if $(m_E > 0)$ then
    Compute the projection matrix $P_{\Delta^E}$
end
while $t < T$ do
    Generate the direction vector $h \in \mathbb{R}^n$;
    if $(m_E = 0)$ then
        $d = h$
    else
        $d = P_{\Delta^E} h$
    end
    Find the line set $L := \{x | x = x_{t,j} + \theta d, x \in \Delta \,\&\, \theta \in \mathbb{R}\}$;
    $j \leftarrow j + 1$;
    Generate a point uniformly distributed in $L \cap \Delta$ and label it $x_{t,j+1}$;
    if $j == \varphi(n)$ then
        $\mathcal{X} = \mathcal{X} \cup x_{t,j}$;
        $t \leftarrow t + 1$;
        $j \leftarrow 0$;
    end
end
```

Proof. See Appendix A.1. □

For simplicity, we will denote the complexity of computing P_{Δ^E} as $\mathcal{O}(\mu_{P_{\Delta^E}})$.

3.2. Non-full-dimensional HAR

We proceed to calculate the cost per sample of HAR for $m_E > 0$. We start by computing the cost per iteration in Lemma 3.2.

Lemma 3.2. *The cost per iteration of HAR for* $0 \leq m_E$ *is* $\mathcal{O}(\max\{m_I n, m_E^{\omega-2} n^2\})$.

Proof. See Appendix A.2 □

Having calculated the cost per iteration of HAR, we can proceed to Theorem 3.3.

Theorem 3.3. *The cost per sample of HAR for* $0 \leq m_E$ *is* $\mathcal{O}^*(n^3 \max\{m_I n, m_E^{\omega-2} n^2\})$ *after proper pre-processing and a warm start.*

Proof. According to Chen et al. (2018), the cost per sample of a sampling algorithm is its mixing time complexity multiplied by its cost per iteration. By Lemma 3.2, the cost per iteration is $\mathcal{O}(\max\{m_I n, m_E^{\omega-2} n^2\})$. Moreover, Tervonen et al. (2013) states that the mixing time, after a warm start, of HAR is $\mathcal{O}^*(n^3)$. Therefore, the cost per sample is $\mathcal{O}^*(n^3 \max\{m_I n, m_E^{\omega-2} n^2\})$.

Recall that if $m_E = 0$ the *cost per sample* is $\mathcal{O}^*(n^3 \max\{m_I n, 0\}) = \mathcal{O}^*(m_I n^4)$ that is the special case of HAR for full-dimensional polytopes. □

4. Matrix Hit-and-Run (MHAR)

This section details our new algorithm, the Matrix Hit-And-Run (**MHAR**). MHAR has a lower cost per sample than the HAR. The algorithm is tailored for exploiting cutting-edge matrix routines that exploit the architectures of machines like GPUs, cache memory, and multiple cores.

4.1. MHAR preliminaries

MHAR explores the polytope using simultaneous walks by drawing multiple vector directions d from the n-dimensional hypersphere. Each independent walk has the same mixing-time as with HAR, but a lower cost per iteration. Instead of running separate threads, we "batch" the walks by "expanding" vector x and d with z columns, creating the matrices $X = (x^1|\dots|x^k|\dots|x^z)$ and $D = (d^1|\dots|d^k|\dots|d^z)$. Super index k denotes the kth walk represented by the kth column in the expanded matrix. The "expansion" hyper-parameter z allows the concatenation of multiple directions d and samples x to form matrices D and $\mathcal{X}$, respectively. Each column of these matrices represents a walk over the polytope. This modification permits the use of efficient matrix-to-matrix operations to simultaneously project many directions d and find their respective line segments. These simultaneous z walks do not represent or need to be executed in a parallel architecture. The matrix X keeps track of the current position of each walk, and obtaining the next state is done by a series of matrix operations. This can be done in single-worker architectures. Here we use the notation $m = m_E + m_I$. Algorithm 2 presents the pseudocode for MHAR.

Algorithm 2: MHAR pseudocode

```
Result: $\mathcal{X}$
Initialization;
$t \leftarrow 0$ (Sample point counter);
$j \leftarrow 0$ (Iteration counter);
$\mathcal{X} = \emptyset$;
Set expansion hyper-parameter $z$ (bigger than $\max\{m, n\}$);
Set the total sample size $T$;
Set a thinning factor $\varphi(n)$;
Find a strictly inner point of the polytope $\Delta$ and label it $x_{t,j}$;
Set $x_{t,j}^k = x_{t,j},\ \forall k \in \{1, ..., z\}$;
Initialize $X_{t,j} = (x_{t,j}^1|...|x_{t,j}^k|...|x_{t,j}^z) \in \mathbb{R}^{n\times z}$;
if $(m_E > 0)$ then
    Compute the projection matrix $P_{\Delta^E}$
while $t < T$ do
    Generate $H = (h^1|...|h^k|...|h^z) \in \mathbb{R}^{n\times z}$, the direction matrix;
    if $(m_E = 0)$ then
        $D = H$;
    else
        $D = P_{\Delta^E} H = (d^1|...|d^k|...|d^z)$;
    Find the line sets $\left\{L^k := \{x | x = x_{t,j}^k + \theta^k d^k,\ x \in \Delta\ \&\ \theta^k \in \mathbb{R}\}\right\}_{k=1}^{z}$;
    $j \leftarrow j + 1$;
    Generate a point uniformly distributed in each $L^k$ and label it $x_{t,j}^k$ in $X_{t,j}$;
    if $j == \varphi(n)$ then
        $\mathcal{X} = \mathcal{X} \cup \{x_{t,j}^1, ..., x_{t,j}^z\}$;
        $t \leftarrow t + z$;
        $j \leftarrow 0$;
```

Even with limited memory, good performance can still be achieved by selecting a smaller expansion hyper-parameter z than $\max\{m, n\}$, as demonstrated in Section 6 of the application. Finding the best value for z in applications will require some experimentation and fine-tuning by the user since it depends on the hardware available. For the theoretical analysis, we will keep z bigger than $\max\{m, n\}$ since the same complexity bounds would be obtained for any z that satisfies these constraints, as shown in the following subsections.

4.2. Starting point

The cost of finding the starting point is generally excluded from the complexity analysis because it is independent of the mixing-time. However, we present it here for completeness even though the literature assumes a warm start in determining cost per sample (Chen et al., 2018; Tervonen et al., 2013; Lee and Vempala, 2018).

MHAR needs to be initialized by a point in the relative interior of the polytope. We suggest Chebyshev's center of the polytope, the center of the largest inscribed ball. Chebyshev's center can be formulated as a linear optimization problem for polytopes and solved using standard methods. Chebyshev's center is presented in Model (5).

$$\begin{aligned} &\max_{x\in\mathbb{R}^n, r\in\mathbb{R}} \; r, \\ &s.t \quad A^E x = b^E, \\ &(a_i^I)^T x + r||a_i^I||_2 \leq b_i^I, \; \forall i = 1, ..., m_I, \end{aligned} \tag{5}$$

where a_i^I and b_i^I represent the ith row of matrix A^I and ith entry from vector b^I, respectively. Model (5) has the original m restrictions plus one additional variable "r". Hence, the size of the problem has m constraints and $n+1$ variables. Then, calculating the $||\cdot||_2$ coefficients takes $\mathcal{O}(mn)$. Thus, it can be formulated and solved in $\mathcal{O}(n^\omega)$ using Vaidya's algorithm (Chen et al., 2017) for linear optimization. After solving Model (5), we use x as the starting point $x_{t=0,j=0}$ for all walks and draw independent walking directions. The matrix $X_{t,j} \in \mathbb{R}^{n\times z}$ introduced in Algorithm 2 is the algorithmic version of X, and it summarizes the state of all walks, where each kth column represents the current point of walk k at iteration $\{t, j\}$. Formally we say $X_{t,j} = (x_{t,j}^1|...|x_{t,j}^k|...|x_{t,j}^z)$ where $x_{t,j}^k \in \mathbb{R}^{n\times 1}$ $\forall k \in \{1, ..., z\}$.

4.3. Generating D

Because the target distribution of HAR and MHAR is the uniform distribution, we follow the procedure established in Montiel and Bickel (2013a) and Lovász (1999) which uses the Margsalia method (Marsaglia, 1972) to generate a random vector h from the hypersphere by generating n i.i.d. samples from a standard normal distribution $\mathcal{N}(0,\ 1)$. However, instead of generating a single direction vector $d \in \mathbb{R}^n$, we create matrices $H,\ D \in \mathbb{R}^{n\times z}$, where each element of the matrix corresponds to an independent execution of the Box-Muller method (Chay et al., 1975) bounded by $\mathcal{O}(nz)$. If the polytope is fully dimensional, $H = D$, and no projection operation is needed. Otherwise, the projection matrix P_{Δ^E} is calculated as in Lemma 3.1 that bounds the number of operations to $\mathcal{O}(m_E^{\omega-2}n^2)$.

Matrices H and D can be visualized as

$$H = (h^1|...|h^k|...|h^z),\ h^k \in \mathbb{R}^n,\ \forall k \in \{1, ..., z\}, \tag{6}$$

$$D = P_{\Delta^E}H = (d^1|...|d^k|...|d^z),\ d^k \in \mathbb{R}^n,\ \forall k \in \{1, ..., z\}. \tag{7}$$

Each column h^k can be projected by the operation $D = P_{\Delta^E}H$. Hence, each column of D satisfies the restrictions in Δ^E and serves as a direction d for an arbitrary walk k. In principle, z can be any number in $\mathbb{N}$, where $z = 1$ is the special case that recovers the original HAR.

Lemma 4.1. *The complexity of generating matrix D in MHAR given P_{Δ^E} and* $\max\{m_I, n\} \leq z$ *is $\mathcal{O}(nz)$ if $m_E = 0$, and $\mathcal{O}(n^{\omega-1}z)$ if $m_E > 0$.*

Proof. See Appendix A.3. □

Lemma 4.1 shows that if $m_E > 0$, the cost of generating new directions d does not scale as if had used z simultaneous and independent HARs. In the HAR case, the operations required would have been carried out in $\mathcal{O}(z\mu_{P_{\Delta^E},h}) = \mathcal{O}(zn^2)$, averaging $\mathcal{O}(\frac{zn^2}{z}) = \mathcal{O}(n^2)$ per direction. In contrast, MHAR is $\mathcal{O}(n^{\omega-1}z)$, averaging $\mathcal{O}(\frac{n^{\omega-1}z}{z}) =$

$\mathcal{O}(n^{\omega-1})$ per direction. When $m_E = 0$, the numbers of operations for both cases are the same.

4.4. Finding the line sets

Given matrices X and D, we now obtain the line sets $\{L^k\}_{k=1}^{z}$:

$$\left\{L^k := \{x | x = x^k + \theta^k d^k,\ x \in \Delta,\ \text{and } \theta^k \in \mathbb{R}\}\right\}_{k=1}^{z}. \tag{8}$$

Each θ^k characterizes the line set for column x^k. The "expanded" or "padded" column-wise representation of restrictions Δ^I is

$$A^I X = \begin{pmatrix} a_1^I x^1 & \dots & a_1^I x^k \\ \vdots & \ddots & \vdots \\ a_{m_I}^I x^1 & \dots & a_{m_I}^I x^k \end{pmatrix} \leq \begin{pmatrix} b_1^I \\ \vdots \\ b_{m_I}^I \end{pmatrix} = b^I, \tag{9}$$

where each element from the left matrix must be less than or equal to the corresponding element (row-wise) in vector b^I. The restrictions for an arbitrary x^k can be rewritten row-wise so that the left side and right side are scalars:

$$a_i^I x^k \leq b_i^I,\ \forall i \in \{1, \dots, m_I\}. \tag{10}$$

Then, each θ^ks must satisfy

$$(a_i^I x^k + \theta^k a_i^I d^k) < b_i^I,\ \forall i \in \{1, \dots, m_I\}. \tag{11}$$

Rearranging the terms obtains restrictions for each walk k, where each θ^k must be bounded by its respective set of lambdas $\{\lambda_i^k\}_{i=1}^{m_I}$, as follows:

$$\theta^k < \lambda_i^k = \frac{b_i^I - a_i^I x^k}{a_i^I d^k}, \quad if\ a_i^I d^k > 0, \tag{12}$$

$$\theta^k > \lambda_i^k = \frac{b_i^I - a_i^I x^k}{a_i^I d^k}, \quad if\ a_i^I d^k < 0. \tag{13}$$

Hence, a walk's boundaries are represented by

$$\lambda^k_{min} = \max \{\lambda^k_i \mid a^I_i d^k < 0\}, \tag{14}$$

$$\lambda^k_{max} = \min \{\lambda^k_i \mid a^I_i d^k > 0\}. \tag{15}$$

These lambdas can be used to construct the intervals $\Lambda^k = (\lambda^k_{\text{min}}, \lambda^k_{\text{max}}),\ k \in \{1, ..., z\}$. By construction, if $\theta^k \in \Lambda^k$ and $x^k \in \Delta$, then $x^k + \theta^k d^k \in L^k$, since $A^I(x^k + \theta^k d^k) \leq b^I$ and $A^E(x^k + \theta^k d^k) = b^E$. The line segment can be found simply by evaluating $\{\Lambda^k\}^z_{k=1}$, because x^k and D were computed previously. We can now state Lemma 4.2.

