volesti: Volume Approximation and Sampling for Convex Polytopes in R

Apostolos Chalkis
National & Kapodistrian University of Athens, Greece

Vissarion Fisikopoulos
National & Kapodistrian University of Athens, Greece

Abstract

Sampling from high dimensional distributions and volume approximation of convex bodies are fundamental operations that appear in optimization, finance, engineering and machine learning. In this paper we present **volesti**, a C++ package with an R interface that provides efficient, scalable algorithms for volume estimation, uniform and Gaussian sampling from convex polytopes. **volesti** scales to hundreds of dimensions, handles efficiently three different types of polyhedra and provides non existing sampling routines to R. We demonstrate the power of **volesti** by solving several challenging problems using the R language.

Keywords: volume approximation, sampling, polytopes, integration, financial crisis detection, zonotope approximation, counting linear extensions, R, C++.

1. Introduction

High-dimensional sampling and integration are fundamental problems with many applications in science and engineering (Iyengar 1988; Somerville 1998; Genz and Bretz 2009; Schellenberger and Palsson 2009; Venzke, Molzahn, and Chatzivasileiadis 2019). Nevertheless, those problems are computationally hard for general dimension. For example computing the volume of convex polytopes (a special case of integration) is a hard problem (Dyer and Frieze 1988). Even worse, deterministic approximation of volume computation is only possible under exponential in the dimension errors for convex bodies in the oracle model[1] (Elekes 1986). Therefore, a great effort has been devoted to randomized approximation algorithms that estimate the volume by efficiently sampling random points from the convex body. Starting with the celebrated result by Dyer, Frieze, and Kannan (1991) with complexity $O^*(d^{23})$ oracle calls[2], around three decades of algorithmic improvements reduced the exponent to 3 for well-rounded convex bodies (Cousins and Vempala 2015). Recently, even faster algorithms

[1] In the oracle model, a black box routine called oracle gives access to the convex body either by answering whether a query points lays in or out of it or by computing the intersection points of a trajectory and the body.

[2] Here, $O^*(\cdot)$ suppresses polylogarithmic factors and dependence on error parameters and d is the dimension.

have been designed and analyzed for the special case of convex polytopes (Chen, Dwivedi, Wainwright, and Yu 2018; Lee and Vempala 2018; Mangoubi and Vishnoi 2019).

All the above mentioned algorithms rely on sampling and enjoy great theoretical guarantees but they cannot be applied efficiently to real life computations. For example, the asymptotic analysis by Lovász and Vempala (2006) hides some large constants in the complexity and in (Lee and Vempala 2018) the step of the random walk used for sampling is too small to be an efficient choice in practice. Therefore, practical algorithms have been designed by relaxing the theoretical guarantees and applying new algorithmic and statistical techniques to perform efficiently while on the same time meet the requirements for high accuracy results. The first practical algorithm that scaled to high dimensions was by Emiris and Fisikopoulos (2014) (Sequence of Balls or SoB algorithm), implemented in C++, highlighted the importance of Coordinate Direction Hit and Run walk. An asymptotically faster practical algorithm followed by Cousins and Vempala (2016) (Cooling Gaussians or CG algorithm), implemented in MATLAB. However, both algorithms can handle only H-polytopes (i.e., sets of linear inequalities) in high-dimensions. A more recent algorithm by Chalkis, Emiris, and Fisikopoulos (2019) (Cooling Bodies or CB algorithm) is designed to scale in high dimensions for other polytope representations as well such as V-polytopes (i.e., convex hulls of sets of points) and Z-polytopes (i.e. Minkowski sums of segments). In this paper, high dimensions typically mean order of few hundreds, which is today's limit in practical volume approximation.

Geometric random walks is an undoubted active area in the intersection of convex geometry and statistics. The Markov chain method by Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953) was the first method to sample from high dimensional distributions and has been used extensively for numerical problems in statistical mechanics. Several research results were based on this method before Hastings (1970) provide a generalization of Metropolis method. Since then, a great amount of effort devoted to high dimensional sampling with Markov chain algorithms. Let us stand out a few important and famous algorithms in the sequel. Geman and Geman (1984) introduce an algorithm known as Gibbs sampler is commonly used in statistical inference. Hamiltonian Monte Carlo (HMC) (Duane, Kennedy, Pendleton, and Roweth 1987) resulted to a remarkable empirical success over the years but only recently Mangoubi and Smith (2017); Lee, Song, and Vempala (2018); Lee, Shen, and Tian (2020) presented the first theoretical results on the mixing of the walk. For two inspirational surveys on Hamiltonian dynamics we suggest those of Neal (2011) and Betancourt (2017). Smith (1984) introduced Hit and Run algorithm, while provided simple, widely used criteria to prove the convergence to the uniform distribution. Later, Kaufman and Smith (1998) discussed efficient direction choices for Hit and Run and provided a variation of the Hit and Run walk that enjoyed a lot of attention across sciences. For example it is widely used in biology to study constraint-based models of metabolic networks (Megchelenbrink, Huynen, and Marchiori 2014; Saa and Nielsen 2016). Another famous family of algorithms is based on Langevin Monte Carlo (LMC) method and its variants (Dalalyan 2017). Recently, Shen and Lee (2019) presented an algorithm based on underdamped Langevin diffusion for sampling from log-concave distributions and proved sub-linear dependence on the input dimension for the mixing time.

An important special case of sampling is from a truncated to a convex body probability distribution. For this case, the random walks need some access to the body given by an oracle. Hit and Run requires a so called boundary oracle for the body—that is, compute the intersection points of a line and the body's boundary. The truncated version of Metropolis Hastings is called Ball walk and requires a membership oracle–that answers whether a point

falls in or out of the body. On the other hand, HMC and LMC require more involved oracles such that compute boundary reflections for the walk's trajectory (Afshar and Domke 2015). Uniform sampling from convex bodies is also of special interest where the currently best asymptotic bounds for the mixing times are presented in Laddha, Lee, and Vempala (2019, Table 1, p. 3). What is missing from that Table is the Coordinate Directions Hit and Run (Smith 1984) and the Billiard walk (Polyak and Gryazina 2014) since there are not known bounds for their mixing times. However, there is experimental evidence for their superior performance in practice (Polyak and Gryazina 2014; Haraldsdòttir, Cousins, Thiele, Fleming, and Vempala 2017).

Considering R software, there are several CRAN packages that provide sampling from multivariate distributions. There are two main categories, truncated and untruncated distributions. For the first category an important case is the truncated Gaussian distribution. For this case there is **tmg** implementing exact HMC with boundary reflections as well as **multinomineq, restrictedMVN, lineqGPR, tmvmixnorm** implementing variations of the Gibbs sampler. The latter can also be used to sample from the t-distribution. To generate uniformly distributed points from a convex polytope there exist **hitandrun**. For the untruncated case, packages **HybridMC, rhmc** provide implementations for HMC without truncation on the target space, while there are two more packages, i.e. **mcmc** and **MHadaptive**, to sample from any distribution using the Metropolis Hastings algorithm. For volume computation in R there exist only one package, namely **geometry**, which provides a deterministic algorithm for volume computation of the convex hull of a set of points; it is based on the C++ package **qhull** (Barber, Dobkin, and Huhdanpaa 1996).

In this paper, we present **volesti**[3], a C++ library with an R interface, that contains a variety of high-dimensional sampling and volume computation algorithms. In particular, it includes efficient implementations of all three practical volume algorithms noted above—that is SoB, CG and CB. Moreover, **volesti** provides efficient implementations for the following random walks (defined in Section 2.2):

- Random-Directions Hit and Run (RDHR) (Smith 1984)
- Coordinate-Directions Hit and Run (CDHR) (Smith 1984)
- Ball Walk (BaW) (Hastings 1970)
- Billiard Walk (BiW) (Polyak and Gryazina 2014)

All the methods above can be used to sample from multivariate uniform or spherical Gaussian distributions (centered at any point), except BiW which, by definition, can be employed only for uniform sampling. The novelty of **volesti** is highlighted by the following points:

(a) is the first R package for efficient high dimensional volume estimation,

(b) is the first open source software that handles efficiently three different types of polyhedra in high dimensions,

(c) provides non existing in R sampling algorithms from uniform and Gaussian distributions,

(d) solves so far intractable problems using R (Section 4).

[3] https://github.com/GeomScale/volume_approximation/tree/v1.1.1

Let us now illustrate the power of **volesti** with few examples of volume estimation and sampling. The scripts of the examples in this paper use some R libraries either for comparison with **volesti** or for plotting. To run all the scripts successfully the reader should enable all those libraries by the following R command:

```
library(volesti, hitandrun, geometry, SimplicialCubature, ggplot2, plotly,
        latex2exp, rgl)
```

Let as start with a simple example by computing the volume of the 10-dimensional cube $[-1, 1]^{10}$, which is know to be 1024.

```
P = gen_cube(10, 'H')
cat(volume(P, seed = 5))
```

```
1022.408
```

Since this is a randomized algorithm it makes sense to compute some statistics for the output values using R. Now we compute the volume of a 40-dimensional cube.

```
P = gen_cube(40, 'H')
cat(P$volume)
```

```
1.099512e+12
```

```
volumes <- list()
for (i in 1:20) {
  volumes[[i]] <- volume(P, settings = list("error" = 0.2))
}
options(digits=10)
summary(as.numeric(volumes))
```

```
        Min.      1st Qu.       Median         Mean      3rd Qu.         Max.
7.998443e+11 9.505622e+11 1.029577e+12 1.035344e+12 1.116768e+12 1.303105e+12
```

By changing the error to 0.02 we can obtain more accurate results.

```
        Min.      1st Qu.       Median         Mean      3rd Qu.         Max.
1.004643e+12 1.032829e+12 1.072707e+12 1.086475e+12 1.110526e+12 1.225569e+12
```

To understand the need of randomized computation in high dimensions implemented in **volesti** we compare with the state-of-the-art volume computation in R today i.e., **geometry** that implements a deterministic algorithm. In dimension 15 using **geometry** is more efficient than **volesti** while starting in dimension 20 **geometry** terminates with error, hence **volesti** is the only way to estimate the volume. The following example illustrates this behavior using random 15- and 20-dimensional polytopes with 30 and 40 vertices each.

```
P = gen_rand_vpoly(15, 30, generator = 'cube', seed = 1729)
time1 = system.time({geom_values = geometry::convhulln(P$V, options = 'FA')})
time2 = system.time({vol2 = volume(P, settings = list("algorithm" = "CB",
                                                      "random_walk" = "BiW"),
                                   seed = 5) })
cat(time1[3], time2[3])
```

```
2.482 7.879
```

```
cat(geom_values$vol, vol2, abs(geom_values$vol-vol2)/geom_values$vol)
```

```
6.613415e-05 6.426176e-05 0.02831212
```

```
P = gen_rand_vpoly(20, 40, generator = 'cube', seed = 1729)
time1 = system.time({geom_values = geometry::convhulln(P$V, options = 'FA')})
```

```
QH6082 qhull error (qh_memalloc): insufficient memory to allocate 1329542232
bytes
```

```
time2 = system.time({ vol2 = volume(P, seed = 5) })
cat(time2[3], vol2)
```

```
14.524 2.68443e-07
```

The R package **hitandrun** (van Valkenhoef and Tervonen 2019) implements RDHR for uniform sampling from convex polytopes. The following R script compares the running time of **hitandrun** with **volesti**[4] on the 100-dimensional hypercube $[-1, 1]^{100}$. In terms of convergence to target distribution, **hitandrun** behaves similarly with the RDHR from **volesti** (see Figure 4).

```
d = 100
P = gen_cube(d, 'H')

constr = list("constr" = P$A, "dir" = rep("<=", 2 * d), "rhs" = P$b)
time1 = system.time({ points1 = t(hitandrun::hitandrun(constr = constr,
                                                       n.samples = 1000,
                                                       thin = 1)) })
time2 = system.time({ points2 = sample_points(P, random_walk =
                                                  list("walk" = "RDHR",
                                                       "walk_length" = 1),
                                              n = 1000, seed = 5) })
cat(time1[3], time2[3])
```

```
47.483  0.010
```

[4]For more detailed comparison with **hitandrun** see https://github.com/GeomScale/volume_approximation/wiki/volesti-vs-hitandrun

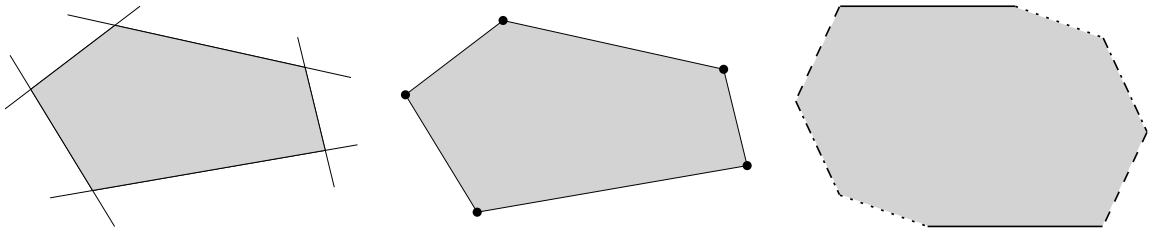

Figure 1: Examples of three different polytope representations. From left to right: an H-polytope, a V-polytope and a Z-polytope (a sum of four segments).

The case of sampling from the truncated Gaussian distribution is more involved. There is a number of R packages described above that implement various sampling algorithms. **volesti** adds two new ones, namely RHDR and CDHR, to this list, with the latter being the most efficient per random walk's step. However, since there are not known bounds for the mixing times of most of those walks it is difficult to compare them. An analytical as well as an empirical comparison would be of great interest but beyond the limits of this paper.

In Section 4 we demonstrate how our software can be useful for several applications. Starting with computational finance we exploit **volesti**'s functionality and the framework given by Calès, Chalkis, Emiris, and Fisikopoulos (2018) to perform complex computations on real-world open data for financial crisis detection. Volume approximation of Z-polytopes can be used to evaluate Z-polytope over-approximation methods in mechanical engineering. This evaluation process involves a volume computation and thus, due to the curse of dimensionality, was limited to low dimensions only (typically less than 15) before the use of **volesti**. Section 4.3 presents how **volesti** algorithms can be used for high dimensional Monte Carlo integration of a multivariate function over a convex polytope. We are not aware of any other R package to perform such computations in high dimensions e.g. more than 15. Moreover, volume can be used to solve hard problems in combinatorics such as counting the number of linear extensions of an acyclic digraph (Section 4.4). Last but not least, motivated by a problem in bio-geography we use our package to approximate the volume of the intersection of two V-polytopes (Section 4.5).

Paper organization. We continue our presentation with Section 2 that provides all the necessary definitions of objects and algorithmic tools that are available in **volesti**, thus providing the technical background needed to understand the package description in Section 3. Section 4 illustrates how **volesti** can be applied in various applications ranging from finance to combinatorics and engineering.

