VOLUME ESTIMATION OF HIGH-DIMENSIONAL CONVEX BODIES

Jonathan Unger, Silvan Läubli, Manuel Bröchin, Emanuel Peter

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

It is known that calculating the volume of a convex body is NP-hard, but several polynomial-time approximation algorithms exist. In this project we implement and optimize a multiphase Markov chain Monte Carlo (MCMC) algorithm for the estimation of the volume of convex bodies. We introduce a caching approach to asymptotically reduce the runtime of the algorithm. Furthermore, we analyse the special case of bodies with sparse constraints and reduce work by generating specialized code at runtime. Finally, we identify the hotspots of our code and address them with single-core optimizations. Our final code outperforms other publicly available tools and manages to well approximate the volume of bodies with over 300 dimensions.

1. INTRODUCTION

Volume estimation of convex bodies is both of theoretical and of practical importance. As volume computation is NP-hard, the first polynomial-time approximation scheme marked a breakthrough and the topic is still an active research area to this day. Volume estimation has many applications, such as bioinformatics [1] and radiology [2]. Despite the importance of the topic not many practical tools for volume estimation exist and the ones that do usually cannot deal with more than several dozens of dimensions [3]. We provide an optimized single-core implementation of a state of the art algorithm for volume estimation of convex bodies. We also consider sparse bodies and achieve a better runtime than for the general case.

Related work. Simonovits [4] provides a survey on volume estimation algorithms, including very similar algorithms to the ones implemented in this work. Cunjing et al. [3] present *PolyVest*, a C++ implementation of an efficient algorithm based on shallow-beta cuts and MCMC coordinate-direction hit-and-run sampling for estimating volumes of convex bodies. Our work relies on the algorithm and analysis of their work, and further optimizes and extends it. Cousins et al. [5] also analyze, implement (Matlab) and benchmark a similar algorithm. They propose some convex bodies for a benchmark. Following their work, Ioannis et al. [6] provided another implementation (C++), called *VolEsti* that is faster for some bodies.

Contribution. While these other works focused on developing efficient algorithms, we try to provide a fast implementation of such an algorithm in terms of runtime and performance. To the best of our knowledge, we are the first to introduce the caching of intermediate results in the most relevant parts of this algorithm. This leads to an asymptotic speedup linear in the dimension. Similar to [5] we run multiple Markov chains in parallel, but in contrast to them we use SIMD vectorization with a single thread to achieve this. To the best of our knowledge, in this context, we are the first to consider the special case of sparse bodies and make use of customized representations to avoid redundancies. Specifically, we explore the benefits of code generation at runtime to reduce work.

2. BACKGROUND

In this section we first give a high-level overview of the algorithm. Then we present a more detailed description of the two main parts of the algorithm. Finally we present a cost analysis for the second part of the algorithm, which is the part we focus on in our optimizations.

Our algorithm estimates the volume of n-dimensional convex bodies, up to a relative error of ε. A body is given as an intersection of simpler and more easily representable convex bodies, namely polytopes and ellipsoids. A polytope is defined as the set $\{x : Ax \leq b\}$ for $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. An ellipsoid is the set $\{x : (x-e)^\intercal E(x-e) \leq 1\}$ for $E \in \mathbb{R}^{n \times n}$ positive-definite and $e \in \mathbb{R}^n$.

The algorithm is divided into two parts. The first preprocesses the body, in preparation for the second part. The second part estimates the volume via point sampling with a MCMC method. As sampling random points in thin, pencil-shaped bodies becomes exponentially hard in high dimensions, an affine transformation is applied in the preprocessing to round the body.

After this, the body is sandwiched between a unit ball and a ball of radius $2n$. We would like to sample points uniformly at random in the body, and count the number of points falling into the unit ball, to estimate the ratio of the volumes. Unfortunately, the ratio between the volume of the body and the unit ball is in general exponential, thus

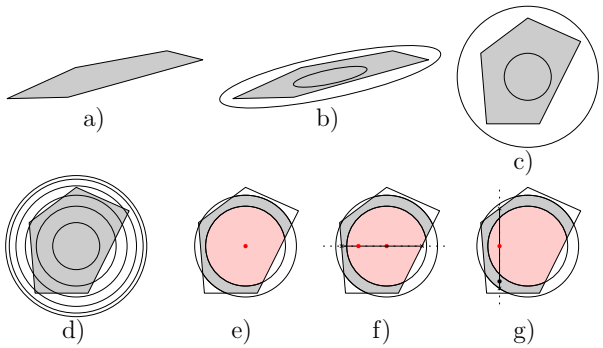

Fig. 1: a-c) depict the preprocessing, d) the zoneing, and e-g) the random point sampling via hit-and-run.

an exponential number of sample points is required. This phenomenon is known as the curse of dimensionality [7].

Our algorithm overcomes this by subdividing the body into nested zones. The zones are designed such that the probability that a sample point from one zone lands in the inner zone is at least $1/2$. Doing this for all pairs of adjacent zones, we can infer the volume of the whole body.

Preprocessing. First, the input body is sandwiched between two concentric ellipsoids, where (e, E) circumscribes the body and $(e, (2n)^2 E)$ is inscribed in the body (see Fig. 1 b). The existence of such ellipsoids is implied by the existence of the John ellipsoid, and can be found via the shallow-beta-cut ellipsoid method [4, 8]. An affine transformation maps the inscribed ellipsoid to the unit ball, and the circumscribed ellipsoid to a ball with radius $2n$, both centered at the origin (see Fig. 1 c).