Lemma 4.2. *The complexity of generating all line sets* $\{L^k\}^z_{k=1}$ *in MHAR given* D*,* X*, and* $\max\{m_I, n\} \leq z$ *is bounded by* $\mathcal{O}(m_I n^{\omega-2} z)$ *if* $n \leq m_I$*, and by* $\mathcal{O}(m_I^{\omega-2} n z)$ *otherwise.*

Proof. See Appendix A.4. □

Lemma 4.2 bounds the complexity of finding the line sets at any iteration of MHAR. This leaves only analyzing the cost of choosing a new sample.

4.5. Choosing samples

The following lemma bounds the complexity of choosing a new $X_{t,j+1}$ or $X_{t+z,0}$ given $\Lambda^k\ \forall k \in \{1, ..., z\}$. The new samples will be padded to create the matrix $X_{t,j+1} = (x^1_{t,j+1}| \ldots |x^k_{t,j+1})$ to be used in the next iteration.

Lemma 4.3. *Sampling* z *new points given* $\{\Lambda^k\}^z_{k=1}$ *has complexity* $\mathcal{O}(zn)$*.*

Proof. See Appendix A.5. □

Having concluded the complexity analysis for each step of the loop, we next calculate the cost per iteration and proceed to measure the cost per sample.

4.6. Iteration and sampling costs of MHAR

The asymptotic behavior of each operation that comprises the main loop of MHAR when $\max\{n, m_I\} \leq z$ is presented in Table 2. The cost of finding the starting point is excluded as in (Chen et al., 2018; Tervonen et al., 2013).

Table 2: Asymptotic cost per sample of MHAR at each step

MHAR complexity at each step, $(n, m) < z$				
Steps	$m_E = 0$, $n \leq m_I$.	$m_E = 0$, $n > m_I$.	$m_E > 0$, $n \leq m_I$.	$m_E > 0$, $n > m_I$.
1. Projection matrix	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m_E^{\omega-2}n^2)$	$\mathcal{O}(m_E^{\omega-2}n^2)$
2. Generating D	$\mathcal{O}(nz)$	$\mathcal{O}(nz)$	$\mathcal{O}(n^{\omega-1}z)$	$\mathcal{O}(n^{\omega-1}z)$
3. Finding $\{L^k\}_{k=1}^z$	$\mathcal{O}(m_I n^{\omega-2}z)$	$\mathcal{O}(m_I^{\omega-2}nz)$	$\mathcal{O}(m_I n^{\omega-2}z)$	$\mathcal{O}(m_I^{\omega-2}nz)$
4. Sampling all $x_{t,j+1}^k$	$\mathcal{O}(nz)$	$\mathcal{O}(nz)$	$\mathcal{O}(nz)$	$\mathcal{O}(nz)$

The following lemmas will help bind the cost per iteration of MHAR. Lemmas 4.4 and 4.5 establish the full-dimensional case for $(n \leq m_I)$ and $(n > m_I)$, respectively. Lemmas 4.8 and 4.9 do likewise in the non-full-dimensional case for $(n \leq m_I)$ and $(n > m_I)$, respectively. Figure 1 summarizes these results as follows.

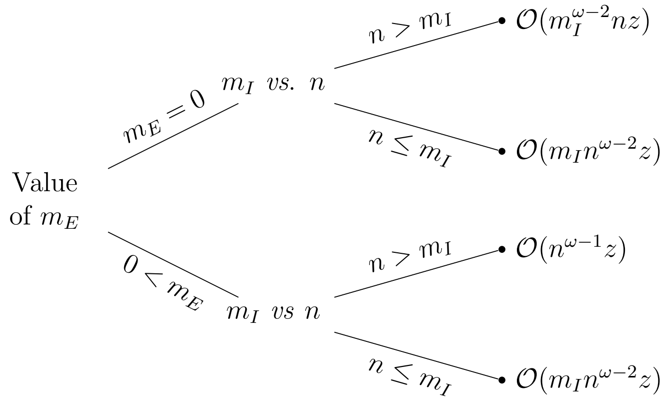

Value of m_E
$m_E = 0$
$0 < m_E$
m_I vs. n
m_I vs n
$n > m_I$
$n \leq m_I$
$\mathcal{O}(m_I^{\omega-2}nz)$
$\mathcal{O}(m_I n^{\omega-2}z)$
$\mathcal{O}(n^{\omega-1}z)$
$\mathcal{O}(m_I n^{\omega-2}z)$

Figure 1: Asymptotic behavior of the cost per iteration of MHAR.

Lemma 4.4. *Assume $m_E = 0$, $\max\{n, m\} < z$, and $n \leq m_I$. Then, the cost per iteration of MHAR is $\mathcal{O}(m_I n^{\omega-2} z)$, which is the number of operations needed for finding all line sets $\{L^k\}_{k=1}^{z}$.*

Proof. See Appendix A.6. □

Lemma 4.5. *Assume $m_E = 0$, $\max\{n, m\} < z$, and $n > m_I$. Then, the cost per iteration of MHAR is $\mathcal{O}(n m_I^{\omega-2} z)$, which is the number of operations needed for finding all line sets $\{L^k\}_{k=1}^{z}$.*

Proof. See Appendix A.7 □

Corollary 4.6. *Assume $m_E = 0$ and $\max\{n, m\} < z$. Then, the cost per iteration of MHAR is bounded by the cost of finding all line sets $\{L^k\}_{k=1}^{z}$.*

Proof. The proof follows from Lemmas 4.4 and 4.5. □

We proceed to find the cost per iteration for the non-full-dimensional case $m_E > 0$.

Lemma 4.7. *Assume $m_E < n$ and $(m, n) < z$. Then, the cost of calculating the projection matrix P_{Δ^E} is bounded by the cost of generating D.*

Proof. See Appendix A.8. □

Lemma 4.8. *Assume $m_E > 0$, $\max\{n, m\} < z$, and $n \leq m_I$. Then, the cost per iteration of MHAR is $\mathcal{O}(m_I n^{\omega-2} z)$, which is the number of operations needed for finding all line sets $\{L^k\}_{k=1}^{z}$.*

Proof. See Appendix A.9. □

Lemma 4.9. *Assume $m_E > 0$, $\max\{n, m\} < z$, and $n > m_I$. Then, the cost per iteration of MHAR is $\mathcal{O}(n m_I^{\omega-2} z)$, which is the number of operations needed for generating D.*

Proof. See Appendix A.10. □

We can now proceed to the main results of the paper, given in Theorem 4.10.

Theorem 4.10. *If* $\max\{n, m\} < z$*, then after proper pre-processing and a warm start, the cost per sample of MHAR is:*

$$\begin{cases} \mathcal{O}^*(m_I n^{\omega+1}), & if\ m_E = 0\ and\ n \leq m_I \\ \mathcal{O}^*(n^{\omega+2}), & if\ m_E = 0\ and\ n > m_I \\ \mathcal{O}^*(m_I n^{\omega+1}), & if\ m_E > 0\ and\ n \leq m_I \\ \mathcal{O}^*(m_I^{\omega-2} n^4), & if\ m_E > 0\ and\ n > m_I. \end{cases} \tag{16}$$

Proof. From Lemmas 4.4, 4.5, 4.8, and 4.9 we have the cost per iteration of MHAR for all four cases:

$$\begin{cases} \mathcal{O}(m_I n^{\omega-2} z), & if\ m_E = 0\ and\ n \leq m_I \\ \mathcal{O}(n^{\omega-1} z), & if\ m_E = 0\ and\ n > m_I \\ \mathcal{O}(m_I n^{\omega-2} z), & if\ m_E > 0\ and\ n \leq m_I \\ \mathcal{O}(m_I^{\omega-2} n z), & if\ m_E > 0\ and\ n > m_I. \end{cases} \tag{17}$$

It was stated that each walk from the "expansion" is independent of the other ones after a warm-start. Then, each individual walk has a mixing time of $\mathcal{O}^*(n^3)$. Then it suffices to apply the rule for Big-O products between the cost per iteration and the mixing time and divide the coefficient by the expansion hyper-parameter z, which is the number of points obtained at each iteration. Hence, multiplying each case in Equation (17) by $\frac{n^3}{z}$ obtains the desired result. □

Theorem 4.10 characterizes the cost per sample of MHAR for all parameter and hyper-parameter values. The Theorem shows that MHAR is always at least as efficient as HAR, and more efficient for $\omega \in (2, 3)$. So, for $\omega = 3$, the schoolbook method, MHAR is not asymptotically more efficient than HAR. Intuitively this is caused by "the expansion," which permits matrix-to-matrix multiplications instead of isolated matrix-to-vector operations when finding the line sets L or the directions D. Furthermore, this approach allows efficient cache usage and state-of-the-art GPU matrix multiplication algorithms. Figure 2 graphically depicts the results of the theorem.

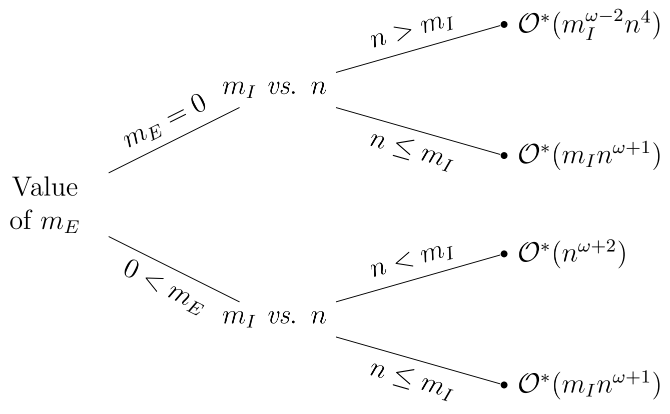

Value of m_E
$m_E = 0$
$0 < m_E$
m_I vs. n
m_I vs. n
$n > m_I$
$n \leq m_I$
$n < m_I$
$n \leq m_I$
$\mathcal{O}^*(m_I^{\omega-2} n^4)$
$\mathcal{O}^*(m_I n^{\omega+1})$
$\mathcal{O}^*(n^{\omega+2})$
$\mathcal{O}^*(m_I n^{\omega+1})$

Figure 2: Asymptotic behavior of the cost per sample of MHAR after a warm start.

4.7. Parallelism in HAR and MHAR

We already analyzed the difference between HAR and MHAR under the single-worker paradigm. Now that we can analyze both under the parallel paradigm. By Theorem 2.1 the complexity of the HAR in a parallel architecture remains the same. This happens, regardless of the parallel version of the HAR; V simultaneous walks (or workers) performing simultaneous operations or some arbitrary number of walks where the V workers share operations. This shows us that the MHAR algorithm is not just a parallel version of the HAR, but something different that allows the z simultaneous walks to share operations and information, hence lowering algorithm complexity. The matrix nature of the MHAR and the z hyper-parameter can be easily exploited by parallel architectures.

In its current state, MHAR shows high performance because the matrix operations have been fine-tuned and optimized in terms of software and hardware. However, we expect that advances in high-performance machine learning related to matrix multiplication and GPU architectures (Chrzeszczyk and Chrzeszczyk, 2013; Mittal and Vaishay, 2019) will automatically improve the performance of our work in the times to come.

5. MHAR Complexity Benchmarks

This section benchmarks the asymptotic behavior of MHAR against that of seven state-of-the-art algorithms. Some of these algorithms cover additional convex figures, like spheres or cones. However, we restrict our focus to polytopes because they are the target of MHAR. For an in-depth analysis of each algorithm see Chen et al. (2018). We prioritize the full-dimensional case ($m = m_I, m_E = 0$) because few algorithms are designed for the non-full-dimensional scenario and their analysis is outside our scope. Table 3 is adapted from Chen et al. (2018) and includes the notation established in Tervonen et al. (2013) and Montiel and Bickel (2013a). The authors of RHCM (Lee and Vempala, 2018), John's walk (Gustafson and Narayanan, 2022), Vaidya walk, and John walk; omitted $m < n$, which is also outside of our scope. Note that John's walk and John walk are different algorithms.

In Section 4 we showed that MHAR has a lower cost per sample than HAR for efficient matrix multiplication algorithms. Furthermore, because the Ball walk (Lovász and Simonovits, 1993) has the same cost per sample as HAR, we can derive the next corollary.