2. Algorithms and polytopes

2.1. Convex polytopes

Convex polytopes are a special case of convex bodies with special interest in many scientific fields and applications. For example, in optimization the feasible region of a linear program is a polytope and in finance the space of portfolios is usually expressed by a polytope (i.e. the simplex).

More formally, an H-polytope is defined as

$$P := \{x \mid Ax \leq b\} \subseteq \mathbb{R}^d$$

where $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^m$, and we say that P is given in H-representation. Each row $a_i^T \in \mathbb{R}^d$ of matrix A corresponds to a normal vector that defines a halfspace $a_i^T x \leq b_i,\ i = [m]$. The intersection of those halfspaces defines the polytope P and the hyperplanes $a_i^T x = b_i,\ i = [m]$ are called facets of P.

A V-polytope is given by a matrix $V \in \mathbb{R}^{d \times n}$ which contains n points column-wise, and we say that P is given in V-representation. The points of P that cannot be written as convex combinations of other points of P are called vertices. The polytope P is defined as the convex hull of those vertices, i.e. the smallest convex set that contains them. Equivalently, a V-polytope can be seen as the linear map of the canonical simplex $\Delta^{n-1} := \{x \in \mathbb{R}^n \mid x_i \geq 0,\ \sum_{i=1}^n x_i = 1\}$ according to matrix V, i.e.,

$$P := \{x \in \mathbb{R}^d \mid \exists y \in \Delta^{n-1} : x = Vy\}$$

A Z-polytope (or zonotope) is given by a matrix $G \in \mathbb{R}^{d \times k}$, which contains k segments column-wise, which are called generators. In this case, P is defined as the Minkowski sum of those segments and we say that it is given in Z-representation. We call *order* of a Z-polytope the ratio between the number of segments over the dimension. Equivalently, P can be expressed as the linear map of the hypercube $[-1, 1]^k$ with matrix G, i.e.

$$P := \{x \in \mathbb{R}^d \mid \exists y \in [-1, 1]^k : x = Gy\}.$$

Thus, a Z-polytope is a centrally symmetric convex body, as a linear map of an other centrally symmetric convex body.

Examples of an H-polytope, a V-polytope and a Z-polytope in two dimensions are illustrated in Figure 1. For an excellent introduction to polytope theory we recommend the book of Ziegler (1995).

2.2. Sampling and geometric random walks

We define here more formally the four geometric random walks implemented in **volesti**, namely, Hit and Run (two variations, RDHR and CDHR), Ball walk (Baw) and Billiard walk (BiW). They are illustrated in Figure 2 for two dimensions. All walks, except BiW, can be used to sample approximately from any distribution truncated in P, while BiW can be used only to generate approximate uniformly distributed points in P.

In general if $f : \mathbb{R}^n \to \mathbb{R}_+$ is a non-negative integrable function then it defines a measure π_f on any measurable subset A of $\mathbb{R}^d$,

$$\pi_f(A) = \frac{\int_A f(x)dx}{\int_{\mathbb{R}^d} f(x)dx}$$

Let ℓ be a line in $\mathbb{R}^d$ and let $\pi_{\ell,f}$ be the restriction of π to ℓ,

$$\pi_{\ell,f}(P) = \frac{\int_{p+tu \in P} f(p + tu)dt}{\int_\ell f(x)dx}$$

where p is a point on ℓ and u a unit vector parallel to ℓ. The following pseudocode describes one step of Hit and Run.

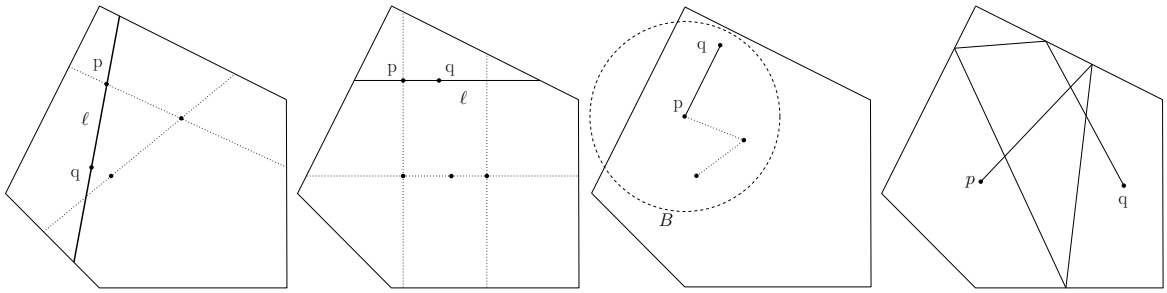

Figure 2: Examples of random walks. From left to right: RDHR, CDHR, BaW, BiW; p is the point at current step and q the new point computed. Dotted lines depict previous steps.

Hit and Run(P, p, f): Polytope $P \subset \mathbb{R}^d$, point $p \in P$, $f : \mathbb{R}^d \to \mathbb{R}_+$

1. Pick a line ℓ through p.
2. **return** a random point on the chord $\ell \cap P$ chosen from the distribution $\pi_{\ell,f}$.

It is easy to prove that π_f is the stationary distribution of the random walk. When the line ℓ in line (1.) of the pseudocode is chosen uniformly at random from all possible lines passing through p then the walk is called Random-Directions Hit and Run (Smith 1984). To pick a random direction through point $p \in \mathbb{R}^d$ we could sample d numbers $g_1, \ldots, g_d$ from $\mathcal{N}(0, 1)$ and then the vector $u = (g_1, \ldots, g_d)/\sqrt{\sum g_i^2}$ is uniformly distributed on the surface of the d-dimensional unit ball. A special case is called Coordinate-Directions Hit and Run (Smith 1984) where we pick ℓ uniformly at random from the set of d lines that passing through p and are parallel to the coordinate axes.

The Ball walk needs, additionally to Hit and Run, a radius δ as input. In particular, Ball walk is Metropolis Hastings (Hastings 1970) when the target distribution is truncated. The following pseudocode describes one step of Ball walk with a Metropolis filter.

Ball Walk(P, p, δ, f): Polytope $P \subset \mathbb{R}^d$, point $p \in P$, radius δ, $f : \mathbb{R}^d \to \mathbb{R}_+$

1. Pick a uniform random point x from the ball of radius δ centered at p
2. **return** x with probability $\min\left\{1, \frac{f(x)}{f(p)}\right\}$; **return** p with the remaining probability.

Again it is easy to prove that π_f is the stationary distribution. If $f(x) = e^{-||x-x_0||^2/2\sigma^2}$ then for both Hit and Run and Ball walk, the target distribution is the multidimensional spherical Gaussian with variance σ^2 and its mode at x_0 and if it is the indicator function of P then the target distribution is the uniform distribution.

Billiard walk, is a random walk for sampling from the uniform distribution (Polyak and Gryazina 2014). It tries to emulate the movement of a gas particle during the physical phenomena of filling uniformly a vessel. The following pseudocode implements one step of Billiard walk, where $\langle \cdot, \cdot \rangle$ is the inner product between two vectors, $|| \cdot ||$ is the ℓ^2 norm and $| \cdot |$ is the length of a segment.

Billiard walk(P, p, τ, R): Polytope $P \subset \mathbb{R}^d$, current point of the random walk $p \in P$, parameter $\tau \in \mathbb{R}_+$, upper bound on the number of reflections $R \in \mathbb{N}$

1. Set the length of the trajectory $L \leftarrow -\tau \ln \eta$, $\eta \sim \mathcal{U}(0,1)$;
   Set the number of reflections $n \leftarrow 0$ and $p_0 \leftarrow p$;
   Pick a uniformly distributed direction on the unit sphere, v;
2. Update $n \leftarrow n+1$; **If** $n > R$ **return** p_0;
3. Set $\ell \leftarrow \{p + tv, 0 \leq t \leq L\}$;
4. **If** $\partial P \cap \ell = \emptyset$ **return** $p + Lv$;
5. Update $p \leftarrow \partial P \cap \ell$; Let s be the inner normal vector of the tangent plane on p, s.t. $||s|| = 1$; Update $L \leftarrow L - |P \cap \ell|$, $v \leftarrow v - 2\langle v, s\rangle s$; **goto** 2;

In general, when starting a random walk from a point $p \in P$ the more points we generate the less correlated with p will be. The number of random walk steps to get an uncorrelated point, that is a point approximately drawn from π_f, is called mixing time. We call cost per step the number of operations performed to generate a point. Hence, the total cost to generate a random point is the mixing time multiplied by the cost per step.

random walk	mixing time	cost/step H-polytope	cost/step V- & Z-polytope
RDHR (Lovász and Vempala 2006)	$O^*(d^3)$	$O(md)$	2 LPs
CDHR (Smith 1984)	**?**	$O(m)$	2 LPs
BaW (Lee and Vempala 2017)	$O^*(d^{2.5})$	$O(md)$	1 LP
BiW (Polyak and Gryazina 2014)	**?**	$O((d+R)m)$	R LPs

Table 1: Overview of the random walks implemented in **volesti**. LP for linear program and R for the number of reflections per point in BiW.

Table 1 displays known complexities for mixing time and cost per step. For the mixing time of RDHR we assume that P is well rounded, i.e. $B_d \subseteq P \subseteq C\sqrt{d}B_d$, where B_d is the unit ball and C a constant. In general if $rB_d \subseteq P \subseteq RB_d$ then RDHR mixing time is $O^*(d^2(R/r)^2)$. For the mixing time of Ball walk in Table 1 we assume that P is in isotropic position and the radius of the ball is $\delta = \Theta(1/\sqrt{d})$ (Lee and Vempala 2017). There are no theoretical bounds on mixing time for CDHR and BiW. Polyak and Gryazina (2014) experimentally show that BiW converges faster than RDHR when $\tau \approx diam(P)$, i.e. the diameter of P. CDHR is the main paradigm for sampling in practice from H-polytopes, e.g. in volume computation (Emiris and Fisikopoulos 2014) and biology (Haraldsdòttir *et al.* 2017). The main reason behind this is the small cost per step and the same convergence in practice as RDHR (Emiris and Fisikopoulos 2014). For V- and Z-polytopes the cost per step of BiW is comparable with that of CDHR and moreover, converges fast to the uniform distribution (Chalkis *et al.* 2019). The fact that all above walks are implemented in **volesti** enable us to empirically evaluate their mixing time using R (e.g., Figure 4).

Apart from sampling from the interior of convex polytopes, there are methods for sampling from their boundary. They are based on RDHR/CDHR and are called BRDHR/BCDHR

respectively in this paper. They performs as RDHR/CDHR but store only the extreme/ boundary points. There is no theoretical guarantee that these methods converge to the uniform distribution, as the Shake-and-Bake algorithm (Dieker and Vempala 2015), but there is some experimental evidence that they both do so (Schneebeli 2015). Both BRDHR and BCDHR are implemented in **volesti**.

2.3. Volume estimation

As mentioned before, volume computation is a hard problem, so given a polytope P we have to employ randomized algorithms to approximate $\text{vol}(P)$ within some target relative error ϵ and high probability. The keys to success of those algorithms are the Multiphase Monte Carlo (MMC) technique and sampling from multivariate distributions with geometric random walks (Section 2.2).

In particular, we define a sequence of functions $\{f_0, \ldots, f_q\}$, $f_i : \mathbb{R}^d \to \mathbb{R}$. Then $\text{vol}(P)$ is given by the following telescopic product:

$$\text{vol}(P) = \int_P dx = \int_P f_q(x)dx \frac{\int_P f_{q-1}(x)dx}{\int_P f_q(x)dx} \cdots \frac{\int_P dx}{\int_P f_0(x)dx} \tag{1}$$

Then, we need to:

- Fix the sequence such that q is as small as possible.
- Select f_i such that each integral ratio can be efficiently computed.
- Compute $\int_P f_q(x)dx$.

For a long time researchers, e.g., Lovász, Kannan, and Simonovits (1997), set f_i to be indicator functions of concentric balls intersecting P (Figure 3). It follows that $\int_P f_i(x)dx = \text{vol}(B_i \cap P)$ and the sequence of convex bodies $P = P_1 \supseteq \cdots \supseteq P_q$, $P_i = B_i \cap P$ forms a telescopic product of ratios of volumes, while for $\text{vol}(P_q)$ there is a closed formula. Assuming $rB_d \subset P \subset RB_d$, then $q = O(d \lg R/r)$. The trick now is that we do not have to compute the exact value of each ratio $r_i = \text{vol}(P_i)/\text{vol}(P_{i+1})$, but we can use sampling-rejection to estimate it within some target relative error ϵ_i. If r_i is bounded then $O(1/\epsilon_i^2)$ uniformly distributed points in P_{i+1} suffices. Another crucial aspect is the sandwiching ratio R/r of P which has to be as small as possible. This was tackled by a rounding algorithm, that is bringing P to nearly isotropic position (Lovász *et al.* 1997).

The SoB algorithm follows this paradigm and deterministically defines the sequence of P_i such that $0.5 \leq \text{vol}(P_i)/\text{vol}(P_{i+1}) \leq 1$. They also introduce a practical method for rounding P to reduce the sandwiching ratio by skipping the transformation to nearly isotropic position.

Lovász and Vempala (2006) were the first to set f_i to general functions and thus asymptotically reduce q, i.e., the length of the sequence of f_i. In the CG algorithm, each f_i is a spherical multidimensional Gaussian distribution. In particular, they are based on (Cousins and Vempala 2015) with an annealing schedule (Lovász and Vempala 2006) to fix the sequence of Gaussians. The sequence is parameterized by the variances $\sigma_0^2 < \cdots < \sigma_q^2$. They compute an inscribed ball (ideally the largest one) of P and use the number of facets to set the variance of the first Gaussian so that $Pr[x \notin P] \leq \epsilon$, $x \sim \mathcal{N}(0, \sigma_0^2 I)$.

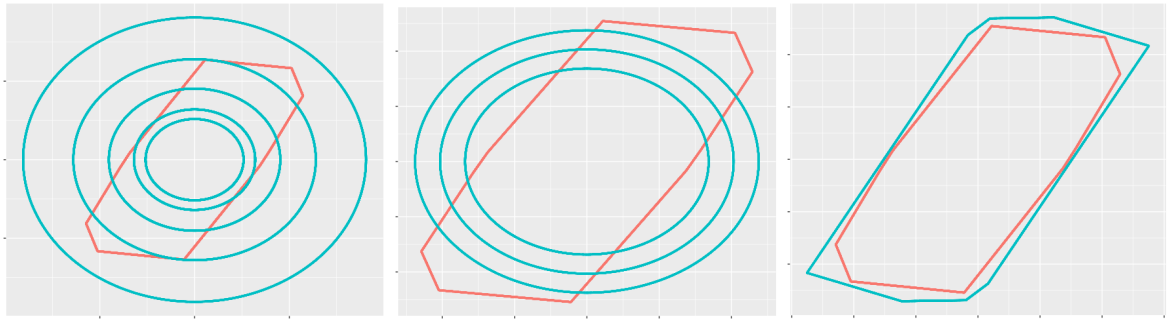

Figure 3: Sequences of bodies (blue) in various MMC procedures for a 2-dimensional zonotope (red). The sequence of balls computed by MMC as in (Emiris and Fisikopoulos 2014) (left). Sequence of balls (middle) and sequence of H-polytopes (right) as computed by MMC of Chalkis *et al.* (2019). Here, $r = 0.8$ and $r + \delta = 0.85$.

Algorithm	H-polytope		V- & Z-polytope
	$d \leq 200$	$d > 200$	
SoB Emiris and Fisikopoulos (2014)			
CG (Cousins and Vempala 2016)		✓	
CB (Chalkis *et al.* 2019)	✓		✓

Table 2: Performance of the volume estimation algorithms in **volesti**. The most efficient algorithm for each scenario is checked.

Motivated by polytopes where the boundary is unknown (e.g., V- and Z-polytopes) CB algorithm use a simulated annealing technique in order to minimize q by defining the sequence of P_i s.t. $r_i = \text{vol}(P_i)/\text{vol}(P_{i+1}) \in [r, r+\delta]$ with high probability, where $r, \delta > 0$ and $0 < r+\delta < 1$ are given as inputs by the user. They also exploit the fast convergence of BiW by showing experimentally that just $O^*(1)$ BiW points per ratio suffices. Last, the simulated annealing allow to generalize the MMC in Figure 3, by using any body C in MMC, instead of a ball. This is crucial when P is a Z-polytope as they can compute bodies that fits better to the input polytope than the ball. Thus, skipping the rounding step and reduce further the number of bodies in MMC than the rounding step could do. For example, in Figure 3 (middle) the algorithm requires 5 balls, but in the right plot it computes an enclosing body and thus a single rejection step suffices.

Table 2 sums up the performance of the three practical methods implemented in **volesti**. The CB algorithm is the most efficient choice for H-polytopes in less than 200 dimensions and for V- and Z-polytopes in any dimension. For the rest of the cases the user should choose CG algorithm. Preliminary tests show that SoB algorithm has the largest probability to approximate $\text{vol}(P)$ within a target relative error ϵ but it is unclear whether this is the rule and further benchmarks are needed.

3. Package

Our package **volesti** combines the efficiency of C++ and the popularity and usability of R. Currently, the package consists of around $10K$ lines of C++ and R code. The package is using

the **eigen** library (Guennebaud, Jacob *et al.* 2010) for linear algebra, **lpsolve** library (Berkelaar, Eikland, and Notebaert 2004) for solving linear programs and **boost random** library (Maurer and Watanabe 2017) (part of Boost C++ libraries) for random numbers and distributions. All the code development is performed on

`https://github.com/GeomScale/volume_approximation`

The package is available in Comprehensive R Archive Network (CRAN) and is regularly updated with new features and bug-fixes (Fisikopoulos and Chalkis 2019). We employ continuous integration for maintainable and testing the C++ part of **volesti**[5]. Additionally, there is a separate test suite for the functions of the R package.

There is a detailed documentation of all the exposed R classes and functions publicly available. We maintain a contribution tutorial[6] to help users and researchers who want to contribute to the development or propose a bug-fix. The package is shipped under the `LGPL-3` license to be open to all the scientific communities.

We use **Rcpp** (Eddelbuettel and François 2011) to interface C++ with R. In particular, we create one **Rcpp** function for each procedure (such as sampling, volume estimation etc.) and we export it as an R function. To export C++ classes that represent convex polytopes we use **Rcpp** modules (Eddelbuettel and François 2017). Currently, the package provides 10 functions, 4 exposed classes for convex polytopes and 8 polytope generators, explained in detail in the following subsections.

3.1. Polytope classes and generators

Our package **volesti** comes with 4 classes to handle different representations of polytopes. Table 3 demonstrates the exposed R classes. The name of the classes are the names of polytope representations as defined in Section 2.1. There is one more class called `VpolytopeIntersection` that represents the intersection of two V-polytopes. Each polytope class has a few variable members that describe a specific polytope, demonstrated in Table 3. The matrices and the vectors in Table 3 correspond to those of Section 2.1 which define a polytope of a particular representation. The `integer` variable `type` implies the representation: `1` is for H-polytopes, `2` for V-polytopes, `3` for Z-polytopes and `4` for the intersection of two V-polytopes. The `numerical` variable `volume` corresponds to the volume of the polytopes if it is known. The **volesti**'s generators of standard, well studied polytopes assign the value of the exact volume to this variable.

The set of polytopes' generators can be used to create and test a variety of different input data. All the random generators could take a generator seed as input.

- `gen_cube(dimension = integer, representation = string)` generates a cube-d: $\{x = (x_1, \ldots, x_d) \,|\, x_i \leq 1, x_i \geq -1, x_i \in \mathbb{R} \text{ for all } i = 1, \ldots, d\}$. The exact volume is equal to 2^d.
- `gen_cross(dimension = integer, representation = string)` generates a cross-d: cross polytope, the dual of cube, i.e. $\text{conv}(\{-e_i, e_i, i = 1, \ldots, d\})$. The exact volume is equal to $2^d/d!$.

[5] `https://circleci.com/gh/GeomScale/volume_approximation`
[6] `https://github.com/GeomScale/volume_approximation/blob/develop/CONTRIBUTING.md`

Class	Constructor	Variable members

`Hpolytope`	`Hpolytope$new(A,b)`	matrix $A \in \mathbb{R}^{m \times d}$
		vector $b \in \mathbb{R}^m$
		`integer type`
		`numerical volume`

`Vpolytope`	`Vpolytope$new(V)`	matrix $V \in \mathbb{R}^{n \times d}$
		`integer type`
		`numerical volume`

`Zonotope`	`Zonotope$new(G)`	matrix $G \in \mathbb{R}^{k \times d}$
		`integer type`
		`numerical volume`

`VpolytopeIntersection`	`VpolytopeIntersection$new(V1,V2)`	matrix $V1 \in \mathbb{R}^{n_1 \times d}$
		matrix $V2 \in \mathbb{R}^{n_2 \times d}$
		`integer type`
		`numerical volume`

Table 3: Overview of the polytopes' classes in **volesti**.

- `gen_simplex(dimension = integer, representation = string)` generates a Δ-d: the d-dimensional simplex $\text{conv}(\{e_i, \text{ for } i = 1, \ldots, d\})$. The exact volume is equal to $1/d!$.

- `gen_rand_vpoly(dimension = integer, nvertices = integer, generator = string, seed = integer)`. When `generator = "cube"` generates a V-polytope with n non redundant vertices, uniformly distributed in the hypercube $[-1, 1]^d$. When `generator = "sphere"` (default value) generates a V-polytope with n vertices uniformly distributed on the d-dimensional unit sphere.

- `gen_rand_zonotope(dimension = integer, nsegments = string, generator = string, seed = integer)`. When `generator = "uniform"` (default value) the length of each d-dimensional segment is picked uniformly from the interval $[0, 100]$. When `generator = "gaussian"` the length of each d-dimensional segment is picked from $\mathcal{N}(50, (50/3)^2)$ truncated to $[0, 100]$. When `generator = "exponential"` the length of each d-dimensional segment is picked from $Exp(1/30)$ truncated to $[0, 100]$.

- `gen_skinny_cube(dimension = integer)` generates the skinny cube $[-1, 1]^{d-1} \times [-100, 100]$; it is available only in H-representation. The exact volume is equal to $100 \cdot 2^d$.

- `gen_prod_simplex(dimension = integer)` generates the Cartesian product of two unit simplices in $\mathbb{R}^d$; it is available only in H-representation. The exact volume is equal to $1/d!^2$.

- `gen_rand_hpoly(dimension = integer, nfacets = integer, seed = integer)` generates a random H-polytope. We choose m hyperplane tangent to the hypersphere of

radius 10 centered at the origin and for each one we construct a halfspace that contains the center of the hypersphere.

Note that for all the random Z-polytope generators we pick the direction of each one of the k segments to be a uniformly distributed vector on the unit sphere. When the exact volume of the polytope is known the generator sets the value of the class variable `volume` equal to that value; otherwise the default value is `NaN`. For the first three generators the user can choose the representation of the generated polytope by setting `representation` either to `Hpolytope` or `Vpolytope` by giving `"H"` or `"V"` as input respectively. The rest of the generators come with a fixed output representation.

3.2. Sampling convex polytopes

In this subsection we present all the functions provided by **volesti** to sample from convex polytopes. Moreover we demonstrate how these functions can be used for empirical study of mixing time, or plotting 2d polygons.

Sampling via random walks

An important aspect of **volesti** is approximate sampling from convex bodies with uniform or spherical Gaussian target distribution using the 4 geometric random walks mentioned in Section 2.2. The sampling function is,

```
sample_points(P = Rcpp_class, n = integer, distribution = List,
              random_walk = List, seed = integer)
```

where `distribution`, `random_walk` and `seed` can be omitted and the default values are going to be used. The default target distribution is the uniform distribution. However, if Gaussian is selected by the user, the default variance is 1 and the default mode is the center of the inscribed ball $rB_d \subseteq P$ (see Section 2.3). The default random walk for the uniform distribution is BiW with walk length equal to 5 for all the representations. For the Gaussian distribution the default random walk is CDHR for H-polytopes and RDHR for all the other representations with walk length equal to $\lfloor 10 + d/10 \rfloor$ for both random walks. The default starting point for all the random walks is also the center of rB_d. The default number of points `nburns` to burn before start sampling is 0, the default maximum length `L` of the BiW trajectory is $2dr$ and the default radius for the BaW is $\delta = 4r/\sqrt{d}$ when the target distribution is the uniform distribution, otherwise $\delta = 4r/\sqrt{\max\{1, 1/2\sigma^2\}d}$ when the latter is the spherical Gaussian distribution. The function is parameterized by:

1. A convex polytope P in any of the 4 representations described in Section 3.1.
2. The number of points n, to sample from P.
3. The `List distribution` is used to set the target distribution with elements: (i) `"density" = {"uniform", "gaussian"}`, (ii) `"variance" = numeric` and (iii) `"mode" = vector`.

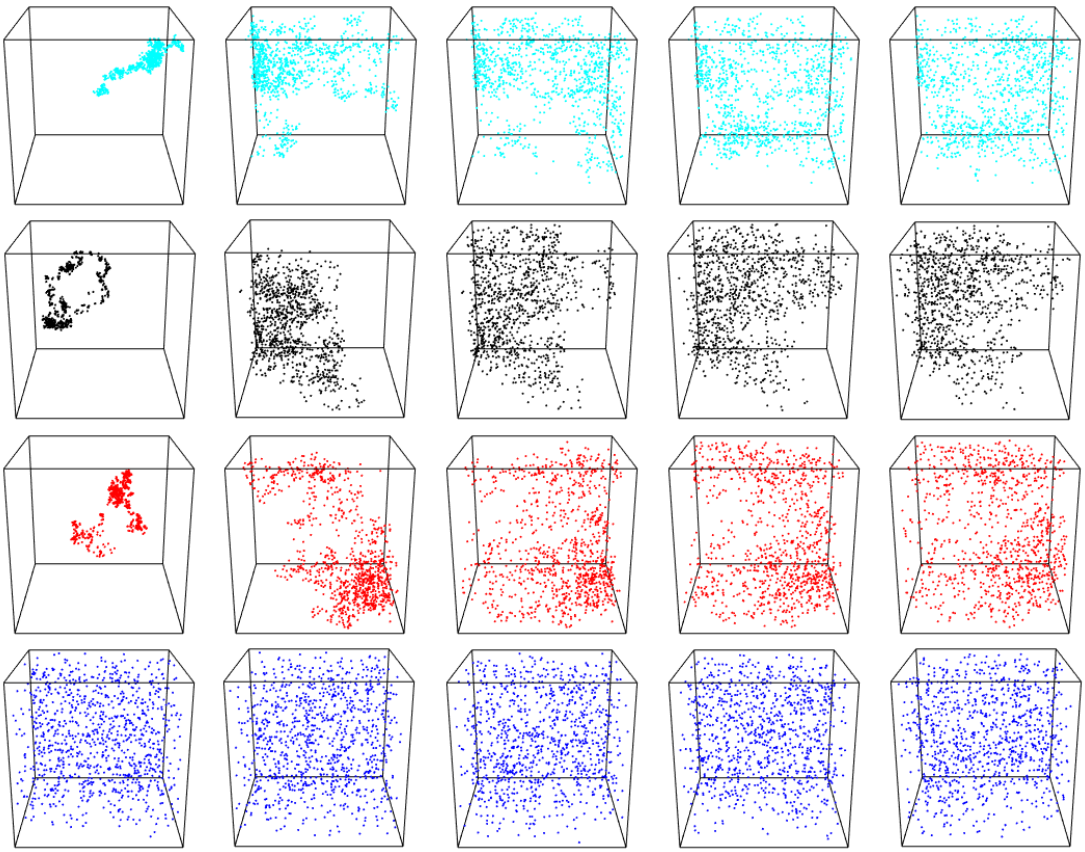

Figure 4: Uniform sampling from a random rotation of the hypercube $[-1,1]^{200}$. We map the sample back to $[-1,1]^{200}$ and then we project them on $\mathbb{R}^3$ by keeping the first three coordinates. Each row corresponds to a different walk: BaW, CDHR, RDHR, BiW. Each column to a different walk length: {1, 50, 100, 150, 200}. That is, the sub-figure in third row and forth column corresponds to RDHR with 150 walk length.

4. The `List random_walk` to choose and parameterize the random walk, with elements: (i) `"walk"= {"CDHR", "RDHR", "BiW", "BaW", "BRDHR", "BCDHR"}`, (ii) `"walk_length"= integer`, (iii) `"starting_point" = vector`, (iv) `"nburns" = integer`, (v) `"BaW_rad" = numeric` and (vi) `"L" = numeric`.

5. A random `seed`.

Using **volesti** and R we can empirically study the mixing time of the geometric random walks implemented in **volesti**. To this end, we sample uniformly from a random rotation of the 200-dimensional hypercube $[-1,1]^{200}$. Then we map the points back to $[-1,1]^{200}$ using the inverse transformation and then we project all the sample points on $\mathbb{R}^3$, or equivalently on the 3D cube $[-1,1]^3$, by keeping the first three coordinates. We plot the results in Figure 4. The following script performs those computations (plot scripts in the Appendix A.1). The function `rotate_polytope()` rotates randomly a polytope and returns the rotated polytope and the matrix of the linear transformation; for more details see Section 3.6.

```
d = 200
```

```
num_of_points = 1000
P = gen_cube(d, 'H')
retList = rotate_polytope(P, seed = 5)
T = retList$T
P = retList$P

for (i in c(1, seq(from = 50, to = 200, by = 50))){
  points1 = t(T) %*% sample_points(P, n = num_of_points,
                                   random_walk = list("walk" = "BaW",
                                                      "walk_length" = i),
                                   seed = 5)
  points2 = t(T) %*% sample_points(P, n = num_of_points,
                                   random_walk = list("walk" = "CDHR",
                                                      "walk_length" = i),
                                   seed = 5)
  points3 = t(T) %*% sample_points(P, n = num_of_points,
                                   random_walk = list("walk" = "RDHR",
                                                      "walk_length" = i),
                                   seed = 5)
  points4 = t(T) %*% sample_points(P, n = num_of_points,
                                   random_walk = list("walk" = "BiW",
                                                      "walk_length" = i),
                                   seed = 5)
}
```

We note that, in general, perfect uniform sampling in the rotated polytope would result to perfect uniformly distributed points in the 3D cube $[-1, 1]^3$. Hence, Figure 4 shows an advantage of BiW in mixing time for this scenario compared to the other walks—it mixes relatively well even with one step (i.e. walk length). Notice also that the mixing of both CDHR and RDHR seem similar while it is sightly better than the mixing of BaW.