Corollary 5.1. *The cost per sample of MHAR is as low as the cost per sample of the Ball walk, after a warm start, if* $\max\{n, m\} < z$. *And strictly lower if efficient matrix-to-matrix algorithms are used* $\big(\omega \in (2, 3)\big)$.

Proof. This follows from comparing Theorem 4.10 against the complexity of the Ball walk. □

The following lemma shows that MHAR has a lower cost per sample than John's walk.

Lemma 5.2. *For* $\max\{n, m\} < zm$, *and* $n < m$, *MHAR has a lower cost per sample than does John's walk after proper pre-processing, warm start, and ignoring the logarithmic and error terms.*

Proof. See Appendix A.11. □

Table 3: Asymptotic behavior of random walks

Random walks behaviour			
Walk	**Mixing time**	**Cost per iteration**	**Cost per sample**
MHAR with $n > m$	n^3	$m^{\omega-2}nz$	$m^{\omega-2}n^4$
MHAR with $n \leq m$	n^3	$mn^{\omega-2}z$	$mn^{\omega+1}$
Ball walk	n^3	mn	mn^4
HAR	n^3	mn	mn^4
Dikin walk with $n \leq m$	mn	$mn^{\omega-1}$	m^2n^{ω}
RHCM with $n \leq m$	$mn^{\frac{2}{3}}$	$mn^{\omega-1}$	$m^2n^{\omega-\frac{1}{3}}$
John's walk with $n \leq m$	n^7	$mn^4 + n^8$	$mn^{11} + n^{15}$
Vaidya walk with $n \leq m$	$m^{\frac{1}{2}}n^{\frac{3}{2}}$	$mn^{\omega-1}$	$m^{1.5}n^{\omega+\frac{1}{2}}$
John walk with $n \leq m$	$n^{\frac{5}{2}}\log^4(\frac{2m}{n})$	$mn^{\omega-1}\log^2(m)$	$mn^{\omega+\frac{3}{2}}$

Extension to results by (Chen et al., 2018). Table 3 contains upper bounds on the cost per sample (after warm start) for nine random walk algorithms applied to polytopes. For MHAR, $\max\{n, m\} < z$ is assumed. Logarithmic terms in the cost per sample are ignored. Bounds in terms of the condition number of the set are avoided for MHAR, Ball walk, and HAR, since after proper pre-processing the condition number is bounded by n.

In the regime of $n \ll m$, the overall upper bound complexity for the cost per sample is represented by John walk $\ll$ Vaidya walk $\ll$ Dikin walk (Chen et al., 2018). We now show that for $n \ll m$, MHAR has a lower cost per sample than John walk.

Lemma 5.3. *For $\max\{n, m\} < z$ and the regime $n \ll m$, MHAR has a lower cost per sample than does the John walk after proper pre-processing, warm start, and ignoring logarithmic and error terms.*

Proof. See Appendix A.12. □

Corollary 5.4. *For $\max\{n, m\} < z$ and the regime $n \ll m$, then MHAR $\ll$ John Walk $\ll$ Vaidya walk $\ll$ Dikin walk after proper pre-processing, warm start, and ignoring logarithmic and error terms.*

Proof. This follows from Lemma 5.3. □

We proceed to compare MHAR and RHMC for the regime $n^{1+\frac{1}{3}} \ll m$.

Lemma 5.5. *For* $\max\{n, m\} < z$ *and* $n^{1+\frac{1}{3}} \ll m$*, then MHAR* $\ll$ *RHMC after proper pre-processing, warm start and ignoring logarithmic and error terms.*

Proof. From proper pre-processing, $n \ll m$, and $n, m < z$, MHAR's cost per sample is $\mathcal{O}^*(mn^{\omega+1})$, and RHMC's is $\mathcal{O}(m^2 n^{\omega-\frac{1}{3}})$. Note that $mn^{\omega+1} \in \mathcal{O}(m^2 n^{\omega-\frac{1}{3}})$, because $n^{1+\frac{1}{3}} \ll m$. Therefore, when ignoring the logarithmic and error terms, MHAR has a lower cost per sample. □

From corollaries 5.1 and 5.2, MHAR $\ll$ Ball walk and MHAR $\ll$ HAR, regardless of the regime between m and n. And MHAR $\ll$ John's Walk for the regime $n \leq m$. From corollary 5.4, MHAR $\ll$ John Walk $\ll$ Vaidya walk $\ll$ Dikin walk if $n < m$. Finally, by Lemma 5.5, if $n^{1+\frac{1}{3}} \ll m$, then MHAR $\ll$ RHMC. Then, if $n^{1+\frac{1}{3}} \ll m$ we have an analytic guarantee that MHAR has a lower cost per sample than all of the other algorithms in Table 3.

6. MHAR Empirical Test

This section details a series of experiments to compare MHAR against the *hitandrun* library used by Tervonen et al. (2013). We compare the running times in simplices and hypercubes of different dimensions and for various values of the expansion hyper-parameter z. We also test the robustness of MHAR by conducting empirical analyses similar to those in Tervonen et al. (2013). MHAR experiments were run in a Colab Notebook equipped with an Nvidia P100 GPU, and a processor Intel® Xeon® CPU running at 2.00 GHz, and 14 GB of RAM. Due to its apparent incompatibility with the Colab Notebook, the *hitandrun*experiments were run in a computer equipped with an Intel® Core™ i7-7700HQ CPU running at 2.80 GHz and 32 GBs of RAM. All experiments used 64 bits of precision.

The *hitandrun* package consumed a considerable amount of memory, therefore we chose the computer with the most RAM available (32 GBs and no GPU). Meanwhile, we used Colab cloud GPUs to test the MHAR. We could not test the *hitandrun* with dimensions as big as the MHAR due to a lack of memory capacity. We are aware of the implications of comparing the algorithms on different computers. However, when the expansion hyper-parameter $z = 1$ the HAR is equivalent to the MHAR, which generates a baseline for comparison of the two algorithms even if they are running on different architectures. We keep the established notation of $m = m_I + m_E$.

We formally define the n-simplex and the n-hypercube as

$$n\text{-simplex} = \{x \in \mathbb{R}^n \| \sum x_i = 1, x \geq 0\}, \tag{18}$$

$$n\text{-hypercube} = \{x \in \mathbb{R}^n \| x \in [-1, 1]^n\}. \tag{19}$$

6.1. The Code

MHAR code was developed using python, and the PyTorch library was chosen because of its flexibility, power, and popularity (Paszke et al., 2019). PyTorch also works in a CPU without the need for a GPU, although the latter is more suitable for large samples in high dimensions. MHAR experiments were performed without observing any numerical instabilities, and the maximum error found for the inversion matrix was on the order 1e-16, which is robust enough for most applications. Operations such as matrix inversion, random number generation, matrix-to-matrix multiplication, and point-wise operations were carried out in the GPU. The only operations that needed to be carried out in the CPU were reading the constraints and saving the samples to disk. The exact algorithms used for these operations are decided by the PyTorch framework and depend on the user's hardware, matrix sizes, and back-end used to install the libraries.

For the rest of this section, the acronyms MHAR and HAR refer to the actual implementations and not the abstract algorithms. The code is available in `https://github.com/uumami/mhar_pytorch`.

6.2. The expansion

The expansion (padding) hyper-parameter z determines the number of simultaneous walks the algorithm performs. We generated 10 MHAR runs for each dimension (5, 25, 50, 100, 500, 1000) and each expansion value (z) on simplices and hypercubes. Recall that if $z = 1$ then the MHAR is the same algorithm as the HAR, therefore the experiments are equivalent to the HAR being executed in the GPU. At each run, we calculated the average number of samples per second as follows:

$$Avg.\ Samples\ per\ Second = \frac{Total\ Samples}{Time} = \frac{z \times \varphi \times T}{Time}.$$

For example, if $z = 100$, the thinning parameter $\varphi = 30,000$, and the number of iterations $T = 1$, then the $Total\ Samples = 3,000,000$. If the experiment took 1,000 seconds, the average number of samples per second would be $3,000$.

Figures 3,4, 5 and 6 show box-plots for the experiments in dimensions 5 and 1000 for the simplex and the hypercube. The box-plots for the simplex and the hypercube in dimensions 25, 50, 100, and 500 can be found in Appendix B.

The box in the box-plots shows the 25%, 50%, and 75% percentiles, and the diamonds mark the outliers. For small values of n, more significant expansions yielded more average samples per second. However, for some dimensions in the simplex and the hypercube, there was a value of z for which efficiency was lower. We conjecture that at some point, large values of z could cause memory contention in the GPU. The main overhead in GPU algorithms is the communication between the CPU (host) and GPU (device), therefore z must be used to execute as many simultaneous walks as possible but without using excessive memory. Ideally, z should be big enough to use all the available memory (or GPU memory) but not overflow it. This can be achieved by some experimentation and GPU profiling on the available hardware.

6.3. Performance Test MHAR vs HAR

To compare MHAR and HAR we generated 10 simulations for different dimensions, and two types of polytopes (simplex and hypercubes). For the simplex, we tested dimensions: 5, 25, 50, 100, and 250, and for the hypercube, we tested dimensions: 5, 25, 50, 100, 500, and 1000. The *hitandrun* routines for sampling the simplex exhibited an extreme drop in performance at dimensions higher than 100 and memory contention at dimensions higher than 300.

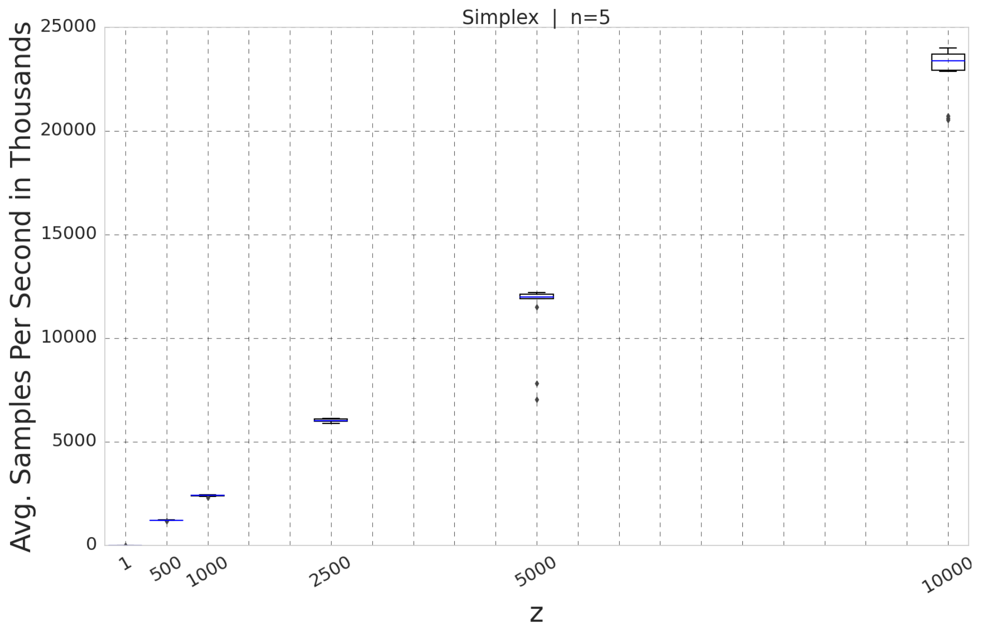


Figure 3: Box-plots for simplices in dimension 5 comparing expansion behavior for different values of the expansion hyper-parameter z.

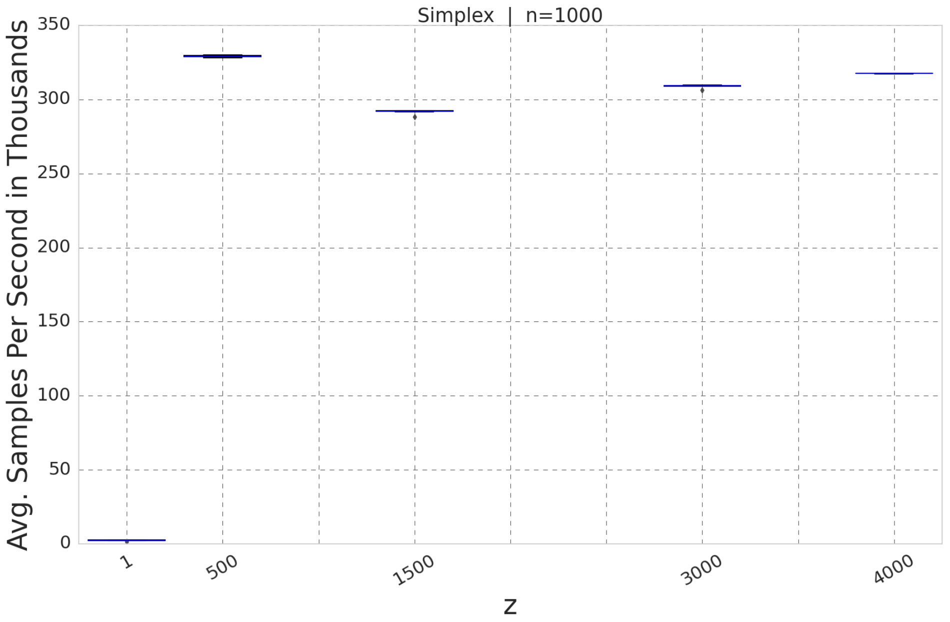


Figure 4: Box-plots for simplices in dimension 1000 comparing expansion behavior for different values of the expansion hyper-parameter z.