Direct sampling

volesti also provides direct uniform sampling from special bodies. By direct sampling we mean that we do not employ random walks. Typically direct sampling is more efficient and more accurate than sampling using random walks. The function that offers this option is,

```
direct_sampling(body = List, n = integer, seed = integer)
```

There are no default values for the input variables of this function, except from `seed`. It is parameterized by,

1. The `List body` to request exact uniform sampling from special well known convex bodies through the following elements:
   (i) `"type"` a string that declares the type of the body for the exact sampling with `'unit_simplex'` for the unit simplex, i.e. $\Delta^d := \{x \in \mathbb{R}^d \mid x_i \geq 0,\ \sum_{i=1}^{d} x_i \leq 1\}$,

or `'canonical_simplex'` for the canonical simplex, i.e. $\Delta^{d-1} := \{x \in \mathbb{R}^d \mid x_i \geq 0, \sum_{i=1}^{d} x_i = 1\}$, or `'hypersphere'` for the boundary of a hypersphere centered at the origin, or `'ball'` for the interior of a hypersphere centered at the origin,
(ii) `"dimension"` and `"radius"` (only for hypersphere and ball).

2. The number of points n, to sample from P.

3. A random `seed`.

The cost per uniformly distributed point for a ball/hypersphere as well as for the unit and the canonical simplex is $O(d)$ (Rubinstein and Melamed 1998). Notice that the random walk with the smallest cost per step is CDHR, which is also $O(d)$ for those bodies. However, one may need several walk steps (at least polynomial in d following the theoretical bounds; see Section 1) to generate an almost uniformly distributed point.

Direct uniform sampling from some well known families of convex bodies such as simplices or hypersphere are useful fundamental operations in many randomized algorithms in convex optimization (Dabbene, Shcherbakov, and Polyak 2010) or computational biology (Herrmann, Dyson, Vass, Johnson, and Schwartz 2019).

Generating uniformly distributed points on the boundary of a hypersphere is equivalent to generate a direction uniformly as discussed in Section 2.2. Uniform sampling from the interior of a hypersphere is used by CB algorithm. Uniform sampling from Δ^{d-1} is useful for applications in finance (Pouchkarev 2005; Guegan, Calès, and Billio 2011; Banerjee and Hung 2011), as it is equivalent to generate uniformly distributed portfolios in a stock market of d assets (see Section 4). Figure 5 illustrates the samples obtained by the following R script (plot scripts in the Appendix A.2).

```
N = 2000
points1 = direct_sampling(body = list("type" = "canonical_simplex",
                                      "dimension" = 3),
                          n = N, seed = 5)
points2 = direct_sampling(body = list("type" = "ball",
                                      "dimension" = 2),
                          n = N, seed = 5)
points3 = direct_sampling(body = list("type" = "hypersphere",
                                      "dimension" = 2),
                          n = N, seed = 5)
```

Plotting a polygon via sampling

The plots of polygons in this paper are drawn by uniform sampling from the boundary of the polygons with `sample_points()`. The following R script illustrates how **volesti** supports sampling from the boundary of the intersection of two V-polytopes illustrated in Figure 6 (plot scripts in the Appendix A.3).

```
P1 = gen_rand_vpoly(2, 7, generator = "sphere", seed = 13)
P2 = gen_rand_vpoly(2, 5, generator = "cube", seed = 151)
P3 = VpolytopeIntersection$new(V1 = P1$V, V2 = P2$V)
```

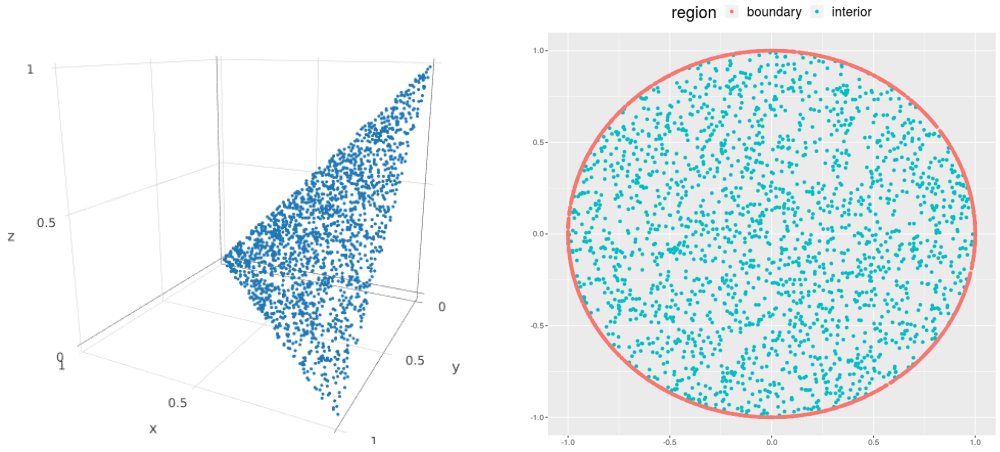

Figure 5: Uniformly distributed points from the 3D canonical simplex and from the interior and the boundary of the 2D unit ball.

```
points1 = sample_points(P1, random_walk = list("walk" = "BRDHR"), n = 10000,
                        seed = 5)
points2 = sample_points(P2, random_walk = list("walk" = "BRDHR"), n = 10000,
                        seed = 5)
points3 = sample_points(P3, random_walk = list("walk" = "BRDHR"), n = 10000,
                        seed = 5)
```

3.3. Volume estimation

A significant function for volume estimation in **volesti** is `volume()`. The user can select any of the three randomized algorithms that are implemented in **volesti** (Section 2.3). The polytope can be given in any of the four representations described in Section 3.1.

```
volume(P = Rcpp_class, settings = List, rounding = boolean, seed = integer)
```

The input variables `settings`, `rounding`, `seed` are optional and when omitted the default values are going to be used. The default algorithm is selected as suggested by Table 2. The default parameters of each algorithm are those suggested by Emiris and Fisikopoulos (2014); Cousins and Vempala (2016); Chalkis *et al.* (2019). The default random walk is CDHR for H-polytopes and BiW for V- and Z-polytopes. The default walk length is 1 for CB and CG algorithms and $\lfloor 10 + d/10 \rfloor$ for SoB. The default enclosed ball in P that the algorithm computes, depending the representation of P, is given later in this subsection. The default body that CB algorithm uses in MMC is ball, except the case of P being a Z-polytope with order < 5, where the H-polytope discussed in Section 1 is used. The default value for the error parameter is 0.1 for both CB and CG and 1 for SoB algorithm.

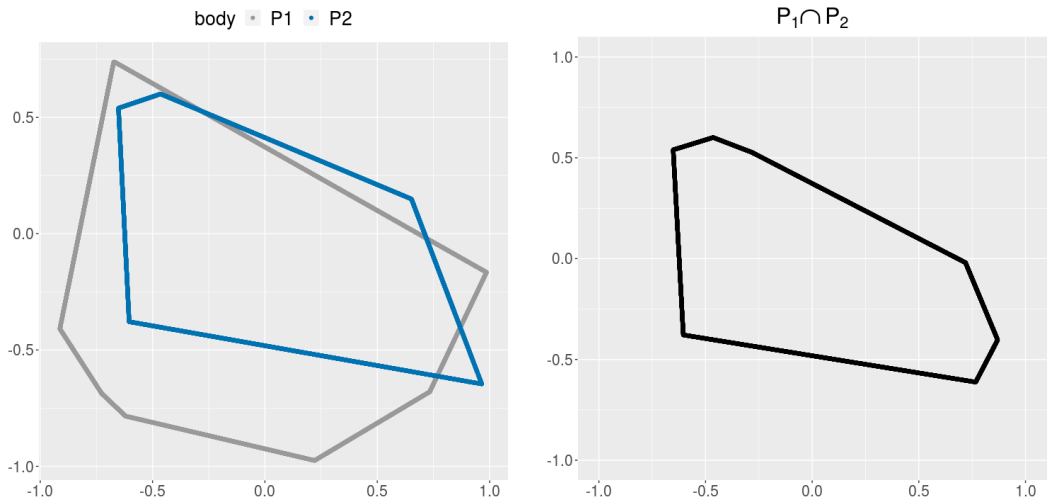

Figure 6: Left, uniform boundary sampling from two V-polytopes P_1, P_2. Right, uniform boundary sampling from the intersection of $P_1 \cap P_2$.

The function `volume()` is parameterized by:

1. The `List settings` to set the algorithm to use and the parameters of the selected algorithm: i) `"algorithm" = {"CB", "CG", "SOB"}`, ii) `"error" = numeric`, iii) `"random_walk" = {"CDHR", "RDHR", "BiW", "BaW"}`, iv) `"walk_length" = integer`, v) `"win_len" = integer` (the length of the sliding window for either CB or CG algorithm), vi) `"hpoly" = boolean` (a flag to use H-polytopes in MMC when the input polytope is a Z-polytope and the algorithm of choice is CB)

2. The `boolean` input variable `rounding` to request rounding before volume computation.

3. A random `seed`.

When P is an H-polytope **volesti** computes the largest inscribed ball, but for V- and Z-polytopes computes non optimal balls as the problem becomes computationally harder. In particular, when P is a zonotope, checking whether a ball $B \subseteq P$ is in co-NP, but it is not known whether it is co-NP-complete (Cerny 2012). When P is a V-polytope, given $p \in P$ the computation of the largest inscribed ball centered at p is NP-hard (Murty 2009). For those cases **volesti** computes suboptimal inscribed balls that work well in practice as default choices for the sampling and volume estimation algorithms implemented in **volesti**. The default computations are the following:

- **H-polytopes:** We compute the Chebychev ball (the largest inscribed ball) by solving a linear program (Boyd and Vandenberghe 2004).

- **V- and Z-polytopes:** We compute the maximal r s.t.: $re_i \in P$ for all $i = 1, \ldots, d$, then the ball centered at the origin with radius $r/\sqrt{d}$ is an inscribed ball of P.

- **Intersection of two V-polytopes P_1, P_2:** Let $V_1 \in \mathbb{R}^{d\times n_1}$, $V_2 \in \mathbb{R}^{d\times n_2}$ be the matrices that contain the n_1 vertices of P_1 and the n_2 vertices of P_2 respectively. Then $P_1 \cap P_2 \neq \emptyset$ if and only if the following linear program is feasible,

$$\left[\begin{array}{c|c} V_1 & -V_2 \end{array}\right] \left[\begin{array}{c} x_1 \\ \vdots \\ x_{n_1+n_2} \end{array}\right] = \mathbf{0},\ x_i \geq 0,\ \forall i \in [n_1+n_2],\ \sum_{i=1}^{n_1} x_i = 1,\ \sum_{j=n_1+1}^{n_1+n_2} x_j = 1.$$

  Then we employ this linear program to compute $d+1$ vertices of $P_1 \cap P_2$: For the first vertex we pick a random direction in $\mathbb{R}^{n_1+n_2}$ as a linear objective function and the optimal solution would be a vertex of $P_1 \cap P_2$. For the i-th vertex, where $1 < i \leq d+1$, we consider the computed vertices $v_1, \ldots, v_{i-1}$ as vectors and we pick an objective function from the orthogonal subspace generated by these vectors. Last, we compute the largest inscribed ball of the corresponding simplex using the algorithm provided by Murty (2009).

3.4. Exact volumes

For exact volume computation **volesti** provides support for specific convex bodies through the following function,

```
exact_vol(P = Rcpp_class, seed = integer)
```

When the input is a Z-polytope, `exact_vol()` computes and returns the exact volume by summing absolute values of determinants as proposed by Gover and Krikorian (2010). For the other representations the function returns the member variable `volume` when it has not been set to `NaN`. For well known polytopes, e.g. cubes, that have been generated by **volesti**'s random generators this variable has been assigned to the exact volume of the polytope. Otherwise the function returns an exception. The following example demonstrates how exact_vol() works.

```
P = gen_cube(100, 'H')
Z = gen_rand_zonotope(5, 10, seed = 20)
rP = gen_rand_hpoly(10, 60, seed = 11)

exact_vol1 = exact_vol(P)
exact_vol2 = exact_vol(Z)

cat(exact_vol1, exact_vol2)
```

```
1.267651e+30 155854541519
```

```
exact_vol3 = exact_vol(rP)
```

```
Error in exact_vol(rP) : Volume unknown!
Called from: exact_vol(rP)
```

Given a halfspace $H := \{x \in \mathbb{R}^d \mid a^T x \leq z_0\}$, $a \in \mathbb{R}^d$, $z_0 \in \mathbb{R}$, **volesti** provides the exact computation of $\text{vol}(\Delta^{d-1} \cap H)/\text{vol}(\Delta^{d-1})$ which is also equal to $\text{vol}(\Delta^d \cap H)/\text{vol}(\Delta^d)$, where $\Delta^d := \{x \in \mathbb{R}^d \mid x_i \geq 0, \sum_{i=1}^d x_i \leq 1\}$. This ratio of volumes is strongly related with the cross sectional score of a portfolio—in terms of return—in a stock market of d assets and therefore of special interest (see Section 4). The function `frustum_of_simplex()` implements the algorithm by Varsi (1973), which performs $O(d^2)$ operations to compute that volume ratio.

```
frustum_of_simplex(a = numeric vector, z0 = numeric)
```

To compute the frustum of an arbitrary simplex, one has i) to apply to H the linear transformation that maps the simplex to Δ^d, ii) to apply Varsi's algorithm and iii) to compute the exact volume of the simplex by calculating a determinant. The following R script demonstrates the efficiency of the algorithm in thousands dimensions. The sampled vector `a` defines the direction of a hyperplane and the sampled vector `x` is used to compute the scalar `z0` which finally defines a halfspace H that intersects the Δ^{d-1}. Notice that a few milliseconds suffices to compute the volume of a frustum of Δ^{d-1} when $d = 5\,000$.

```
d = 5000
a = t(direct_sampling(n = 1, body = list("type" = "hypersphere",
                                         "dimension" = d),
                      seed = 50))
x = direct_sampling(n = 1, body = list("type" = "canonical_simplex",
                                       "dimension" = d),
                    seed = 50)
z0 = a%*%x
tim = system.time({ volume = frustum_of_simplex(a, z0) })
cat(tim[3], volume)
```

```
0.057 0.01729134
```