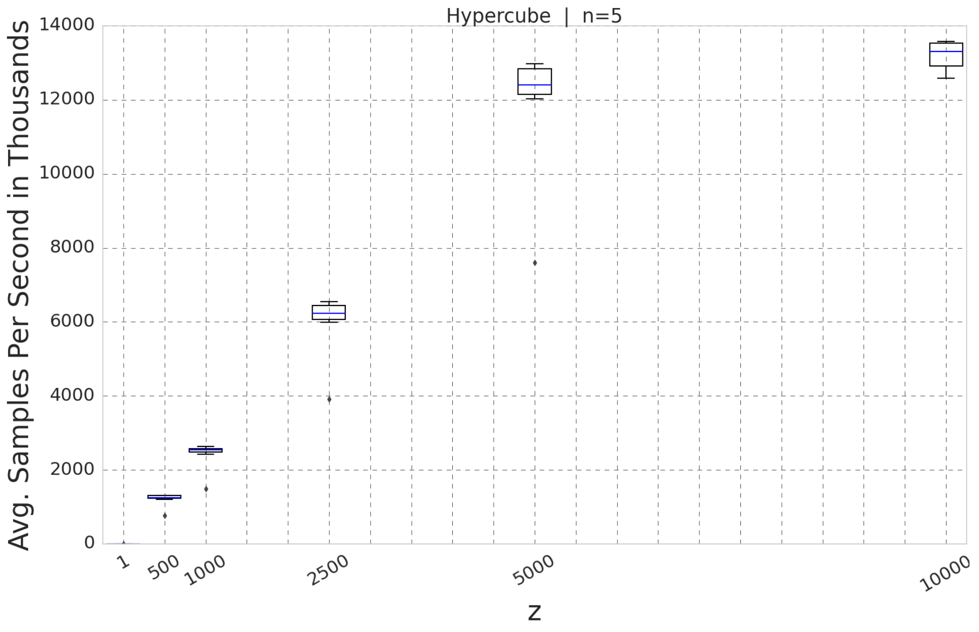


Figure 5: Box-plots for hypercubes in dimension 5 comparing expansion behavior for different values of the expansion hyper-parameter z.

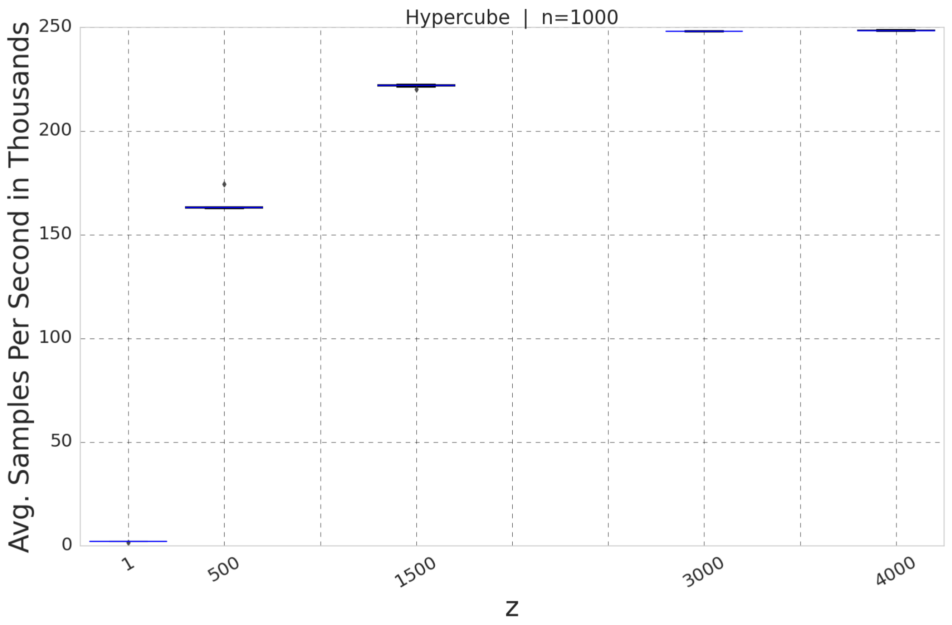


Figure 6: Box-plots for hypercubes in dimension 1000 comparing expansion behavior for different values of the expansion hyper-parameter z.

For *hitandrun*, the total number of samples equals the number of iterations times the thinning parameter. Because *hitandrun* does not make use of the GPU, the times are dependent on the CPU. Before running a given combination of convex body and dimension in MHAR, we selected the expansion hyper-parameter z^* that had the highest average sampled points per second according to our expansion experiments. So, the z^* can differ by dimension. We used $\varphi = 30,000$ and $T = 1$. Table 4 summarizes the results.

Table 4 shows substantial performance gains for MHAR. For the simplex, the gains were greater at higher dimensions. The performance ratio (average samples per second for MHAR divided by that for HAR) was 23 for $n = 5$ and 2.5 million for $n = 250$. For the hypercube, performance gain for MHAR was also greater at higher dimensions. Nevertheless, the performance ratio was 14 for $n = 5$ and 248 for $n = 1,000$.

To test the limits of our implementation, we conducted an additional set of experiments for lower and higher dimensions and different expansion hyper-parameter. We present these results in Appendix C.

Table 4: Performance of MHAR versus HAR for the optimal value of z^*

Figure	n	z	Perf. ratio MHAR/HAR mean	Avg. Samples Per Second MHAR mean	HAR mean	MHAR Std. Dev.	HAR Std. Dev.
Hypercube	5	10,000	14	13,206,090	931,369	376,069	57,728
Hypercube	25	5,000	29	10,839,474	373,128	1,236,620	77,787
Hypercube	50	2,500	22	5,151,517	235,742	612,2422	20,636
Hypercube	100	4,000	117	4,363,526	37,368	10,620	1,487
Hypercube	500	4,000	95	621,555	6,529	783	158
Hypercube	1,000	4,000	248	248,514	1,001	183	18
Simplex	5	10,000	23	22,878,783	988,581	1,258,482	126,255
Simplex	25	10,000	1,344	24,338,761	18,115	168,301	409
Simplex	50	10,000	12,631	13,425,901	1,063	16,404	17
Simplex	100	3,000	128,349	7,255,837	57	135,617	1
Simplex	250	4,000	2,551,224	2,656,449	1	4,441	0

It is important to recall that although MHAR and HAR were tested using different computers, the ratio MHAR/HAR shows improvement as we increase n, this could only be explained by the difference in performance of the algorithms.

6.4. Independence Test

To assess the convergence of MHAR to a uniform distribution, we conducted the Friedman-Rafsky two-sample Minimum Spanning Tree (MST) test (see Friedman and Rafsky, 1979) as was done in Tervonen et al. (2013). The test compares a sample S from MHAR with a sample U from the uniform distribution with support on the polytope Δ, notice that this test only works for very simple polytopes such as hypercubes and simplices. The test defines an MST using S and U by counting the number of samples on equal and different sectors of the support to assess if both samples come from the same distribution. Then the null hypothesis "Both samples are drawn from the same distribution" is not rejected or rejected depending on the *W-value*. We replicate the setting established by Tervonen et al. (2013). A threshold of $-1.64 \leq$ *W-value* to not reject the null hypothesis corresponding to a 5% significance test with a one-sided alternative for an approximately normally distributed test statistic W.

Rubin (1981) shows a method to draw a uniform sample U from the hypercube and the simplex using known statistical methods. Hence, we generated 10 simulations in simplices and hypercubes with dimensions: 5, 15, 25, and 50, for a total of 80 simulations. We used a single expansion hyper-parameter (z) of $1,000$; and a "burning rate" $\varphi = (n-1)^3$ for the simplices, and $\varphi = n^3$ for the hypercubes. Each simulation draws a total of $5,000$ samples that were compared to an independently generated sample U each time.

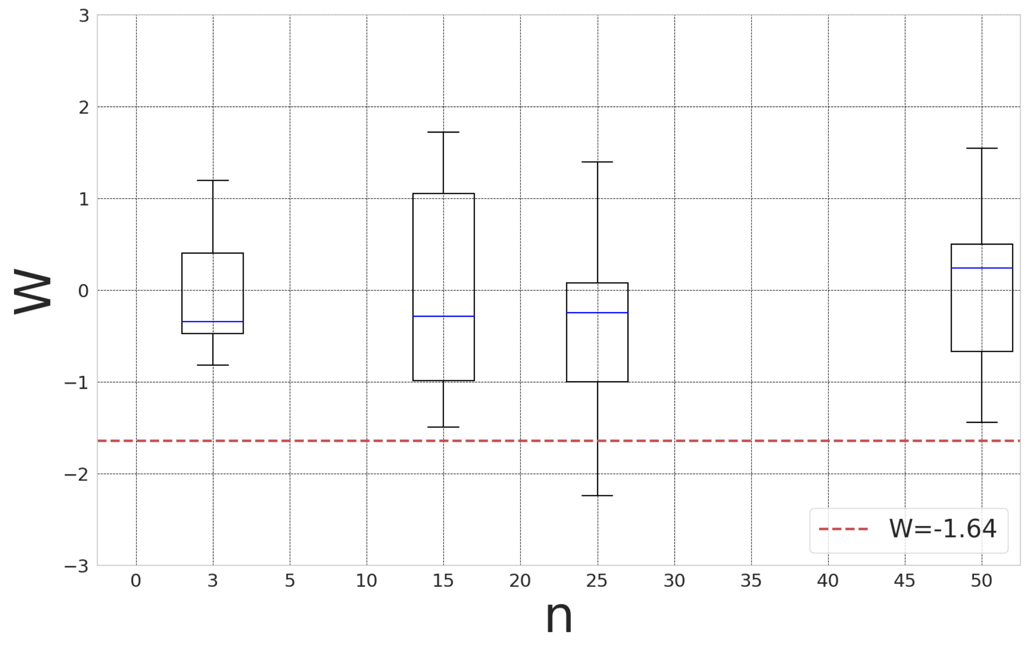
Figure 7 shows the results for the MST test on simplices and hypercubes. The red dashed line represents the threshold of $-1.64 =$ *W-value*, and each box represents the results for 10 simulations. We can observe that only one of the 80 simulations falls below the threshold (simplex $n = 25$), supporting the conclusion that MHAR mixes fast from any starting point.

7. Future Work

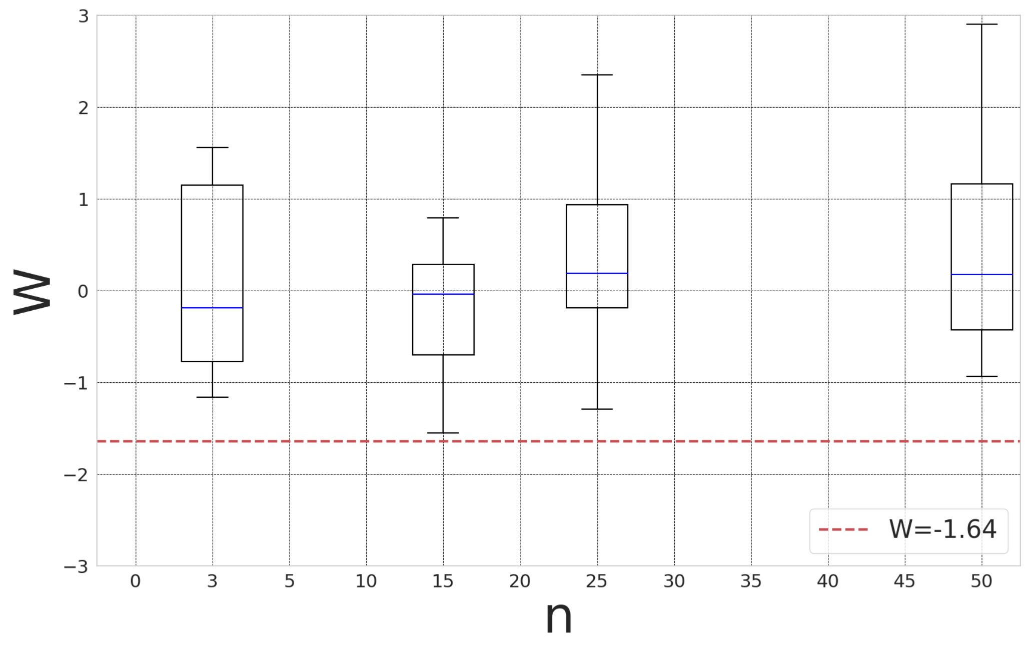
We theoretically analyzed the MHAR in detail and conducted experiments to assess its performance and statistical robustness. Yet there are many opportunities for further research. We proceed to enunciate some of them.