3.5. Rounding polytopes

A critical complexity issue for all the volume estimation algorithms implemented in **volesti** is to bring a skinny polytope to well-rounded or isotropic position. To achieve this goal in practice, **volesti** follows the method proposed by Emiris and Fisikopoulos (2014). For H- and Z-polytopes the method generates $10d$ uniformly distributed points in P and computes an approximation of the minimum volume enclosing ellipsoid of that pointset using Khachiyan's algorithm (Todd and Yildirim 2007) and the implementation by Nikolic (2015). Then it maps the ellipsoid to the unit ball and applies the same linear transformation to P. When P is a V-polytope **volesti** computes the same ellipsoid, but now the pointset consists of the vertices of P. This rounding preprocessing can be also computed by the function,

```
round_polytope(P = Rcpp_class, seed = integer)
```

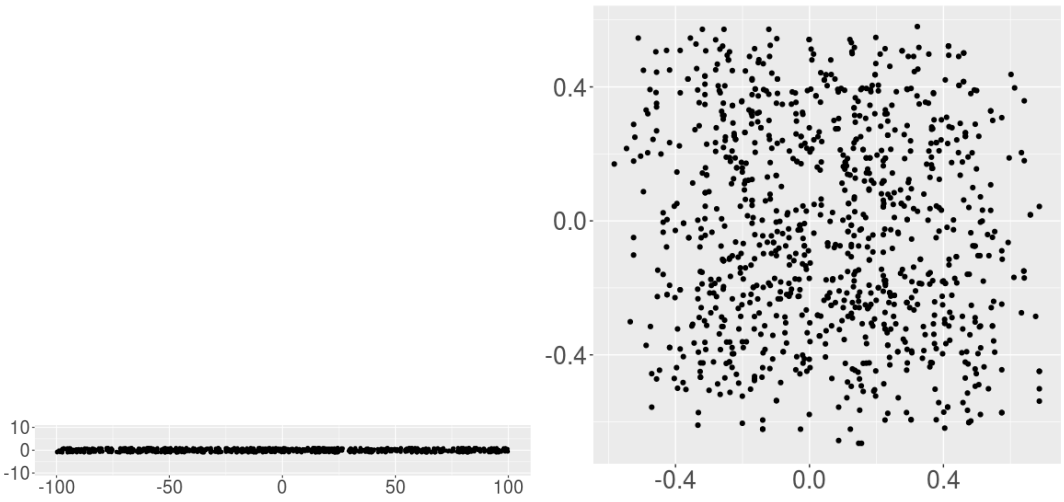

Figure 7: Left: the projected points of a skinny cube. Right: the projected points of the rounded skinny cube.

It returns a List that contains the rounded polytope, the matrix of the linear transformation, the vector/point which shifts input polytope P and the determinant of the linear map to be used for other volume computations. The following R script shows how useful rounding can be for volume computation and results to Figure 7 (plot scripts in the Appendix A.4).

```
d = 10
P = gen_skinny_cube(d)
points1 = sample_points(P, random_walk = list("walk" = "CDHR"), n = 1000,
                        seed =5)
P_rounded = round_polytope(P)$P
points2 = sample_points(P_rounded, random_walk = list("walk" = "CDHR"),
                        n = 1000, seed = 5)
cat(P$volume, volume(P, seed = 50), volume(P, rounding = TRUE, seed = 50))
```

```
102400 76324.25 99695.16
```

Notice that the accuracy is much better when we apply rounding as a preprocessing step before volume computation. The main reason behind this, is the better mixing time of all the random walks implemented in **volesti** for rounded bodies than for skinny ones.

When P is a Z-polytope the rounding step is computationally too costly for some cases, especially when the order is high. The use of the centrally symmetric H-polytope in MMC, reduces significantly the number of phases and the total running time. The following script show how one might select this option by declaring as `TRUE` the flag `"hpoly"`.

```
Z = gen_rand_zonotope(30, 35, generator = "uniform", seed = 250)
time1 = system.time({ vol1 = volume(Z, settings = list("hpoly" = FALSE),
                                    seed = 5) })
time2 = system.time({ vol2 = volume(Z, settings = list("hpoly" = TRUE),
```

Figure 8: Left: Uniform sampling with BiW from the 2D skinny cube $P := [-10, 10] \times [-100, 100]$. Right: We apply the inverse linear map to the sample obtained by BiW from the rounded polytope. We use walk length equal to one in both cases.

```
                                        seed = 5) })
cat(time1[3], time2[3])
```

```
501.224 69.792
```

```
cat(vol1, vol2)
```

```
1.434612e+57 1.133094e+57
```

Rounding is also particularly useful for sampling from skinny polytopes. One could round a skinny polytope P, sample from the rounded polytope and apply the inverse linear map to obtain a sample in P. The next R script demonstrates how useful this property can be even for 2D sampling and results to Figure 8 (plot scripts in the Appendix A.4).

```
d = 2
P = gen_skinny_cube(d)
points1 = sample_points(P, random_walk =
                          list("walk" = "BiW", "walk_length" = 1,
                               "starting_point" = c(50,0), "L" = 2*sqrt(d)),
                        n = 100, seed = 5)
ret_list = round_polytope(P, seed = 5)
P_rounded = ret_list$P
T = ret_list$T
shift = ret_list$shift
points2 = (T %*% sample_points(P_rounded,
                               random_walk = list("walk" = "BiW",
                                                  "walk_length" = 1),
                               n = 100, seed = 5)) +
          kronecker(matrix(1, 1, 100), matrix(shift, ncol = 1))
```

Notice that the sample points in the left plot of Figure 8 are concentrated around the starting point of the random walk. Furthermore, the sampling from the rounded polytope (right plot of Figure 8) results to much better convergence to the uniform distribution.

3.6. Random rotation and inscribed ball

The package provides more functions for the preprocessing of the input polytope.

The function `inner_ball()` takes as input a convex polytope and computes the inscribed ball of P as discussed in Section 3.3. The following script results to the left plot of Figure 9 (plot scripts in the Appendix A.5).

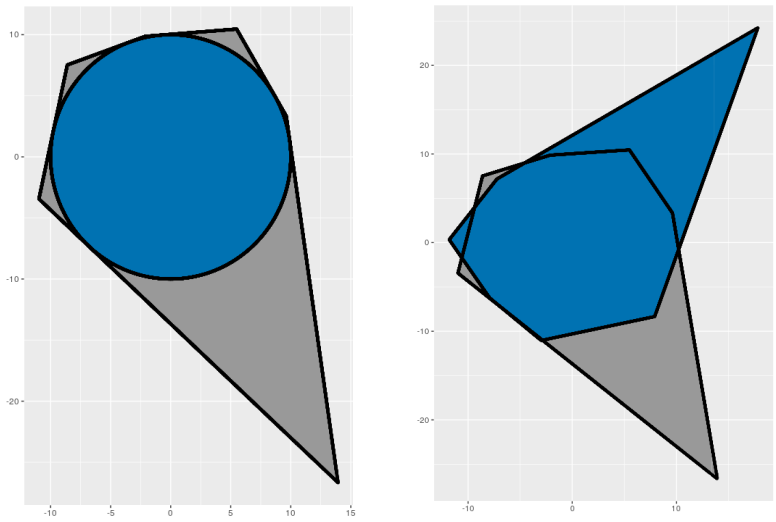

Figure 9: Left a H-polytope and the largest inscribed ball. Right the same H-polytope and with blue color a random rotation.

```
P1 = gen_rand_hpoly(2, 6, seed = 729)
points1 = sample_points(P1, random_walk = list("walk" = "BRDHR"), n = 10000,
                        seed = 5)
r = inner_ball(P1)[3]
points2 = direct_sampling(body = list("type" = "hypersphere",
                                      "dimension" = 2, "radius" = r), n = 10000,
                          seed = 5)
```

The function `rotate_polytope()` can be used to rotate a convex polytope P.

```
rotate_polytope(P = Rcpp_class, T = matrix, seed = integer)
```

It takes as input a polytope and a matrix T of a linear map and then apply the map on P. There is also the option to set a fixed seed for the linear map random generator. If it is given only a polytope then it generates a random linear map. It returns the rotated polytope and the linear transformation applied on input polytope P. The following script results to the right plot of Figure 9.

```
P1 = gen_rand_hpoly(2, 6, seed = 729)
points1 = sample_points(P1, random_walk = list("walk" = "BRDHR"), n = 10000,
                        seed = 5)
P2 = rotate_polytope(P1, seed = 12496)$P
points2 = sample_points(P2, random_walk = list("walk" = "BRDHR"), n = 10000,
                        seed = 5)
```

4. Applications

In this section we demonstrate **volesti**'s potential to solve challenging problems. More specifically, we provide detailed use-cases for applications in finance, mechanical engineering, multivariate integration, biology, and combinatorics.

4.1. Algorithmic tools in finance

In this subsection we present how one could employ **volesti** to detect financial crises in big stock markets. For all the examples in the sequel we use a set of 52 popular exchange traded funds (ETFs) and the US central bank (FED) rate of return publicly available from https://stanford.edu/class/ee103/portfolio.html. The following script is used to load the data.

```
MatReturns = read.table("https://stanford.edu/class/ee103/data/returns.txt",
                        sep = ",")
MatReturns = MatReturns[-c(1, 2), ]
dates = as.character(MatReturns$V1)
MatReturns = as.matrix(MatReturns[ ,-c(1, 54)])
MatReturns = matrix(as.numeric(MatReturns), nrow = dim(MatReturns )[1],
                    ncol = dim(MatReturns )[2], byrow = FALSE)
nassets = dim(MatReturns)[2]
```

For a specific set of assets in a stock market, portfolios are characterized by their return and their risk which is the variance of the portfolios' returns (volatility). Financial markets exhibit 3 types of behavior (Billio, Getmansky, and Pelizzon 2012). In normal times, portfolios are characterized by slightly positive returns and a moderate volatility, in up-market times (typically bubbles) by high returns and low volatility, and during financial crises by strongly negative returns and high volatility. These features motivate researchers to describe the time-varying dependency between portfolios' return and volatility. **volesti** relies on the work of Calès *et al.* (2018) and copula representation. A copula is an approximation of the bivariate joint distribution of portfolios' return/volatility. First, consider the set of portfolios in a stock market of d assets being the canonical simplex Δ^{d-1}. A vector of assets' returns $R \in \mathbb{R}^d$ defines a family of hyperplanes $R \cdot x = const.$ For a specific return (constant) R_0 the corresponding hyperplane intersecting Δ^{d-1} defines the set of portfolios with return R_0. The portfolios above this hyperplane have larger return than R_0 and those below smaller. **volesti** provides fast computations for the proportion of portfolios that lie in $R \cdot x \leq R_0,\ x \in \Delta^{d-1}$, which also called cross sectional score of portfolio, given a vector of assets' returns (Guegan *et al.* 2011).

The fast growth of asset management industry during the past few decades has highlighted the analysis of portfolio allocation performance as an important aspect of modern finance. The cross sectional score is an alternative to more classical choices for the evaluation of the performance of a portfolio as the Sharpe-like ratios proposed in the 1960's by Jensen (1967); Sharpe (1966); Treynor (2015). These ratios are provided by the `CRAN` package **PerformanceAnalytics** (Peterson and Carl 2020). However, they suffer from estimation errors as shown by Lo (2003), which prevent any performance comparison to be significant. The cross sectional score of a portfolio by Pouchkarev (2005); Banerjee and Hung (2011) is computed with uniform sampling from Δ^{d-1} and a rejection step. However, Calès *et al.* (2018) notice that

Varsi's algorithm, which is provided by function `frustum_of_simplex()`, performs robust computations of such volumes for d in the thousands in just a few milliseconds. Interestingly, the following R script let us know that in 03/13/2009 almost 48% of the portfolios had lower scoring than 0.002. We use uniform sampling from Δ^{d-1} for approximate computation and Varsi's algorithm for exact computation of the score.

```
N = 500000
R = MatReturns[which(dates %in% "2009-03-13"), ]
R0 = 0.002
tim1 = system.time({
          points = direct_sampling(n = N, seed = 5,
                                   body = list("type" ="canonical_simplex",
                                               "dimension" = nassets))
          vals = R %*% points
          points_in = 0
          for (i in 1:N) {
            if (vals[i] < R0){
              points_in = points_in + 1
            }
          }
          approximate_score = points_in / N
        })
tim2 = system.time({ exact_score = frustum_of_simplex(R, R0) })
cat(approximate_score, tim1[3], "\n", exact_score, tim2[3])
```

```
0.469328  1.44
0.4773961 0
```

The volatility of portfolios is represented by a family of concentric ellipsoids, centered at the origin, which matrix $\Sigma \in \mathbb{R}^{d\times d}$ is the covariance matrix of the distribution of the assets' returns. For a constant $v_0 \in \mathbb{R}^d$, the set of points $x^T\Sigma x = v_0,\ x \in \Delta^{d-1}$ is the set of portfolios with volatility v_0. The points above or below correspond to portfolios with larger or smaller volatility respectively. When M parallel hyperplanes and M concentric ellipsoids intersect with Δ^{d-1} they define a grid of $M \times M$ bodies. The copulas that are derived from the computation of all the proportions of portfolios that lie in each body in the grid is an approximation of the joint distribution between portfolios' return and volatility. In particular, we use compound returns which are computed on W consecutive rows of assets' returns and for the volatility we compute the covariance matrix of those returns. The function

```
copula(r1 = numeric vector, r2 = numeric vector, sigma = matrix,
       m = integer, n = integer, seed = integer)
```

can be used to compute such copulas. It takes as input (i) the vectors `r1, r2` that denote a family of parallel hyperplanes each; when both are given by the user then the computed copula is related to the problem of momentum effect in stock markets (for more details are given by

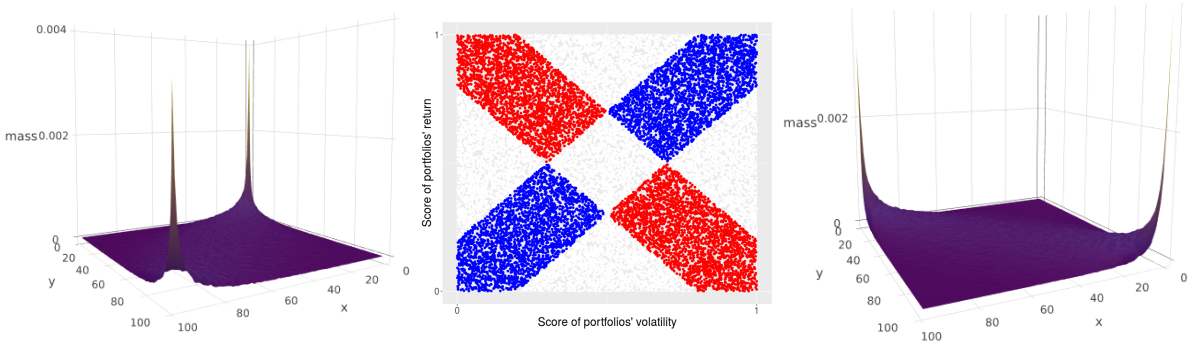

Figure 10: Left, a copula that corresponds to normal period (07/03/2007 − 31/05/2007), $I = 0.2316412$. Right, a copula that corresponds to a crisis period (18/12/2008−13/03/2009), $I = 5.610785$; x axis is for return and y axis is for volatility. The plot in the middle shows the mass of interest to characterize the market state.