Floating-point arithmetic: Benchmarking the quality and speed of algorithms while varying their precision (8-bit, 16-bit, etc.) is known as Quantization (see Dubhir et al., 2021). All our experiments were conducted using 64-bit float precision numbers. We decided to do it like this to compute as accurately as possible the projection matrix P_{Δ^E}. Yet many applications may not need this level for precision and will benefit from faster running times by reducing the floating number precision, especially if the polytope in question is full-dimensional ($m_E = 0$) the inverse matrix will not be required.

Reducing the precision can improve the algorithm in two ways. First, the hardware (CPUs, GPUs, and TPUs) usually multiplies 16 or 32-bit numbers faster than their 64-



(a)



(b)

Figure 7: Friedman-Rafsky two-sample MST tests for (a) simplex and (b) hypercubes.

bit counterparts. Second, it allows the user to increase the expansion hyper-parameter z without saturating the memory since each matrix used by the algorithm will be "lighter" in terms of memory. This will allow the user to sample "bigger" polytopes with more restrictions or "greater" values of z using the same hardware. Reducing the precision has been successfully implemented in the machine learning community, where neural networks have been trained using 16 bits or even 8 bits of precision (Wang et al., 2018).

Matrix inversion precision: The numerical robustness of the sampler for non-full-dimensional polytopes was successfully tested in our experiments. Additionally, Montiel and Bickel (2013b) presented a fair-convergence test that shows that the chain produced by the MCMC from HAR accurately matches the first two moments of the theoretical distribution for non-full-dimensional polytopes. Yet more empirical tests would be of interest, especially for bigger dimensions and ill-posed matrices that may affect the projection matrix P_{Δ^E}.

Convergence: We tested the statistical performance of the MHAR using the framework established by Tervonen et al. (2013). However, additional statistical tests should be considered besides the MST test we performed, especially for larger values of n. Two interesting examples are the auto-correlation coefficient test and the Geweke test (see Matteucci and Veldkamp, 2013). This will clarify the convergence of the MHAR. We suspect that the results will be like the ones obtained in similar works that tested the HAR algorithm by Montiel and Bickel (2013a) and Tervonen et al. (2013) since there are no differences in the statistical properties between the MHAR and HAR.

Hardware Improvement: The speed and memory of commercially available GPUs are continually increasing. Testing the speed of the MHAR in multiple GPUs or even Tensor Processing Units (TPUs) is a great area of interest. For example, our tests were conducted on a single machine with a P100 GPU, yet cloud services (AWS, Google Cloud, Azure) allow the users to use up to 6 machines with 8 GPUs each for a total of 24 GPUs. Some of those GPUs like the A10G can perform floating-point operations of 2 to 200 times faster than the one we used, depending on the floating-point representation. Since MHAR uses PyTorch as the back-end, it is easy to take advantage of the distributing capabilities of this framework.

Optimizing z: We tested the MHAR with different z values, yet we did not try to find the optimal value of z given the problem at hand and the hardware architecture we were using. Intuitively, a "small" z that does not fill the GPU memory is under-using resources and a "big" z saturates the GPU memory and increases the communication between CPU and GPU, degrading the performance. Therefore, z should use all the GPU available memory without overflowing it. The library PyTorch-lighting currently offers a "Batch Size Finder" that can be used for this purpose [2].

8. Conclusions

MHAR showed sustainable performance improvements over HAR while having a robust uniform sampling. We hope that these technical advances move the scientific community towards simulation approaches to complement the already established analytical solutions. Our contribution was in creating the MHAR, analyzing its asymptotic behavior in terms of complexity and convergence, alongside a robust and easy-to-use implementation ready for deployment, including the cloud. Our implementation is substantially faster than existing libraries, especially for bigger dimensions. Additionally, we showed the versatility that Deep Learning frameworks, like PyTorch, can bring to support research. Moreover, the matrix operations of the MHAR implemented with PyTorch will provide further improvements to our implementation, creating additional benefits for practitioners that use our implementation.

We would like to emphasize the relevance of this work as a cornerstone to exploratory optimization algorithms. The speedups we presented in high dimensions make it possible for many new practical applications to become a normal trend, expanding the range of solutions that engineering can provide. In particular, our previous work in decision

[2]PyTorch-lighting Batch Size Finder `https://pytorch-lightning.readthedocs.io/en/stable/advanced/training_tricks.html`.

analysis, optimization, game theory, and ambiguity optimization will be significantly improved with this tool, and we think that many practitioners and researchers will be beneficial as well.

Our implementation could be extended to multiple GPUs, TPUs, and possibly distributed architectures. This will allow us to sample even larger polytopes using cloud architectures. Given the speed-up results, an approach for more general convex figures alongside accept-and-reject methods is worth exploring, especially for volume calculations.

Acknowledgments

Mario Vazquez Corte wants to acknowledge CONACYT and ITAM for the support provided in the completion of his academic work. Additionally, he also acknowledges Dr. Fernando Eponda, Dr. Jose Octavio Gutierrez, and Dr. Rodolfo Conde for their support and insight in the development of this work, and want to express his special gratitude to Saul Caballero, Daniel Guerrero, Alfredo Carrillo, Jesus Ledezma, and Erick Palacios Moreno for their invaluable feedback during this process. Dr. Montiel thanks, unique and exclusively, two anonymous reviewers whose comments help us to improve this manuscript.

References

Chay SC, Fardo RD, Mazumdar M (1975) On using the Box-Muller transformation with multiplicative congruential pseudo-random number generators. Journal of the Royal Statistical Society Series C (Applied Statistics) 24(1):132–135, https://doi.org/10.2307/2346711

Chen Y, Dwivedi R, Wainwright MJ, Yu B (2017) Vaidya walk: A sampling algorithm based on the volumetric barrier. In: 2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton), IEEE, pp 1220–1227, https://doi.org/10.1109/ALLERTON.2017.8262876

Chen Y, Dwivedi R, Wainwright MJ, Yu B (2018) Fast MCMC sampling algorithms on polytopes. Journal of Machine Learning Research 19(55):1–86, http://jmlr.org/papers/v19/18-158.html

Chrzeszczyk A, Chrzeszczyk J (2013) Matrix computations on the GPU: Cublas and magma by example. Accessed: 28 May 2023.

Cid GM, Montiel LV (2019) Negociaciones de máxima probabilidad para juegos cooperativos con fines comerciales. Revista mexicana de economía y finanzas 14(2):245–259, https://doi.org/10.21919/remef.v14i2.382

Cormen TH, Leiserson C, Rivest R, Stein C (2009) Introduction to Algorithms, 3rd edn, MIT Press, Cambridge, pp 827–831

Dubhir T, Mishra M, Singhal R (2021) Benchmarking of quantization libraries in popular frameworks. In: 2021 IEEE International Conference on Big Data (Big Data), IEEE, pp 3050–3055, https://doi.org/10.1109/BigData52589.2021.9671500

Emiris IZ, Fisikopoulos V (2014) Efficient random-walk methods for approximating polytope volume. In: Proceedings of the Thirtieth Annual Symposium on Computational Geometry, Association for Computing Machinery, New York, NY, USA, pp 318–327, https://doi.org/10.1145/2582112.2582133

Feldman J, Wainwright MJ, Karger DR (2005) Using linear programming to decode binary linear codes. IEEE Transactions on Information Theory 51(3):954–972, https://doi.org/10.1109/TIT.2004.842696

Friedman JH, Rafsky LC (1979) Multivariate generalizations of the Wald-Wolfowitz and Smirnov two-sample tests. The Annals of Statistics 7(4):697–717, https://doi.org/10.1214/aos/1176344722

Geyer, Charles J (1992) Practical Markov chain Monte Carlo. Statistical Science 7(4):473–483, https://doi.org/10.1214/ss/1177011137

Gustafson A, Narayanan H (2022) John's Walk. Advances in Applied Probability p 1–19, https://doi.org/10.1017/apr.2022.34

Huang J, Yu CD, van de Geijn RA (1993) Implementing Strassen's algorithm with CUTLASS on NVIDIA Volta GPUs. Tech. rep., The University of Texas at Austin, https://doi.org/10.48550/arXiv.1808.07984

Huang KL, Mehrotra S (2015) An empirical evaluation of a walk-relax-round heuristic for mixed integer convex programs. Computational Optimization and Applications 60(3):559–585, https://doi.org/10.1007/s10589-014-9693-5

Kannan R, Narayanan H (2013) Random walks on polytopes and an affine interior point method for linear programming. Mathematics of Operations Research 37(1):1–20, https://doi.org/10.1287/moor.1110.0519

Kapfer SC, Krauth W (2013) Sampling from a polytope and hard-disk Monte Carlo. Journal of Physics: Conference Series 454(1):012031, https://doi.org/10.1088/1742-6596/454/1/012031

Kimm H, Paik I, Kimm H (2021) Performance comparision of tpu, gpu, cpu on google colaboratory over distributed deep learning. In: 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), IEEE, pp 312–319, https://doi.org/10.1109/MCSoC51149.2021.00053

Knight PA (1995) Fast rectangular matrix multiplication and QR decomposition. Linear Algebra and Its Applications 221:69–81, https://doi.org/10.1016/0024-3795(93)00230-W

Lai PW, Arafat H, Elango V, Sadayappan P (2013) Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs. In: 20th Annual International Conference on High Performance Computing, pp 139–148, https://doi.org/10.1109/HiPC.2013.6799109

Lawrence J (1991) Polytope volume computation. Mathematics of Computation 57(195):259–271, https://doi.org/10.1090/S0025-5718-1991-1079024-2

Le Gall F (2014) Powers of tensors and fast matrix multiplication. In: Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation, Association for Computing Machinery, New York, NY, USA, ISSAC '14, p 296–303, https://doi.org/10.1145/2608628.2608664

Lee YT, Vempala SS (2018) Convergence rate of Riemannian Hamiltonian Monte Carlo and faster polytope volume computation. In: Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, Association for Computing Machinery, New York, NY, USA, STOC 2018, pp 1115–1121, https://doi.org/10.1145/3188745.3188774

Lee YT, Vempala SS (2022) Geodesic walks in polytopes. SIAM Journal on Computing 51(2):400–488, https://doi.org/10.1137/17M1145999

Li J, Ranka S, Sahni S (2011) Strassen's matrix multiplication on gpus. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems, pp 157–164, https://doi.org/10.1109/ICPADS.2011.130

Lovász L (1999) Hit-and-Run mixes fast. Mathematical Programming 86(3):443–461, https://doi.org/10.1007/s101070050099

Lovász L, Simonovits M (1993) Random walks in a convex body and an improved volume algorithm. Random Structures and Algorithms 4(4):359–412, https://doi.org/10.1002/rsa.3240040402

Ma YA, Chen Y, Jin C, Flammarion N, Jordan MI (2019) Sampling can be faster than optimization. Proceedings of the National Academy of Sciences 116(42):20881–20885, https://doi.org/10.1073/pnas.1820003116

Marsaglia G (1972) Choosing a point from the surface of a sphere. The Annals of Mathematical Statistics 43(2):645–646, https://doi.org/10.1214/aoms/1177692644

Matteucci M, Veldkamp BP (2013) On the use of MCMC computerized adaptive testing with empirical prior information to improve efficiency. Statistical Methods & Applications 22(2):243–267, https://doi.org/10.1007/s10260-012-0216-1

Mittal S, Vaishay S (2019) A survey of techniques for optimizing deep learning on GPUs. Journal of Systems Architecture 99:101635, https://doi.org/10.1016/j.sysarc.2019.101635

Montiel LV, Bickel EJ (2012) A simulation-based approach to decision making with partial information. Decision Analysis 9(4):329–347, https://doi.org/10.1287/deca.1120.0252

Montiel LV, Bickel EJ (2013a) Approximating joint probability distributions given partial information. Decision Analysis 10(1):26–41, https://doi.org/10.1287/deca.1120.0261

Montiel LV, Bickel EJ (2013b) Generating a random collection of discrete joint probability distributions subject to partial information. Methodology and Computing in Applied Probability 15(4):951–967, https://doi.org/10.1007/s11009-012-9292-9

Montiel LV, Bickel EJ (2014) A generalized sampling approach for multilinear utility functions given partial preference information. Decision Analysis 11(3):147–170, https://doi.org/10.1287/deca.2014.0296

Nikolić GS, Dimitrijević BR, Nikolić TR, Stojcev MK (2022) A survey of three types of processing units: Cpu, gpu and tpu. In: 2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST), IEEE, pp 1–6, https://doi.org/10.1109/ICEST55168.2022.9828625

Paszke A, et al. (2019) Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32, Curran Associates, Inc., Vancouver, pp 8026–8037, https://doi.org/10.48550/arXiv.1912.01703

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (2007) Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3rd edn. Cambridge University Press, NY

Rubin DB (1981) The Bayesian bootstrap. The Annals of Statistics 6(1):130–134, https://doi.org/10.1214/aos/1176345338

Sharkawi SS, Chochia GA (2020) Communication protocol optimization for enhanced gpu performance. IBM Journal of Research and Development 64(3/4):9–1, https://doi.org/10.1147/JRD.2020.2967311

Smith RL (1984) Efficient Monte Carlo procedures for generating points uniformly distributed over bounded regions. Operations Research 32(6):1296–1308, https://doi.org/10.1287/opre.32.6.1296

Smith RL (1996) The Hit-and-Run sampler: a globally reaching Markov chain sampler for generating arbitrary multivariate distributions. In: Proceedings Winter Simulation Conference, pp 260–264, https://doi.org/10.1145/256562.256619

Tervonen T, Valkenhoef v G, Basturk N, Postmus D (2013) Hit-and-Run enables efficient weight generation for simulation-based multiple criteria decision analysis. European Journal of Operational Research 224(3):168–184, https://doi.org/10.1016/j.ejor.2012.08.026

Vempala S, Bertsimas D (2004) Solving convex programs by Random Walks. Journal of the ACM 51(4):540–556, https://doi.org/10.1145/1008731.1008733

Wang N, Choi J, Brand D, Chen CY, Gopalakrishnan K (2018) Training deep neural networks with 8-bit floating point numbers. In: Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., Vancouver, vol 31, pp 1–10

Appendix A. Mathematical Proofs of Lemmas and Theorems

Lemma Appendix A.1. *If $m_E < n$, then the complexity of calculating P_{Δ^E} is $\mathcal{O}(m_E^{\omega-2}n^2)$.*

Proof. Computing P_{Δ^E} is done in three matrix multiplications, one matrix-to-matrix subtraction, and one matrix inversion operation over $(A^E A'^E)$. The number of operations needed to calculate the inverse matrix depends on the algorithm used for matrix multiplication Cormen et al. (2009). The order of number of operations for computing P_{Δ^E} is the sum of the following:

1. Obtain $(A^E A'^E)$ in $\mathcal{O}(\mu_{A^E,A'^E}) = \mathcal{O}(\mu(m_E, n, m_E)) = \mathcal{O}(m_E^{\omega-1}n)$ operations.