Calès *et al.* (2018)), (ii) the matrix `sigma` that denotes a family of concentric ellipsoids, (iv) the integer `m` that denotes the number of objects for each family and (v) the integer `n` for the number of uniformly distributed points to generate in Δ^{d-1}. The method counts the number of points in each body of the grid to obtain a copula.

The following script results to Figure 10 by setting the starting and the stopping date for the left and the right plot respectively (code of the additional function `get_compound_returns()` in Appendix A.6). We computed two copulas; one in normal times (left plot) and a second in crisis period (right plot), by using the compound return and an estimation of the covariance matrix of $W = 60$ consecutive days of assets' returns. Notice that the mass of a copula concentrates in a specific diagonal depending on the market state.

```
row1 = which(dates %in% "2008-12-18")
row2 = which(dates %in% "2009-03-13")
cR = get_compound_return(MatReturns[row1:row2, ])
mass = copula(r1 = cR, sigma = cov(MatReturns[row1:row2, ]),
              m = 100, n = 1e+06, seed = 5)
plot_ly(z = ~mass) %>% add_surface(showscale=FALSE)
```

In particular, we can derive information by considering the mass on the two main diagonals of such a copula as the plot in the middle in Figure 10 illustrates. We define an indicator as the ratio between two masses (red / blue). If the indicator is ≥ 1 then the copula corresponds to a crises, otherwise corresponds to a normal period.

```
compute_indicators(returns =  matrix, win_len = integer, m = integer,
                   n = integer, nwarning = integer, ncrisis = integer,
                   seed = integer)
```

The function `compute_indicators()` takes as input (i) a set of assets' returns as a matrix, (ii) the length, `win_len` (or W), of the sliding window, (iii) the number of objects, `m`, for each family of hyperplanes or ellipsoids, (iv) the number of points `n` to sample. The input variable

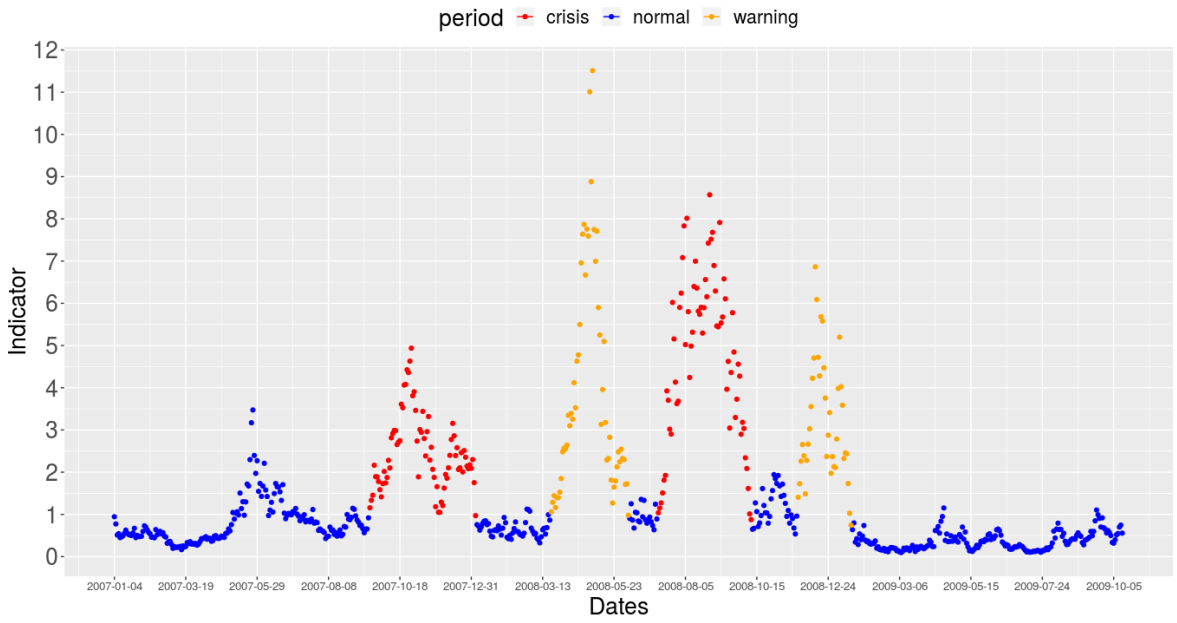

Figure 11: The values of the indicators from 2007-01-04 until 2010-01-04. We mark with red the crisis periods, with orange the warning periods and with blue the normal periods that **volesti** identifies.

`nwarning` implies the number of consecutive indicators larger than 1 to declare a warning and the variable `ncrisis` the number of consecutive indicators larger than 1 to declare a crisis. The function computes the copulas of all the sets of W consecutive days and returns the corresponding indicators and the states of the market during the given time period.

The following R script takes as input the daily returns of all the 52 assets from 01/04/2007 until 04/01/2010. When the indicator is ≥ 1 for more than 30 days we issue a warning and when it is for more than 60 days we mark this period as a crisis.

```
row1 = which(dates %in% "2007-01-04")
row2 = which(dates %in% "2010-01-04")
market_analysis = compute_indicators(returns =  MatReturns[row1:row2, ],
                                     win_len = 60, m = 100, n = 1e+06,
                                     nwarning = 30, ncrisis = 60, seed = 5)
I = market_analysis$indicators
market_states = market_analysis$market_states
```

We compare the results with the database for financial crises in European countries proposed in (Duca, Koban, Basten, Bengtsson, Klaus, Kusmierczyk, Lang, Detken, and Peltonen 2017). The only listed crisis for this period is the sub-prime crisis (from December 2007 to June 2009). Notice that Figure 11 (plot scripts in the Appendix A.7) successfully points out 4 crisis events in that period (2 crisis and 2 warning periods) and detects sub-prime crisis as a W-shape crisis.

4.2. Evaluating Z-polytope approximation methods

Volume approximation for Z-polytopes (or zonotopes) is very useful in several applications in decision and control (Kopetzki, Schürmann, and Althoff 2017), in autonomous driving (Althoff and Dolan 2014) or human-robot collaboration (Pereira and Althoff 2015). The complexity of algorithms that work on Z-polytopes strongly depends on their order. Thus, to achieve efficient computations a solution that is common in practice is to over-approximate P, as tight as possible, with a second Z-polytope P_{red} of smaller order, while $\text{vol}(P_{red})$ is given by, an easy to compute, closed formula. A good measure for the quality of the approximation is the following ratio of fitness,

$$\rho = \left(\frac{\text{vol}(P_{red})}{\text{vol}(P)}\right)^{1/d}. \tag{2}$$

This involves a volume computation problem. In (Kopetzki *et al.* 2017) they use exact - deterministic volume computation and thus cannot compute the quality of the approximation for $d > 10$. **volesti** is the first software that provides efficient volume estimation for Z-polytopes such that it is possible to evaluate an over approximation of a high dimensional Z-polytope P or a Z-polytope of very high order in lower dimensions. Moreover, the function,

```
zonotope_approximation(Z = Rcpp_class, fit_ratio = boolean,
                       settings = List, seed = integer)
```

provides both an over-approximation P_{red} of a given zonotope P using PCA method and an evaluation of P_{red} estimating the ratio of fitness, ρ. The `List settings` can be used to set the parameters of the CB algorithm as in Section 3.3, while the `boolean fit_ratio` can be used to request the computation of ρ. In the following pseudocode of PCA method the $IH(\cdot)$ is the interval hull by Kühn (1998).

PCA (Z-polytope P with generators' matrix $G \in \mathbb{R}^{d\times k}$)
$X = [G| - G]^T$
$USV^T = \text{SVD}(X^T X)$
Return: $G_{red} = U \cdot IH(U^T G)$

Notice that $G_{red} = U \cdot IH(U^T G)$ defines a box and generates P_{red} and thus $\text{vol}(P_{red})$ is given by a closed formula and its H-representation can be easily derived. The following R script, generates a random 2D zonotope and computes the over-approximation with PCA method; it results to Figure 12 (plot scripts in the Appendix A.8).

```
Z = gen_rand_zonotope(2, 8, generator = "uniform", seed = 1729)
points1 = sample_points(Z, random_walk = list("walk" = "BRDHR"), n = 10000)
retList = zonotope_approximation(Z = Z, fit_ratio = TRUE, seed = 5)
P = retList$P
points2 = sample_points(P, random_walk = list("walk" = "BRDHR"), n = 10000,
                        seed = 5)
cat(retList$fit_ratio)
```

```
1.116799539
```

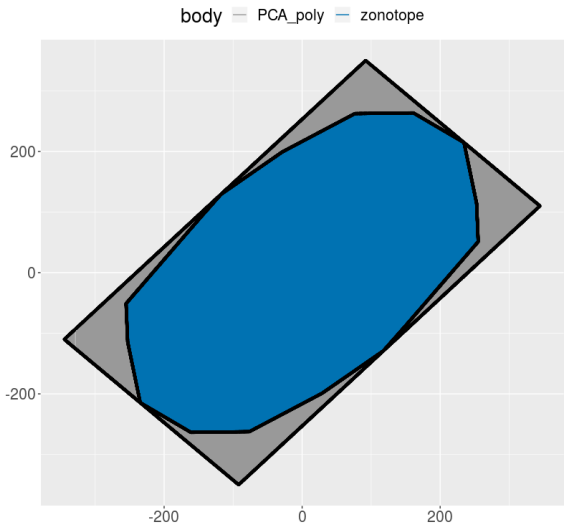

Figure 12: With blue color a 2D Z-polytope. With grey color the over-approximation of P computed with PCA method.

The next R script displays an example in $d = 30$ for which exact volume computation is not possible. In particular, computing ρ using exact volume computation would take years in an ordinary PC. Inevitably, Kopetzki *et al.* (2017) report values of ρ with almost 100% error as they replace, in Equation (2), the volume of P with the volume of the over-approximation computed by other methods (e.g. BOX method).

```
Z = gen_rand_zonotope(20, 500, generator = "uniform", seed = 127)
retList = zonotope_approximation(Z = Z, fit_ratio = TRUE, seed = 5)
cat(retList$fit_ratio)
```

```
2.454694
```

4.3. Approximating multidimensional integrals

Computing the integral of a function over a convex set (i.e. convex polytope) is a hard fundamental problem with numerous applications. The only R packages that support such computations are **SimplicialCubature** by Nolan and Genz (2016) and **cubature** by Narasimhan, Koller, Johnson, Hahn, Bouvier, Kiêu, and Gaure (2019). The first computes multivariate integrals over simplices and the second over hypercubes. **cubature** is a R wrapper of C packages **cuba** by Hahn (2018) and **cubature** by Johnson (2018).

Hence, to compute an integral of a function over a convex polytope P in R one should compute the Delaunay triangulation with package **geometry** and then use the package **SimplicialCubature** to sum the values of all the integrals over the simplices computed by the triangulation. On the other hand **volesti** can be used to approximate the value of such an integral by a simple MCMC integration method, which employs the vol(P) and a uniform sample from P. In particular, let

$$I = \int_P f(x)dx. \tag{3}$$

Then sample N uniformly distributed points $x_1, \ldots, x_N$ from P and,

$$I \approx \text{vol}(P)\frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{4}$$

The following R script generates a V-polytope for $d = 5, 10, 15, 20$ and defines a function f. Then computes the exact value of I and in the sequel approximates that value employing **volesti**. The pattern is similar to volume computation, for $d = 5, 10$ the exact computation is faster than the approximate, for $d = 15$ **volesti** is 13 times faster and for $d = 20$ the exact approach halts while **volesti** returns an estimation in less than a minute.

```
for (d in seq(from = 5, to = 20, by = 5)) {
  P = gen_rand_vpoly(d, 2 * d, seed = 127)
  tim1 = system.time({
    triang = geometry::delaunayn(P$V)
    f = function(x) { sum(x^2) + (2 * x[1]^2 + x[2] + x[3]) }
    I1 = 0
    for (i in 1:dim(triang)[1]) {
      I1 = I1 + SimplicialCubature::adaptIntegrateSimplex(f,
                t(P$V[triang[i,], ]))$integral
    }
  })
  tim2 = system.time({
    num_of_points = 5000
    points = sample_points(P, random_walk = list("walk" = "BiW",
                                                 "walk_length" = 1),
                           n = num_of_points, seed = 5)
    int = 0
    for (i in 1:num_of_points){
      int = int + f(points[, i])
    }
    V = volume(P, settings = list("error" = 0.05), seed = 5)
    I2 = (int * V) / num_of_points
  })
  cat(d, I1, I2, abs(I1 - I2) / I1, tim1[3], tim2[3], "\n")
}
```

```
5   0.02738404    0.02597928    0.05129854  0.41     3.095
10  3.224286e-06  2.935246e-06  0.08964482  2.945    12.01
15  4.504834e-11  4.574285e-11  0.01541695  471.479  33.256
20  error         9.947623e-17  Inf         -        64.058
```

4.4. Counting linear extensions

Let $G = (V, E)$ be an acyclic digraph with $V = [n] := \{1, 2, \ldots, n\}$. One might want to consider G as a representation of the partially ordered set (poset) $V : i > j$ if and only if

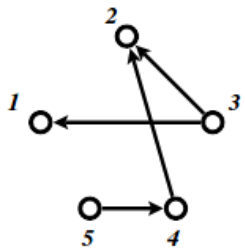

Figure 13: An acyclic directed graph with 5 nodes, 4 edges and 9 linear extensions.

there is a directed path from node i to node j. A permutation π of $[n]$ is called a linear extension of G (or the associated poset V) if $\pi^{-1}(i) > \pi^{-1}(j)$ for every edge $i \to j \in E$.
Let $P_{LE}(G)$ be the polytope in $\mathbb{R}^n$ defined by

$$P_{LE}(G) = \{x \in \mathbb{R}^n \mid 1 \geq x_i \geq 0 \text{ for all } i = 1, 2, \ldots, n\},$$

and $x_i \geq x_j$ for all directed edges $i \to j \in E$.
It is well known (Stanley 1986) that the number of linear extensions of G equals the normalized volume of $P_{LE}(G)$ i.e.,

$$\#_{LE}G = \text{vol}(P_{LE}(G))\ n!$$

The graph in Figure 13 has 9 linear extensions[7]. This number can be estimated using **volesti** as in the following script.

```
A = matrix(c(
  -1,0,1,0,0,0,
  -1,1,0,0,0,-1,
  0,1,0,0,0,0,-1,
  1,1,0,0,0,0,0,
  1,0,0,0,0,0,1,
  0,0,0,0,0,1,0,
  0,0,0,0,1,-1,
  0,0,0,0,0,-1,
  0,0,0,0,0,-1,
  0,0,0,0,0,-1,
  0,0,0,0,0,-1),
ncol = 5, nrow = 14, byrow = TRUE)
b = c(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 , 1, 1, 1)
P_LE = Hpolytope$new(A, b)
cat(volume(P_LE, settings = list("error" = 0.2), seed = 5) * factorial(5))
```

```
8.456771
```

As the number of nodes (i.e., the dimension of $P_{LE}(G)$) grows the problem becomes intractable for exact methods while **volesti** provides an efficient alternative. There is a recent

[7]Example taken from https://inf.ethz.ch/personal/fukudak/lect/pclect/notes2016/expoly_order.pdf

interest on the practical aspects of counting linear extensions from the perspective of artificial intelligence (Talvitie, Kangas, Niinimäki, and Koivisto 2018).

4.5. Intersections of V-polytopes

Challenging volume computations appear in biogeography research and in particular in biodiversity analysis of bird species Barnagaud, Kissling, Tsirogiannis, Fisikopoulos, Villeger, Sekercioglu, and Svenning (2017). This computation involves volume calculations for intersections of V-polytopes. Our package, **volesti**, provides an R class to represent such convex bodies and an option to sample from or to estimate their volume. While up to dimension 5 **geometry** is clearly faster than **volesti** in volume computation, starting in dimension 6 **volesti** computes faster estimates and this is expected to be the rule as the dimension grows.

```
d = 6
P1 = gen_rand_vpoly(d, 14, seed=7)
P2 = gen_rand_vpoly(d, 15, seed=4)
VP = VpolytopeIntersection$new(P1$V, P2$V)

time1 = system.time({ exact_volume = geometry::intersectn(P1$V, P2$V)$ch$vol })
time2 = system.time({ approximate_volume = volume(VP, seed = 50) })

cat(exact_volume, approximate_volume, time1[3], time2[3])
```

```
0.005736105 0.005579085 13.711 3.089
```

5. Conclusion

The R package, **volesti**, is a key to guarantee that our software is accessible from scientific or business communities that are not familiar with programming in C++ and need a friendly environment to use all these statistical and geometrical tools we provide.

Computational details

The results in this paper were obtained using R 3.4.4 and **volesti** 1.1.1. The versions of the imported by **volesti** packages are **stats** 3.4.4 and **methods** 3.4.4; of the linked by **volesti** packages, **Rcpp** 1.0.3, **BH** 1.69.0.1, **RcppEigen** 0.3.3.7.0, and the suggested package **testthat** 2.0.1. For comparison to **volesti** and for plots this paper uses, **geometry** 0.4.5, **hitandrun** 0.5.5, **SimplicialCubature** 1.2, **ggplot2** 3.1.0, **plotly** 4.8.0, **latex2exp** 0.4.0, **rgl** 0.100.50. All packages used are available from CRAN at http://CRAN.R-project.org.

All experiments were performed on a PC with Intel® Pentium(R) CPU G4400 @ 3.30GHz × 2 CPU and 16GB RAM.

Acknowledgments

The main part of the work has been done while A.C. was supported by Google Summer of Code[8] 2018 and 2019 grants and V.F. was his mentor. The authors acknowledge fruitful discussions with Ioannis Emiris, Elias Tsigaridas, the R-project for statistical computing and the R community.

References

Afshar HM, Domke J (2015). "Reflection, Refraction, and Hamiltonian Monte Carlo." In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pp. 3007–3015. MIT Press, Cambridge, MA, USA. URL https://dl.acm.org/doi/10.5555/2969442.2969575.

Althoff M, Dolan JM (2014). "Online Verification of Automated Road Vehicles Using Reachability Analysis." *IEEE Transactions on Robotics*, **30**(4), 903–918. ISSN 1552-3098. doi:10.1109/TRO.2014.2312453.

Banerjee A, Hung CH (2011). "Informed Momentum Trading Versus Uninformed "Naive" Investors Strategies." *Journal of Banking & Finance*, **35**(11), 3077–3089. ISSN 0378-4266. doi:https://doi.org/10.1016/j.jbankfin.2011.04.005.

Barber CB, Dobkin DP, Huhdanpaa H (1996). "The Quickhull Algorithm for Convex Hulls." *ACM Transactions on Mathematical Software*, **22**(4), 469–483. ISSN 0098-3500. doi:10.1145/235815.235821.

Barnagaud J, Kissling W, Tsirogiannis C, Fisikopoulos V, Villeger S, Sekercioglu C, Svenning J (2017). "Biogeographic, Environmental and Anthropogenic Determinants of Global Patterns in Functional and Taxonomic Turnover in Birds." *Global Ecology and Biogeography*, **26**, 1190–1200. doi:https://doi.org/10.1111/geb.12629.

Berkelaar M, Eikland K, Notebaert P (2004). "**lp_solve** 5.5, Open Source (Mixed-Integer) Linear Programming System." Software. URL http://lpsolve.sourceforge.net/5.5/.

Betancourt M (2017). "A Conceptual Introduction to Hamiltonian Monte Carlo." https://arxiv.org/pdf/1701.02434.pdf.

Billio M, Getmansky M, Pelizzon L (2012). "Dynamic Risk Exposures in Hedge Funds." *Computational Statistics & Data Analysis*, **56**(11), 3517–3532. ISSN 0167-9473. doi:https://doi.org/10.1016/j.csda.2010.08.015.

Boyd S, Vandenberghe L (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA.

Calès L, Chalkis A, Emiris I, Fisikopoulos V (2018). "Practical Volume Computation of Structured Convex Bodies, and an Application to Modeling Portfolio Dependencies and Financial Crises." In B Speckmann, CD Tóth (eds.), *34th International Symposium on Computational*

[8] https://summerofcode.withgoogle.com

Geometry (SoCG 2018), volume 99 of *LIPIcs*, pp. 19:1–19:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. doi:10.4230/LIPIcs.SoCG.2018.19.

Cerny M (2012). "Goffin's Algorithm for Zonotopes." *Kybernetika*, **48**(5), 890–906. URL http://eudml.org/doc/251418.

Chalkis A, Emiris IZ, Fisikopoulos V (2019). "Practical Volume Estimation by a new Annealing Schedule for Cooling Convex Bodies." URL http://arxiv.org/abs/1905.05494.

Chen Y, Dwivedi R, Wainwright M, Yu B (2018). "Fast MCMC Sampling Algorithms on Polytopes." *Journal of Machine Learning Research*, **19**(55), 1–86. URL http://jmlr.org/papers/v19/18-158.html.

Cousins B, Vempala S (2015). "Bypassing KLS: Gaussian Cooling and an $O^*(n^3)$ Volume Algorithm." In *The Annual ACM Symposium on Theory of Computing*, pp. 539–548. doi:10.1145/2746539.2746563.

Cousins B, Vempala S (2016). "A Practical Volume Algorithm." *Mathematical Programming Computation*, **8**(2). doi:10.1007/s12532-015-0097-z.

Dabbene F, Shcherbakov PS, Polyak BT (2010). "A Randomized Cutting Plane Method with Probabilistic Geometric Convergence." *SIAM Journal on Optimization*, **20**(6), 3185–3207. ISSN 1052-6234. doi:10.1137/080742506.

Dalalyan AS (2017). "Theoretical Guarantees for Approximate Sampling from a Smooth and Log-Concave Density." *Journal of the Royal Statistical Society: Series B*, **79**, 651–676. doi:10.1111/rssb.12183. URL http://arxiv.org/pdf/1412.7392v3.pdf.

Dieker AB, Vempala SS (2015). "Stochastic Billiards for Sampling from the Boundary of a Convex Set." *Mathematics of Operations Research*, **40**(4), 888–901. doi:10.1287/moor.2014.0701.

Duane S, Kennedy A, Pendleton BJ, Roweth D (1987). "Hybrid Monte Carlo." *Physics Letters B*, **195**(2), 216 – 222. ISSN 0370-2693. doi:https://doi.org/10.1016/0370-2693(87)91197-X.

Duca M, Koban A, Basten M, Bengtsson E, Klaus B, Kusmierczyk P, Lang J, Detken C, Peltonen T (2017). "A new Database for Financial Crises in European Countries." *Occasional Paper Series 194*, European Central Bank. URL https://ideas.repec.org/p/ecb/ecbops/2017194.html.

Dyer M, Frieze A (1988). "On the Complexity of Computing the Volume of a Polyhedron." *SIAM Journal on Computing*, **17**(5), 967–974. doi:10.1137/0217060.

Dyer M, Frieze A, Kannan R (1991). "A Random Polynomial-Time Algorithm for Approximating the Volume of Convex Bodies." *Journal of the ACM*, **38**(1), 1–17. URL http://doi.acm.org/10.1145/102782.102783.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08. URL http://www.jstatsoft.org/v40/i08/.

Eddelbuettel D, François R (2017). "Exposing C++ Functions and Classes With **Rcpp** Modules." ***Rcpp*** *version 0.12.10 as of March 17, 2017*.

Elekes G (1986). "A Geometric Inequality and the Complexity of Computing Volume." *Discrete and Computational Geometry*, **1**(4), 289–292. ISSN 1432-0444. doi:10.1007/BF02187701.

Emiris I, Fisikopoulos V (2014). "Practical Polytope Volume Approximation." *ACM Transactions of Mathematical Software, 2018*, **44**(4), 38:1–38:21. ISSN 0098-3500. doi:10.1145/3194656. Conf. version: Proc. Symp. Comp. Geometry, 2014.

Fisikopoulos V, Chalkis A (2019). ***volesti***: *Volume Approximation and Sampling of Convex Polytopes in R*. R package version 1.1.0, URL https://CRAN.R-project.org/package=volesti.

Geman S, Geman D (1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-6**(6), 721–741. ISSN 1939-3539. doi:10.1109/TPAMI.1984.4767596.

Genz A, Bretz F (2009). *Computation of Multivariate Normal and t Probabilities*. 1st edition. Springer Publishing Company, Incorporated. ISBN 364201688X, 9783642016882.

Gover E, Krikorian N (2010). "Determinants and the Volumes of Parallelotopes and Zonotopes." *Linear Algebra and its Applications*, **433**(1), 28 – 40. ISSN 0024-3795. doi:https://doi.org/10.1016/j.laa.2010.01.031.

Guegan D, Calès L, Billio M (2011). "A Cross-Sectional Score for the Relative Performance of an Allocation." *Université Paris 1 Panthéon-Sorbonne (Post-Print and Working Papers) halshs-00646070*, HAL. URL https://ideas.repec.org/p/hal/cesptp/halshs-00646070.html.

Guennebaud G, Jacob B, *et al.* (2010). ***Eigen*** *v3*. URL http://eigen.tuxfamily.org.

Hahn T (2018). ***Cuba*** *- A Library for Multidimensional Numerical Integration*. Max Planck Institute for Physics, Munich. URL http://www.feynarts.de/cuba/.

Haraldsdòttir H, Cousins B, Thiele I, Fleming R, Vempala S (2017). "CHRR: Coordinate Hit-and-Run with Rounding for Uniform Sampling of Constraint-Based Models." *Bioinformatics*, **33**(11), 1741–1743. doi:10.1093/bioinformatics/btx052.

Hastings WK (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications." *Biometrika*, **57**(1), 97–109. ISSN 00063444. URL http://www.jstor.org/stable/2334940.

Herrmann H, Dyson B, Vass L, Johnson G, Schwartz J (2019). "Flux Sampling is a Powerful Tool to Study Metabolism Under Changing Environmental Conditions." *Systems Biology and Applications*, **5**(32). doi:10.1038/s41540-019-0109-0.

Iyengar S (1988). "Evaluation of Normal Probabilities of Symmetric Regions." *SIAM Journal on Scientific and Statistical Computing*, **9**(3), 418–423. doi:https://doi.org/10.1137/0909028.

Jensen MC (1967). "The Performance of Mutual Funds in the Period 1945-1964." *SSRN Scholarly Paper ID 244153*, Social Science Research Network, Rochester, NY. URL https://papers.ssrn.com/abstract=244153.

Johnson SG (2018). ***Cubature*** - *Multi-Dimensional Adaptive Integration (Cubature) in C*. Massachusetts Institute of Technology. URL https://github.com/stevengj/cubature.

Kaufman DE, Smith RL (1998). "Direction Choice for Accelerated Convergence in Hit-and-Run Sampling." *Operations Research*, **46**(1), 84–95. doi:10.1287/opre.46.1.84.

Kopetzki A, Schürmann B, Althoff M (2017). "Methods for Order Reduction of Zonotopes." In *IEEE Conference on Decision and Control*, pp. 5626–5633. doi:10.1109/CDC.2017.8264508.

Kühn W (1998). "Rigorously Computed Orbits of Dynamical Systems Without the Wrapping Effect." *Computing*, **61**, 47–67. doi:https://doi.org/10.1007/BF02684450.

Laddha A, Lee YT, Vempala S (2019). "Strong Self-Concordance and Sampling." https://arxiv.org/abs/1911.05656.

Lee Y, Vempala S (2017). "Eldan's Stochastic Localization and the KLS Hyperplane Conjecture: An Improved Lower Bound for Expansion." In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pp. 998–1007. doi:10.1109/FOCS.2017.96.

Lee Y, Vempala S (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*, STOC 2018, pp. 1115–1121. ISBN 978-1-4503-5559-9. doi:10.1145/3188745.3188774.

Lee YT, Shen R, Tian K (2020). "Logsmooth Gradient Concentration and Tighter Runtimes for Metropolized Hamiltonian Monte Carlo." 2002.04121.

Lee YT, Song Z, Vempala SS (2018). "Algorithmic Theory of ODEs and Sampling from Well-conditioned Logconcave Densities." https://arxiv.org/abs/1812.06243.

Lo A (2003). "The Statistics of Sharpe Ratios." *Financial Analysts Journal*, **58**. doi:10.2469/faj.v58.n4.2453.

Lovász L, Kannan R, Simonovits M (1997). "Random Walks and an $O^*(n^5)$ Volume Algorithm for Convex Bodies." *Random Structures and Algorithms*, **11**, 1–50. doi:10.1002/(SICI)1098-2418(199708)11:1<1::AID-RSA1>3.0.CO;2-X.

Lovász L, Vempala S (2006). "Hit-and-Run from a Corner." *SIAM Journal on Computing*, **35**(4), 985–1005. ISSN 0097-5397. doi:10.1137/S009753970544727X.

Lovász L, Vempala S (2006). "Simulated Annealing in Convex Bodies and an $O^*(n^4)$ Volume Algorithm." *Journal of Computer and System Sciences*, **72**, 392–417. doi:https://doi.org/10.1016/j.jcss.2005.08.004.