2. Find the inverse $(A^E A'^E)^{-1}$ in $\mathcal{O}(m_E^{\omega})$, since $(A^E A'^E)^{-1}$ has dimension $m_E \times m_E$.
3. Multiply $A'^E(A^E A'^E)^{-1}$ in $\mathcal{O}(\mu_{A'^E,(A^E A'^E)^{-1}}) = \mathcal{O}(\mu(n, m_E, m_E)) = \mathcal{O}(m_E^{\omega-1}n)$.
4. Calculate $A'^E(A^E A'^E)^{-1}A^E$ in $\mathcal{O}(\mu_{A'^E(A^E A'^E)^{-1},A^E}) = \mathcal{O}(\mu(n, m_E, n)) = \mathcal{O}(m_E^{\omega-2}n^2)$.
5. Subtract $I - A'^E(A^E A'^E)^{-1}A^E$ in $\mathcal{O}(n^2)$.

These sum to $2 \times \mathcal{O}(m_E^{\omega-1}n) + \mathcal{O}(m_E^{\omega}) + \mathcal{O}(m_E^{\omega-2}n^2) + \mathcal{O}(n^2)$. Hence the complexity of calculating P_{Δ^E} is $\mathcal{O}(\mu_{A'^E(A^E A'^E)^{-1},A^E}) = \mathcal{O}(m_E^{\omega-2}n^2)$. □

Lemma Appendix A.2. *The cost per iteration of HAR for $0 \leq m_E$ is $\mathcal{O}(\max\{m_I n, m_E^{\omega-2}n^2\})$.*

Proof. As seen in Algorithm 1, the only difference between the full and non-full-dimensional cases is the projection step $P_{\Delta^E}h = d$. Then, the cost per iteration is defined by the larger of the original cost per iteration $\mathcal{O}(m_I n)$ of HAR for $m_E = 0$, and the extra cost induced by the projection when $m_E > 0$.

Because P_{Δ^E} has dimension $n \times n$ and h is an $n \times 1$ vector, $\mu_{P_{\Delta^E},h} = n^2$ and the complexity is $\mathcal{O}(n^2)$. By Lemma 3.1, finding P_{Δ^E} has an asymptotic complexity of $\mathcal{O}(m_E^{\omega-2}n^2)$. Therefore, the cost of projecting h at each iteration is $\mathcal{O}(n^2) + \mathcal{O}(m_E^{\omega-2}n^2) = \mathcal{O}(m_E^{\omega-2}n^2)$, since $m_E > 0$. Therefore, the cost per iteration for $m_E > 0$ is $\mathcal{O}(\max\{m_I n, m_E^{\omega-2}n^2)\})$. If $m_E = 0$, then the coefficient $\max\{m_I n, m_E^{\omega-2}n^2)\}$ equals $\max\{m_I n, 0\} = m_I n$ and the cost per sample is $\mathcal{O}^*(\max\{m_I n, 0)\}) = \mathcal{O}^*(m_I n)$. □

Lemma Appendix A.3. *The complexity of generating matrix D in MHAR given P_{Δ^E} and $\max\{m_I, n\} \leq z$ is $\mathcal{O}(nz)$ if $m_E = 0$, and $\mathcal{O}(n^{\omega-1}z)$ if $m_E > 0$.*

Proof. Generating H has complexity $\mathcal{O}(nz)$ using the Box-Muller method. If $m_E = 0$, then $D = H$, implying a total asymptotic cost $\mathcal{O}(nz)$. If $m_E > 0$, then $D = P_{\Delta^E}H$, whose cost $\mathcal{O}(\mu_{P_{\Delta^E},H}) = \mathcal{O}(n^{\omega-1}z)$ given by $\max\{m_I, n\} \leq z$, needs to be included. $\mathcal{O}(n^{\omega-1}z)$ bounds $\mathcal{O}(nz)$. Therefore, the total cost of computing D for $m_E > 0$ is bounded by $\mathcal{O}(n^{\omega-1}z)$. $\square$

Lemma Appendix A.4. *The complexity of generating all line sets $\{L^k\}_{k=1}^{z}$ in MHAR given D, X, and $\max\{m_I, n\} \leq z$ is bounded by $\mathcal{O}(m_I n^{\omega-2}z)$ if $n \leq m_I$, and by $\mathcal{O}(m_I^{\omega-2}nz)$ otherwise.*

Proof. All Λ^ks can be obtained as follows:

1. Obtain matrix $A^I X$ in $\mathcal{O}(\mu_{A^I,X})$. This is done in $\mathcal{O}(m_I n^{\omega-2}z)$ if $n \leq m_I$, and in $\mathcal{O}(m_I^{\omega-2}nz)$ otherwise.

2. Compute $B^I - A^I X$, where $B_I = (b^I|...|b^I) \in \mathbb{R}^{m^I \times z}$, which takes $\mathcal{O}(m_I z)$ operations.

3. Calculate $A^I D$, which is bounded by $\mathcal{O}(\mu_{A^I,D})$, which is done in $\mathcal{O}(m_I n^{\omega-2}z)$ if $n \leq m_I$, and in $\mathcal{O}(m_I^{\omega-2}nz)$ otherwise.

4. Divide $\frac{B^I - A^I X}{A^I D}$ (entry-wise) to obtain all λ_i^k. All the necessary point-wise operations for this calculation have a combined order of $\mathcal{O}(m_I z)$.

5. For each $k \in \{1, ..., z\}$, find which coefficients $a_i^I d^k$ are positive or negative, which takes $\mathcal{O}(m_I z)$.

6. For each $k \in \{1, ..., z\}$, find the intervals $\lambda_{min}^k = \max\ \{\lambda_i^k \mid a_i^I d^k < 0\}$ and $\lambda_{max}^k = \min\ \{\lambda_i^k \mid a_i^I d^k > 0\}$, which can be done in $\mathcal{O}(m_I z)$.

This procedure constructs all the intervals $\Lambda^k = (\lambda_{\min}^k, \lambda_{\max}^k)$. The complexity of this operation is bounded by $\mathcal{O}(\mu_{A^I,X}) = \mathcal{O}(\mu_{A^I,D})$. Hence, the complexity of finding all line sets is bounded by $\mathcal{O}(m_I n^{\omega-2}z)$ if $n \leq m_I$, and by $\mathcal{O}(m_I^{\omega-2}nz)$ otherwise. $\square$

Lemma Appendix A.5. *Sampling z new points given $\{\Lambda^k\}_{k=1}^z$ has complexity $\mathcal{O}(zn)$.*

Proof. Selecting a random $\theta^k \in \Lambda^k$ takes $\mathcal{O}(1)$. Sampling a new point $x_{t,j+1}^k = x_{t,j}^k + \theta d_{t,j}^k$ has complexity $\mathcal{O}(n)$ because it requires n scalar multiplications and n sums. Then, sampling all new $x_{t,j+1}^k$ points is bounded by $\mathcal{O}(zn)$. □

Lemma Appendix A.6. *Assume $m_E = 0$, $\max\{n, m\} < z$, and $n \leq m_I$. Then, the cost per iteration of MHAR is $\mathcal{O}(m_I n^{\omega-2} z)$, which is the number of operations needed for finding all line sets $\{L^k\}_{k=1}^z$.*

Proof. First, we enumerate the cost of each step of the iteration for $m_E = 0$ and $n \leq m_I$ if $\max\{n, m\} < z$:

1. By Lemma 3.1, generating P_{Δ^E} is bounded by $\mathcal{O}(1)$.
2. By Lemma 4.1, generating D is bounded by $\mathcal{O}(nz)$.
3. By Lemma 4.2, generating $\{L^k\}_{k=1}^z$ for $n \leq m_I$ is bounded by $\mathcal{O}(m_I n^{\omega-2} z)$.
4. By Lemma 4.3, generating all new $x_{t,j+1}^k$ is bounded by $\mathcal{O}(zn)$.

By hypothesis, $0 < n \leq m_I$. Then, $nz \leq m_I z < m_I n^{\omega-2} z$, because $\omega \in (2, 3]$. Therefore, $\mathcal{O}(1) \subseteq \mathcal{O}(nz) \subseteq \mathcal{O}(m_I n^{\omega-2} z)$, where the first term is the complexity of finding the projection matrix (omitted for $m_E = 0$), the second one bounds generating D and sampling new points, and the third one is the asymptotic cost of finding all line sets $\{L^k\}_{k=1}^z$. □

Lemma Appendix A.7. *Assume $m_E = 0$, $\max\{n, m\} < z$, and $n > m_I$. Then, the cost per iteration of MHAR is $\mathcal{O}(n m_I^{\omega-2} z)$, which is the number of operations needed for finding all line sets $\{L^k\}_{k=1}^z$.*

Proof. As in the proof of Lemma 4.4, the complexity of the projection matrix, generating D, and sampling all-new $x_{t,j+1}^k$ points is the same, given by $m_E = 0$ and $n > m_I$. Hence, the only change is provided by Lemma 4.2, in which the cost of finding all line sets $\{L^k\}_{k=1}^z$ for $n > m_I$ is $\mathcal{O}(n m_I^{\omega-2} z)$. By hypothesis, $0 < m_I$ and $\max\{n, m\} < z$, thus $nz < n m_I^{\omega-2} z$. Therefore, $\mathcal{O}(1) \subseteq \mathcal{O}(nz) \subseteq \mathcal{O}(n m_I^{\omega-2} z)$, where the third term is the cost of finding all line sets $\{L^k\}_{k=1}^z$. □

Lemma Appendix A.8. *Assume $m_E < n$ and $(m, n) < z$. Then, the cost of calculating the projection matrix P_{Δ^E} is bounded by the cost of generating D.*

Proof. By hypothesis $m_E < n$, implying that $m_E^{\omega-2}n^2 < n^{\omega-2}n^2 = n^{\omega}$. Because $n < z$, $n^{\omega} = n^{\omega-1}n < n^{\omega-1}z$. Combining both inequalities yields $m_E^{\omega-2}n^2 < n^{\omega} < n^{\omega-1}z$. Therefore, $\mathcal{O}(m_E^{\omega-2}n^2) \subseteq \mathcal{O}(n^{\omega-1}z)$, where the first term is the complexity of computing P_{Δ^E} (by Lemma 3.1), and the second term is the complexity of projecting H in order to obtain D (by Lemma 4.1). $\square$

Lemma Appendix A.9. *Assume $m_E > 0$, $\max\{n, m\} < z$, and $n \leq m_I$. Then, the cost per iteration of MHAR is $\mathcal{O}(m_I n^{\omega-2}z)$, which is the number of operations needed for finding all line sets $\{L^k\}_{k=1}^{z}$.*

Proof. First, we enumerate the cost of each step of the iteration for $m_E > 0$, $n \leq m_I$, and $\max\{n, m\} < z$:

1. By Lemma 3.1, generating P_{Δ^E} is bounded by $\mathcal{O}(m_E^{\omega-2}n^2)$.
2. By Lemma 4.1, generating D is bounded by $\mathcal{O}(n^{\omega-1}z)$.
3. By Lemma 4.2, generating $\{L^k\}_{k=1}^{z}$ for $n \leq m_I$ is bounded by $\mathcal{O}(m_I n^{\omega-2}z)$.
4. By Lemma 4.3, generating all new $x_{t,j+1}^{k}$ is bounded by $\mathcal{O}(zn)$.

Using Lemma 4.7, the Big-O term for finding P_{Δ^E} (step 1) is bounded by the term of generating D (step 2). Because $n < m_I$, $n^{\omega-1}z = n^{\omega-2}nz < n^{\omega-2}m_I z$. Therefore, $\mathcal{O}(m_E^{\omega-2}n^2) \subseteq \mathcal{O}(n^{\omega-1}z) \subseteq \mathcal{O}(m_I n^{\omega-2}z)$, which are the respective costs of steps 1, 2, and 3. Furthermore, $nz \leq n^{\omega-2}m_I z$, implying that step 4 is also bounded by step 3 in terms of complexity. This implies that all the operations above are bounded by the term $\mathcal{O}(m_I n^{\omega-2}z)$, which is the asymptotic complexity of finding all line sets $\{L^k\}_{k=1}^{z}$. $\square$

Lemma Appendix A.10. *Assume* $m_E > 0$, $\max\{n, m\} < z$, *and* $n > m_I$. *Then, the cost per iteration of MHAR is* $\mathcal{O}(nm_I^{\omega-2}z)$, *which is the number of operations needed for generating* D.

Proof. As in the proof of Lemma 4.8, the cost of the projection matrix, generating D, and sampling all-new $x^k_{t,j+1}$ points is the same, given by $m_E > 0$ and $n > m_I$. Hence, the only change is provided by Lemma 4.2, in which the cost of finding all line sets $\{L^k\}_{k=1}^z$ for $n > m_I$ is $\mathcal{O}(nm_I^{\omega-2}z)$.

By Lemma 4.7, the Big-O term for finding P_{Δ^E} is bounded by the term of generating D. Because $n > m_I$, $m_I^{\omega-2}nz < n^{\omega-2}nz = n^{\omega-1}z$. Therefore, $\mathcal{O}(m_E^{\omega-2}n^2) \subseteq \mathcal{O}(nm_I^{\omega-2}z) \subseteq \mathcal{O}(n^{\omega-1}z)$, which are the respective costs of the projection matrix, finding all line sets, and generating D. Furthermore, $nz \leq n^{\omega-2}nz = n^{\omega-1}z$, implying that the cost of sampling all new $x^k_{t,j+1}$ is also bounded by the cost of generating D. This implies that all the operations above are bounded by $\mathcal{O}(nm_I^{\omega-2}z)$. $\square$

Lemma Appendix A.11. *For* $\max\{n, m\} < zm$, *and* $n < m$, *MHAR has a lower cost per sample than does John's walk after proper pre-processing, warm start, and ignoring the logarithmic and error terms.*

Proof. Given proper pre-processing, $n \ll m$, and $\max\{n, m\} < z$, then MHAR's cost per sample is $\mathcal{O}^*(mn^{\omega+1})$, and that for John's walk is $\mathcal{O}(mn^{11} + n^{15})$. Note that $mn^{\omega+1} \in \mathcal{O}(mn^{11} + n^{15})$. Therefore, when ignoring the logarithmic and error terms, MHAR has a lower cost per sample. $\square$

Lemma Appendix A.12. *For* $\max\{n, m\} < z$ *and the regime* $n \ll m$, *MHAR has a lower cost per sample than does the John walk after proper pre-processing, warm start, and ignoring logarithmic and error terms.*

Proof. From proper pre-processing, $n \ll m$, and $\max\{n, m\} < z$, MHAR's cost per sample is $\mathcal{O}^*(mn^{\omega+1})$ and that for John walk is $\mathcal{O}(mn^{\omega+\frac{3}{2}})$. Note that $mn^{\omega+1} \in \mathcal{O}(mn^{\omega+\frac{3}{2}})$. Therefore when ignoring the logarithmic and error terms, MHAR has a lower cost per sample. $\square$

Appendix B. Additional Optimal expansion Experiments

Here we present the results for different expansion parameters using 10 MHAR runs for each dimension (25, 50, 100, 500) on simplices and hypercubes. Figures B.8 to B.11 shows the box-plots for simplices while Figures B.12 to B.15 shows the box-plots for hypercubes.

The box in the boxplots shows the 25%, 50%, and 75% percentiles. The dots mark the outliers, and the upper and lower limits mark the maximum and minimum values without considering outliers.

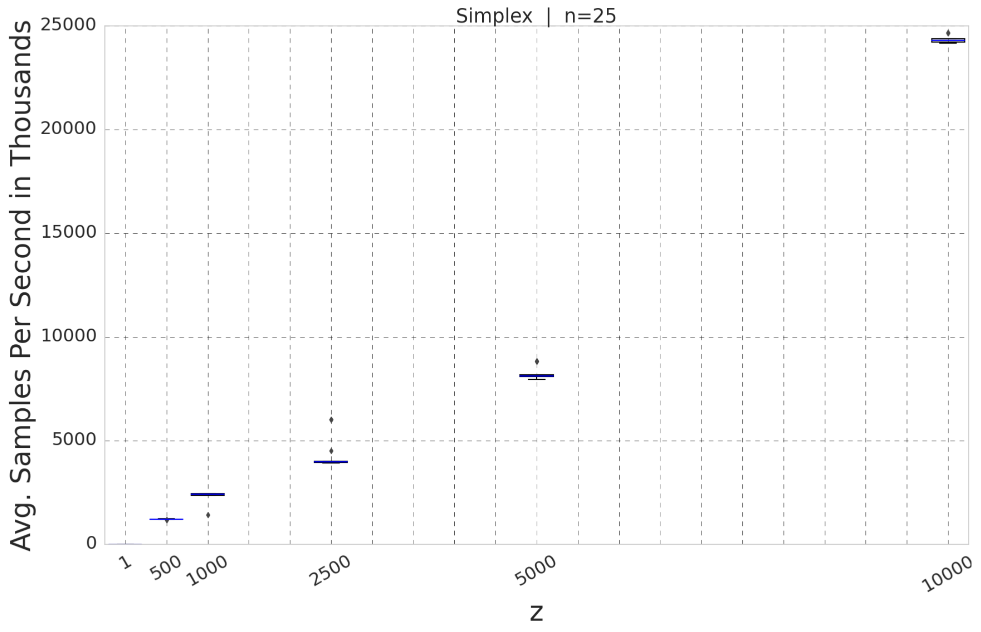


Figure B.8: Box-plots for simplices in dimension 25 comparing expansion behavior for different values of the expansion hyper-parameter z.

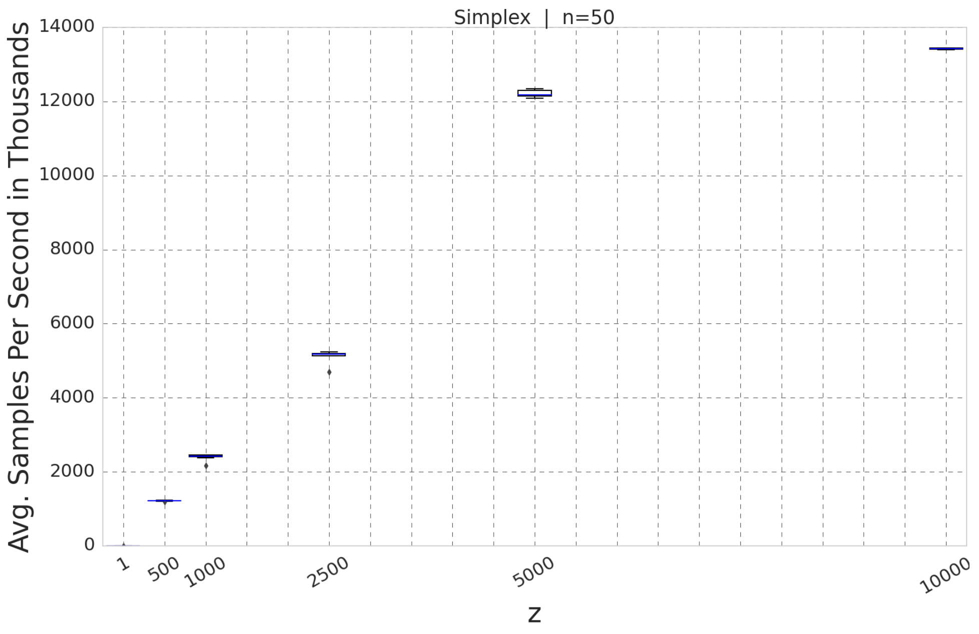


Figure B.9: Box-plots for simplices in dimension 50 comparing expansion behavior for different values of the expansion hyper-parameter z.

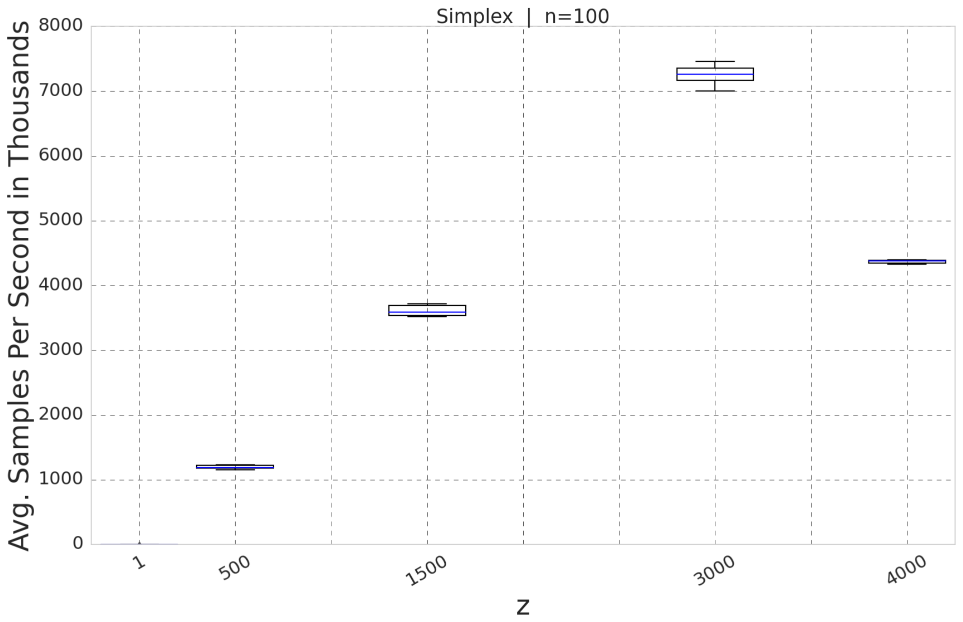


Figure B.10: Box-plots for simplices in dimension 100 comparing expansion behavior for different values of the expansion hyper-parameter z.

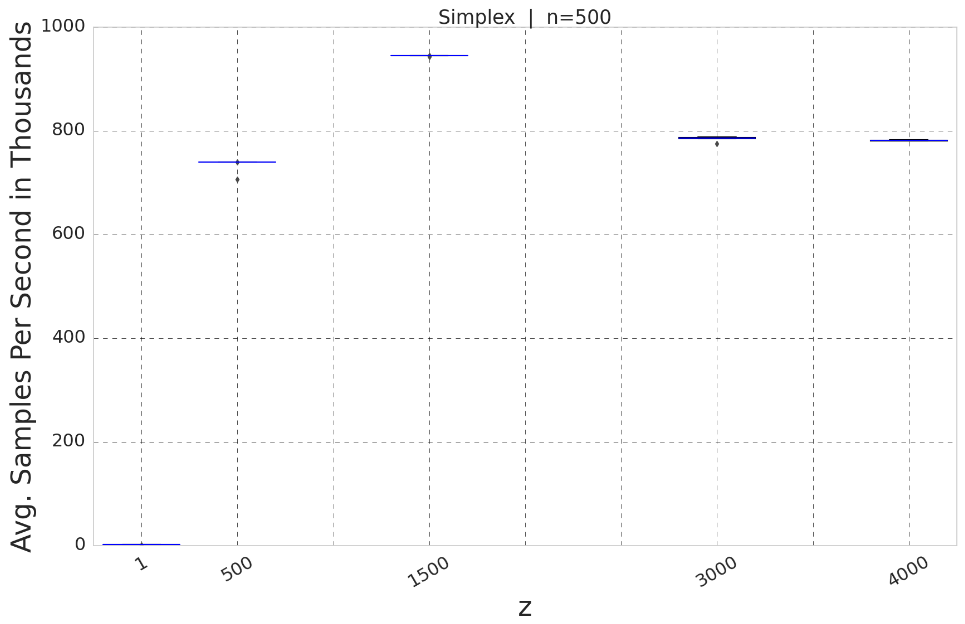


Figure B.11: Box-plots for simplices in dimension 500 comparing expansion behavior for different values of the expansion hyper-parameter z.