Mangoubi O, Smith A (2017). "Rapid Mixing of Hamiltonian Monte Carlo on Strongly Log-Concave Distributions." 1708.07114.

Mangoubi O, Vishnoi NK (2019). "Faster Polytope Rounding, Sampling, and Volume Computation via a Sub-Linear Ball Walk." In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 1338–1357. doi:doi:10.1109/FOCS.2019.00082.

Maurer J, Watanabe S (2017). "Boost Random Number Library." Software. URL https://www.boost.org/doc/libs/1_73_0/doc/html/boost_random.html.

Megchelenbrink W, Huynen M, Marchiori E (2014). "optGpSampler: An Improved Tool for Uniformly Sampling the Solution-Space of Genome-Scale Metabolic Networks." *PLOS ONE*, **9**(2), 1–8. doi:10.1371/journal.pone.0086587.

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). "Equation of State Calculations by Fast Computing Machines." *The Journal of Chemical Physics*, **21**(6), 1087–1092. doi:10.1063/1.1699114.

Murty K (2009). "Ball Centers of Special Polytopes." *Dept Industrial & Operations Engineering, U. Michigan*. URL http://www-personal.umich.edu/~murty/Ballcenter2.pdf.

Narasimhan B, Koller M, Johnson SG, Hahn T, Bouvier A, Kiêu K, Gaure S (2019). ***cubature**: Adaptive Multivariate Integration over Hypercubes*. R package version 2.0.4, URL https://CRAN.R-project.org/package=cubature.

Neal RM (2011). *MCMC Using Hamiltonian Dynamics*, chapter 5. CRC Press. doi:10.1201/b10905-7.

Nikolic B (2015). ***BNMin1**: A Minimisation Library*. C++ package version 1.11, URL https://www.mrao.cam.ac.uk/~bn204/oof/bnmin1.html.

Nolan JP, Genz A (2016). ***SimplicialCubature**: Integration of Functions over Simplices*. R package version 1.2, URL https://CRAN.R-project.org/package=SimplicialCubature.

Pereira A, Althoff M (2015). "Safety Control of Robots Under Computed Torque Control Using Reachable Sets." In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 331–338. ISSN 1050-4729. doi:10.1109/ICRA.2015.7139020.

Peterson BG, Carl P (2020). ***PerformanceAnalytics**: Econometric Tools for Performance and Risk Analysis*. R package version 2.0.4, URL https://CRAN.R-project.org/package=PerformanceAnalytics.

Polyak B, Gryazina E (2014). "Billiard Walk - a new Sampling Algorithm for Control and Optimization." *IFAC Proceedings Volumes*, **47**(3), 6123 – 6128. ISSN 1474-6670. doi:https://doi.org/10.3182/20140824-6-ZA-1003.02312. 19th IFAC World Congress.

Pouchkarev I (2005). *Performance Evaluation of Constrained Portfolios*. Ph.D. thesis, Erasmus Research Institute of Management, The Netherlands.

Rubinstein R, Melamed B (1998). *Modern Simulation and Modeling*. Wiley, New York.

Saa PA, Nielsen LK (2016). "ll-ACHRB: A Scalable Algorithm for Sampling the Feasible Solution Space of Metabolic Networks." *Bioinformatics*, **32**(15), 2330–2337. ISSN 1367-4803. doi:10.1093/bioinformatics/btw132.

Schellenberger J, Palsson B (2009). "Use of Randomized Sampling for Analysis of Metabolic Networks." *The Journal of biological Chemistry*, **284 9**, 5457–61. doi:10.1074/jbc.R800048200.

Schneebeli C (2015). "Randomized Algorithms to Sample from the Boundary of Convex Polytopes." Department of Mathematics, ETH Zurich. Bachelor's Thesis.

Sharpe WF (1966). "Mutual Fund Performance." *The Journal of Business*, **39**(1), 119–138. ISSN 0021-9398. URL https://www.jstor.org/stable/2351741.

Shen R, Lee YT (2019). "The Randomized Midpoint Method for Log-Concave Sampling." https://arxiv.org/abs/1909.05503.

Smith RL (1984). "Efficient Monte Carlo Procedures for Generating Points Uniformly Distributed Over Bounded Regions." *Operations Research*, **32**(6), 1296–1308. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/170949.

Somerville P (1998). "Numerical Computation of Multivariate Normal and Multivariate-t Probabilities over Convex Regions." *Journal of Computational and Graphical Statistics*, **7**(4), 529–544. doi:10.1080/10618600.1998.10474793.

Stanley RP (1986). "Two Poset Polytopes." *Discrete & Computational Geometry*, **1**(1), 9–23. ISSN 1432-0444. doi:10.1007/BF02187680.

Talvitie T, Kangas K, Niinimäki T, Koivisto M (2018). "Counting Linear Extensions in Practice: MCMC Versus Exponential Monte Carlo." In *AAAI Conference on Artificial Intelligence*. URL https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16957.

Todd MJ, Yildirim EA (2007). "On Khachiyan's Algorithm for the Computation of Minimum-Volume Enclosing Ellipsoids." *Discrete Applied Mathematics*, **155**(13), 1731 – 1744. ISSN 0166-218X. doi:https://doi.org/10.1016/j.dam.2007.02.013.

Treynor JL (2015). "How to Rate Management of Investment Funds." In *Treynor on Institutional Investing*, pp. 69–87. John Wiley & Sons, Ltd. ISBN 978-1-119-19667-9. doi:10.1002/9781119196679.ch10.

van Valkenhoef G, Tervonen T (2019). ***hitandrun**: "Hit and Run" and "Shake and Bake" for Sampling Uniformly from Convex Shapes*. R package version 0.5-5, URL https://CRAN.R-project.org/package=hitandrun.

Varsi G (1973). "The Multidimensional Content of the Frustum of the Simplex." *Pacific Journal of Mathematics*, **46**(1), 303–314. URL https://projecteuclid.org:443/euclid.pjm/1102946623.

Venzke A, Molzahn D, Chatzivasileiadis S (2019). "Efficient Creation of Datasets for Data-Driven Power System Applications." URL https://arxiv.org/abs/1910.01794.

Ziegler G (1995). *Lectures on Polytopes*. Springer-Verlag, New York. URL https://doi.org/10.1007/978-1-4613-8431-1.

A. R scripts for plotting

A.1. Random walks on hypercubes

The following script was used to generate Figure 4. The generated sample points are stored in files parameterized by walk length i. For example, `BaW_2.png` stores the points for Ball walk with walk length 2.

```
open3d()
plot3d(x=points1[1,], y=points1[2,], z=points1[3,], xlim = c(-1,1),
       ylim = c(-1,1), zlim = c(-1,1), col = "cyan", size=2, main = "",
       sub = "", ann = FALSE, axes = FALSE, xlab="", ylab="", zlab="")
box3d()
rgl.snapshot( filename=paste0("BaW_",i,".png"), fmt = "png", top = TRUE )
rgl.close()

open3d()
plot3d(x=points2[1,], y=points2[2,], z=points2[3,], xlim = c(-1,1),
       ylim = c(-1,1), zlim = c(-1,1), col = "black", size=2, main = "",
       sub = "", ann = FALSE, axes = FALSE, xlab="", ylab="", zlab="")
box3d()
rgl.snapshot( filename=paste0("CDHR_",i,".png"), fmt = "png", top = TRUE )
rgl.close()

open3d()
plot3d(x=points3[1,], y=points3[2,], z=points3[3,], xlim = c(-1,1),
       ylim = c(-1,1), zlim = c(-1,1), col = "red", size=2, main = "",
       sub = "", ann = FALSE, axes = FALSE, xlab="", ylab="", zlab="")
box3d()
rgl.snapshot( filename=paste0("RDHR_",i,".png"), fmt = "png", top = TRUE )
rgl.close()

open3d()
plot3d(x=points4[1,], y=points4[2,], z=points4[3,], xlim = c(-1,1),
       ylim = c(-1,1), zlim = c(-1,1), col = "blue", size=2, main = "",
       sub = "", ann = FALSE, axes = FALSE, xlab="", ylab="", zlab="")
box3d()
rgl.snapshot( filename=paste0("BiW_",i,".png"), fmt = "png", top = TRUE )
rgl.close()
```

A.2. Direct sampling

The following script can be used to generate Figure 5.

```
## left plot
p.data <- data.frame(
       x = c(points1[1,],points1[1,]),
```

```
       y = c(points1[2,],points1[2,]),
       z = c(points1[3,],points1[3,])
   )
plot_ly(p.data, x = ~x, y = ~y, z = ~z,marker=list(size=1),
colors = c('#BF382A'))

## right plot
ggplot(data.frame( x=c(points2[1,], points3[1,]), y=c(points2[2,], points3[2,]),
       region = c(rep("interior",N), rep("boundary",N)) ), aes(x=x, y=y)) +
       geom_point(aes(color = region)) + labs(x="",y="") +
       theme(legend.position="top",text = element_text(size=20),
       axis.text.x = element_text(size=10), axis.text.y = element_text(size=10))
```

A.3. Intersections of polytopes

The following script can be used to generate Figure 6.

```
## left plot
ggplot(data.frame(y = c(points1[1,],points2[1,]), body = c(rep("P1",10000),
       rep("P2",10000)), x = c(points1[2,],points2[2,])) , aes(x=x, y=y)) +
       geom_point(aes(color=body)) +labs(x =" ", y = " ") +
       scale_color_manual(values = c("#999999", "#0072B2")) +
       theme(legend.position="top",text = element_text(size=20),
       legend.text = element_text(size=20))

## right plot
ggplot(data.frame(x=points3[2,], y=points3[1,]) , aes(x=x, y=y)) +
       geom_point() +labs(x =" ", y = " ") +
       scale_x_continuous(limits = c(-1, 1)) +
       scale_y_continuous(limits = c(-1, 1)) +
       theme(legend.position="top",text = element_text(size=20),
       plot.title = element_text(hjust = 0.5)) +
       ggtitle(TeX("$P_1 \\bigcap P_2$"))
```

A.4. Rounding

The following scripts can be used to generate Figures 7, 8.

```
## figure 7
## left plot
ggplot(data.frame(x = c(points1[1,]), y = c(points1[2,])), aes(x=x,
       y=y)) + geom_point() +labs(x =" ", y = " ")+coord_fixed(ylim =
       c(-10,10))+scale_y_continuous(breaks = c(-10,0,10)) +
       theme(legend.position="top",text = element_text(size=20),
       axis.text.x = element_text(size=20), axis.text.y = element_text(size=20))
```

```
##right plot
ggplot(data.frame(x = c(points2[1,]), y = c(points2[2,])), aes(x=x,
       y=y)) + geom_point() +labs(x =" ", y = " ")+coord_fixed() +
       theme(legend.position="top",text = element_text(size=20),
       axis.text.x = element_text(size=20), axis.text.y =
       element_text(size=20))

## figure 8
#left plot
ggplot(data.frame(x = c(points1[1,]), y = c(points1[2,])), aes(x=x,y=y)) +
       geom_point() +labs(x =" ", y = " ")+coord_fixed(ylim =c(-10,10),
       xlim=c(-100,100))+scale_y_continuous(breaks = c(-10,0,10)) +
       scale_x_continuous(breaks = c(-100,0,100))+
       theme(legend.position="top",text =
       element_text(size=20), axis.text.x = element_text(size=20),
       axis.text.y = element_text(size=20))

#right plot
ggplot(data.frame(x = c(points2[1,]), y = c(points2[2,])), aes(x=x,y=y)) +
       geom_point() +labs(x =" ", y = " ")+coord_fixed(ylim =c(-10,10),
       xlim=c(-100,100))+scale_y_continuous(breaks = c(-10,0,10)) +
       scale_x_continuous(breaks = c(-100,0,100))+
       theme(legend.position="top",text = element_text(size=20),
       axis.text.x = element_text(size=20), axis.text.y =
       element_text(size=20))
```

A.5. Rotation and inscribed ball

The following script can be used to generate Figure 9.

```
## left plot
     cbp1 <- c("#999999", "#0072B2")
ggplot(data.frame(x = c(points2[1,],points1[1,]), body = c(rep("P",10000),
     rep("ball",10000)), y = c(points2[2,],points1[2,])) , aes(x=x, y=y)) +
     geom_line(aes(color=body)) + scale_color_manual(values =cbp1) +
     geom_point(shape=20) +labs(x =" ", y = " ")

## right plot
cbp1 <- c("#999999", "#0072B2")
ggplot(data.frame(x = c(points2[1,],points1[1,]),
     body = c(rep("rotated_P",10000), rep("P",10000)),
     y = c(points2[2,],points1[2,])) , aes(x=x, y=y)) +
     geom_line(aes(color=body)) + scale_color_manual(values = cbp1) +
     geom_point(shape=20) +labs(x =" ", y = " ")
```

A.6. Copola characterization and compound returns

The following R function is used in the script of Section 4.1 which generates Figure 10.

```
get_compound_return <- function(MatReturns) {
nassets = dim(MatReturns)[2]
compRet = rep(1,nassets)
  for (j in 1:nassets) {
    for (k in 1:dim(MatReturns)[1]) {
      compRet[j] = compRet[j] * (1 + MatReturns[k, j])
    }
    compRet[j] = compRet[j] - 1
  }
  return(compRet)
}
```

A.7. Values of indicators

The following script can be used to generate Figure 11.

```
n = length(I)
crisis_per = rep(NaN, n)
crisis_per[which(market_states %in% "crisis")] =
             I[which(market_states %in% "crisis")]

normal_per = rep(NaN, n)
normal_per[which(market_states %in% "normal")] =
             I[which(market_states %in% "normal")]

warning_per = rep(NaN, n)
warning_per[which(market_states %in% "warning")] =
             I[which(market_states %in% "warning")]

indi.data <- data.frame(x=rep(1:707,3), y=c(normal_per, warning_per,
                        crisis_per), period = rep(market_states,3))
ggplot(indi.data, aes(x=x, y=y))+ geom_line(aes(color=period)) +
  geom_point(aes(color=period)) + labs(x ="Dates", y="Indicator") +
  scale_y_continuous(breaks = 0:12)+
  scale_x_continuous(breaks=seq(from=1, to =707, by=50),
                     labels=c(dates[seq(from=1, to =707, by=50)])) +
  scale_color_manual(values=c("red", "blue", "orange")) +
  theme(legend.position="top",text = element_text(size=20),
        axis.text.x = element_text(size=10), axis.text.y =
        element_text(size=20))
```

A.8. Zonotope approximation

The following script can be used to generate Figure 12.

```
cbp1 <- c("#999999", "#0072B2")
ggplot(data.frame(x = c(points1[1,], points2[1,]),
       body = c(rep("zonotope",10000), rep("PCA_poly",10000)),
       y = c(points1[2,],points2[2,])) , aes(x=x, y=y)) +
       geom_line(aes(color=body)) + geom_point(shape=20) +
       labs(x =" ", y = " ") + scale_color_manual(values = cbp1) +
       theme(legend.position="top",text = element_text(size=20))
```