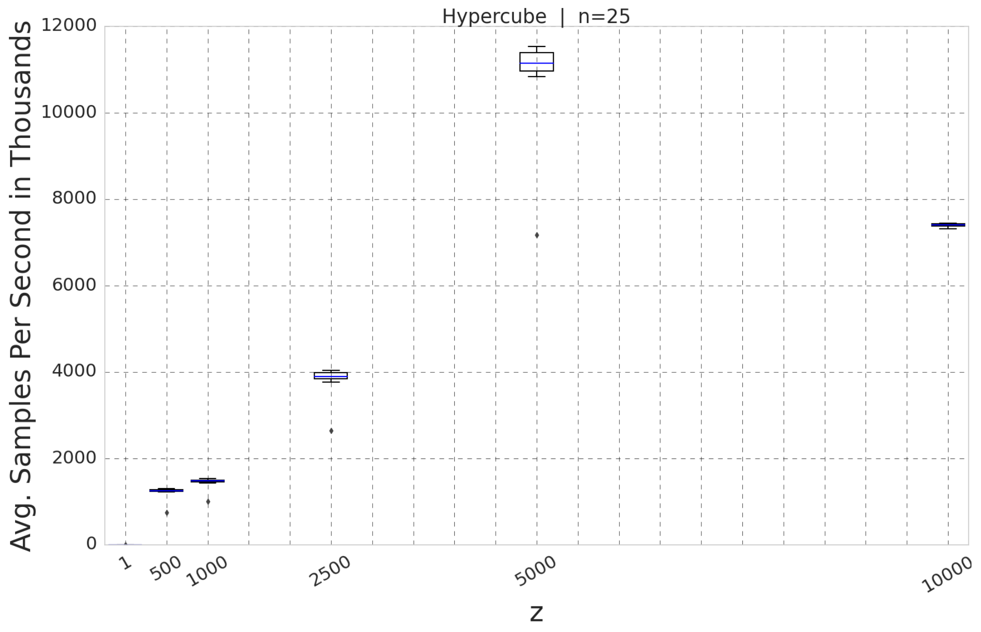


Figure B.12: Box-plots for hypercube in dimension 25 comparing expansion behavior for different values of the expansion hyper-parameter z.

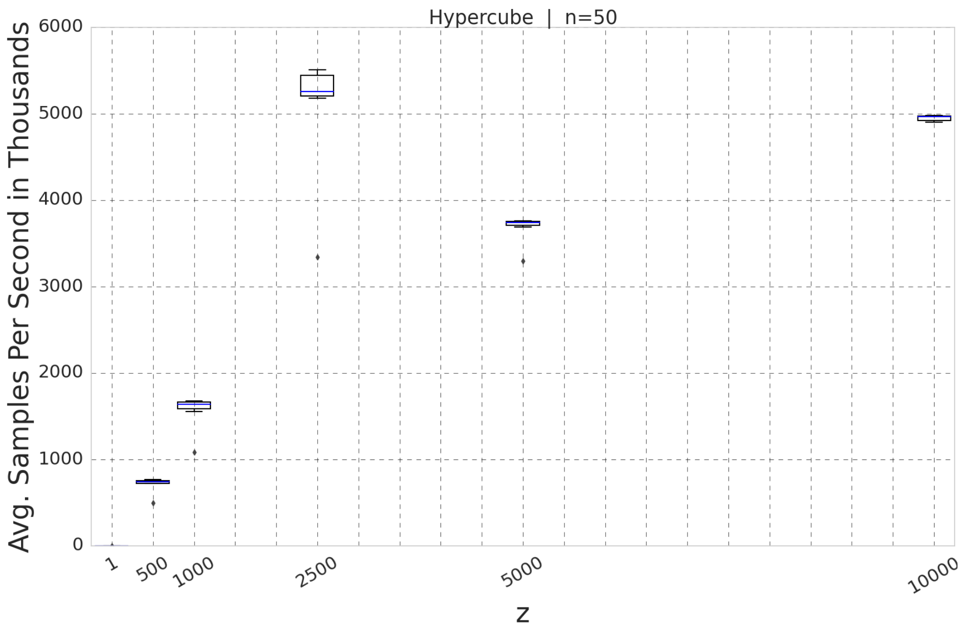


Figure B.13: Box-plots for hypercube in dimension 50 comparing expansion behavior for different values of the expansion hyper-parameter z.

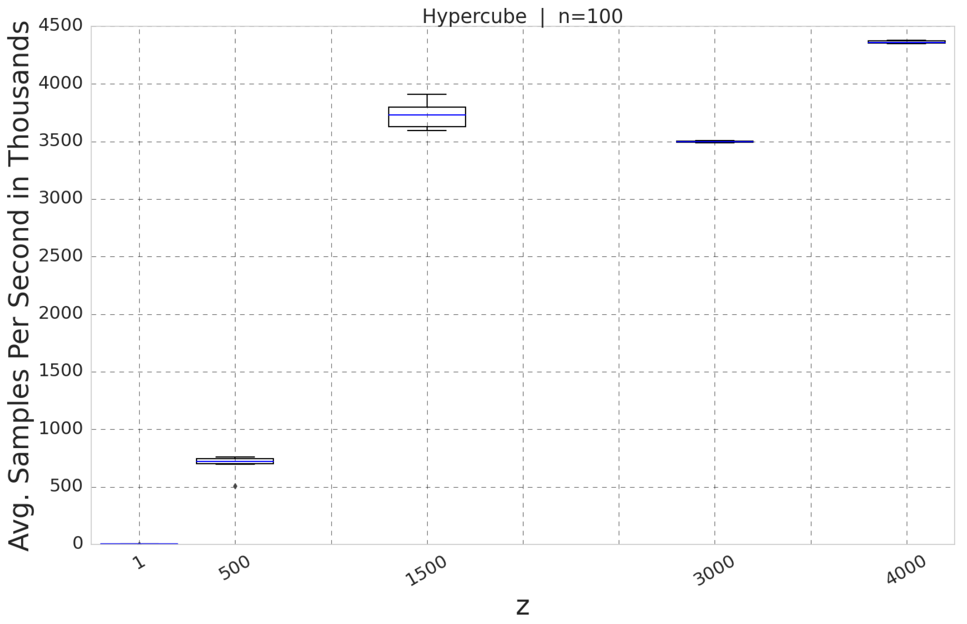


Figure B.14: Box-plots for hypercube in dimension 100 comparing expansion behavior for different values of the expansion hyper-parameter z.

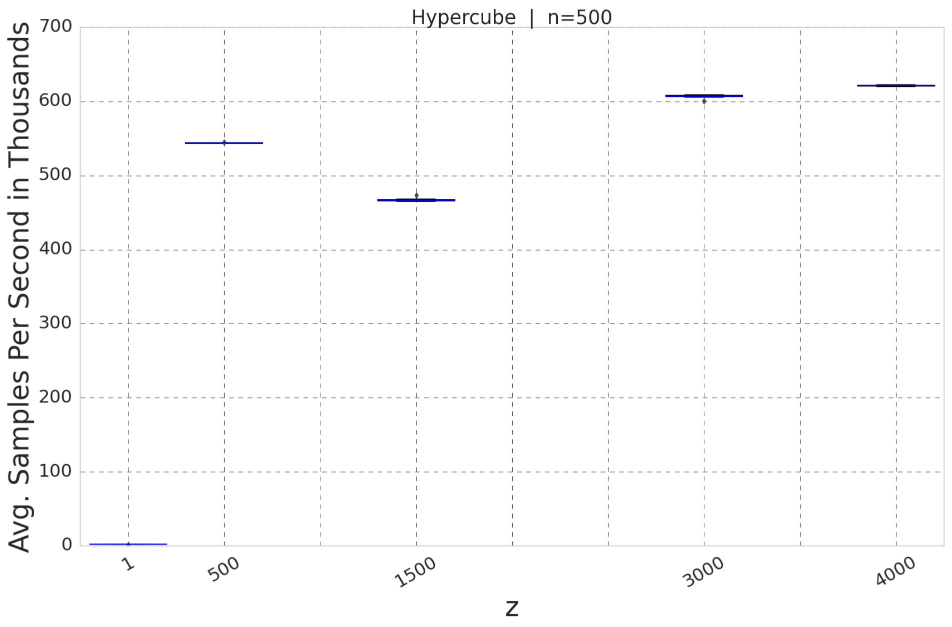


Figure B.15: Box-plots for hypercube in dimension 500 comparing expansion behavior for different values of the expansion hyper-parameter z.

Appendix C. Additional Performance Experiments

Here we present additional experiments on the fitness of MHAR. Table C.5 reports the running times and the average sampled points per second for the best values of z for each combination of Figure and dimension. For each combination, we conducted the experiment 10 times. Table C.5 shows that average samples per second are lower for higher dimensions, due to the curse of dimensionality. However, the performance of MHAR is outstanding.

Table C.5: Samples Per Second of the MHAR.

				Avg. Samples Per Sec.		Running Time (sec)	
Figure	n	z	Tot. Samples	Mean	Std. Dev.	Mean	Std. Dev.
Hypercube	3	10,000	300,000,000	25,357,074	675,444	12	0.3
Hypercube	5	10,000	300,000,000	13,206,090	376,069	23	0.7
Hypercube	15	10,000	300,000,000	25,344,795	655,021	12	0.3
Hypercube	25	5,000	150,000,000	10,839,474	1,236,620	14	2.3
Hypercube	50	2,500	75,000,000	5,151,517	612,242	15	2.5
Hypercube	100	4,000	120,000,000	4,363,526	10,620	28	0.1
Hypercube	250	3,000	90,000,000	1,219,420	8,630	74	0.5
Hypercube	500	4,000	120,000,000	621,554	783	193	0.2
Hypercube	1,000	4,000	120,000,000	248,514	183	483	0.4
Hypercube	2,500	1,500	15,000,000	50,809	15	295	0.1
Hypercube	5,000	1,000	10,000,000	16,162	6	619	0.2
Simplex	3	10,000	300,000,000	19,795,014	2,628,558	15	1.8
Simplex	5	10,000	300,000,000	22,878,783	1,258,482	13	0.8
Simplex	15	10,000	300,000,000	24,269,548	302,854	12	0.2
Simplex	25	10,000	300,000,000	24,338,761	168,301	12	0.1
Simplex	50	10,000	300,000,000	13,425,901	16,404	22	0.1
Simplex	100	3,000	90,000,000	7,255,837	135,617	12	0.2
Simplex	250	4,000	120,000,000	2,656,449	4,441	45	0.1
Simplex	500	1,500	45,000,000	944,785	583	48	0.1
Simplex	1,000	500	15,000,000	329,315	557	46	0.1
Simplex	2,500	500	5,000,000	77,312	3,046	65	2.9
Simplex	5,000	1,000	10,000,000	22,438	62	446	1.3

Note: The table contains the performance statistics obtained during the MHAR experiments for the best possible value of z we could find.

Appendix D. Declarations

Appendix D.1. Funding

The research was conducted without external funding.

Appendix D.2. Conflicts of interest/Competing interests

The authors state that there are no conflicts or competing interests.

Appendix D.3. Availability of data and material

The code for replicating the experiments is available on github. The source code is available in the github repository, and can be replicated in the free online colab platform: https://github.com/uumami/mhar_pytorch

The authors created a library for testing at https://github.com/uumami/mhar

Appendix D.3.1. Code availability

Code: https://github.com/uumami/mhar_pytorch

Python library: https://pypi.org/project/mhar/

Library Code: https://github.com/uumami/mhar

Appendix D.4. Ethics approval

We do not perform any actions or experiments that require ethics approval.

Appendix D.5. Consent to participate

No people were involved in experiments that required consent from subjects.

Appendix D.6. Consent for publication

Both authors express consent to publish this article.