To initialize the shallow-beta-cut, a bounding box is calculated, with the use of LP solving in the case of polytope input bodies. For this we rely on third party software; LP solving is out of scope for this project. Moreover, we experienced numerical issues during preprocessing: Because the initial bodies can be very skewed or pencil-like, the matrix of the bounding ellipsoid soon becomes skewed also, leading to numerical issues in high dimensions. Hence, the number of cuts that can be applied is fairly limited. Given these inherent difficulties we decided not to optimize and analyze our preprocessing code but focus on the MCMC part.

MCMC: Volume Estimation. Let C be a convex body. We subdivide the body into zones

$$Z_k := C \cap B(0, 2^{\frac{k}{n}})$$

for $k \in \{0, \ldots, n \log_2(2n)\}$, where $B(0, r)$ is the ball with radius r centered at the origin. This yields $N = n \log_2(2n)$ zones, where adjacent zones vary in volume by at most a factor 2 (see Fig. 1 d).

We want to sample in a zone Z_{k+1} and check how many sample points land in Z_k. This provides an approximation of the ratio $\frac{V(Z_{k+1})}{V(Z_k)}$. Doing this for all consecutive zones, we get the volume of the body by calculating

$$V(C) = \frac{V(Z_N)}{V(Z_{N-1})} \cdot \frac{V(Z_{N-1})}{V(Z_{N-2})} \cdots \frac{V(Z_2)}{V(Z_1)} \cdot V(Z_1)$$

Note that by construction Z_N equals the whole body and that $V(Z_1)$ is known analytically, as it is just the volume of an n-dimensional unit ball.

For each ratio approximation, we require $W = \mathcal{O}(N \cdot \varepsilon^{-2})$ sample points. W depends on the number of zones N, because relative errors accumulate. W quadratically depends on the desired relative error of estimation ε [3]. The point sampling inside a Z_k is done with a coordinate direction hit-and-run approach. The sample point is initialized to the origin, and updated in step $s + 1$ as follows (see Fig. 1 e-g):

- Pick random coordinate direction: From all n, we pick a coordinate direction j uniformly at random. Let d be the unit vector along j, that is $d_j = 1$ and $d_i = 0$ for $i \neq j$. The line in coordinate direction is given as $x_s + t \cdot d$ and parametrized by t. We want to pick the next sample point on this line.

- Calculate endpoints: As the next sample point has to also lie inside Z_k, we have to calculate the segment of the line $x_s + t \cdot d$ in Z_k, specifically t^+ and t^- that correspond to the endpoints. We go through all the bodies composing C, and the ball $B(0, 2^{\frac{k}{n}})$ limiting them to Z_k, calculate their intersection points with the line and update t^+ or t^- if necessary. These are the intersections:

- Polytope intersect: We intersect each constraint $a_i^\intercal x \leq b_i$ with the line $x_s + td$. This gives us intersection points $t_i = \frac{b_i - a_i^\intercal x_1}{a_i^\intercal d}$ and we update t^+ or t^- if necessary.

- Ellipsoid intersect: For each ellipsoid we plug in $x_s + td$ for x and get the quadratic equation $(x_s + td - e)^\intercal E(x_1 + td - e) = 1$. The two solutions t_0 and t_1 are the two endpoints of the segment in the ellipsoid. We update t^+ or t^- if necessary.

- Ball intersect: The balls are given by $\{x : ||x|| \leq r^2\}$. As for the ellipsoids, the intersection has two solutions t_0, t_1.

- Update sample point: Having collected the endpoints from the polytopes, ellipsoids and balls in t^+ and t^-, We pick a $\tilde{t}$ uniformly at random on the segment $[t^-, t^+]$. This gives us the new sample point $x_{s+1} = x_s + \tilde{t}d$ in Z_k.

- Determine the zone: To count x_{s+1} for the correct zone, we compute $\log_{2^{\frac{1}{n}}}(\|x_{s+1}\|)$ to get the zone index (c.f. [3] for reuse of sampling points).

Cost analysis. As explained in the paragraph about preprocessing, we focus on the MCMC part of the algorithm. The volume estimation requires the approximation of N ratios, with W sample points each. However, the total number

of sample points is not deterministic because sample points may be reused (c.f. [3] where this approach is explained thoroughly). Still, the runtime is asymptotically bounded by $\mathcal{O}(T_C \cdot n^2 \log^2 n \cdot \varepsilon^{-2})$, where T_C is the cost for a single sample point in a body C.

In what follows, we analyze T_C. For this, our cost measure counts the total number of floating point operations (*add, mul, cmp, min, max, div, sqrt*). But we put a special focus on the most expensive flops (*div, sqrt*).

- Pick coordinate direction: Picking the direction requires generating a random integer, thus no flops.
- Polytope intersect: For each of the m constraints, we calculate $t_i = \frac{b_i - a_i^\intercal x}{a_i^\intercal d}$, where $a_i^\intercal d = a_{i,j}$. We determine the sign for each t_i with a *cmp*. We update t^+ and t^-, by calculating the *min* of all positive t_i, and the *max* of all negative t_i respectively. Thus, per constraint, we have one dot-product ($2n$ flops), a *div*, and 5 *add/cmp/min/max*, and in total $2nm + 6m$ flops.
- Ellipsoid intersect: We solve the quadratic expression for t_0 and t_1 with the quadratic formula. This requires a matrix-vector product, a vector-vector product, n subtractions, and the computation of the roots of the quadratic equation which includes computing a *sqrt* and a *div*. In total, we obtain a flop count of $3n^2 + 3n + 11$.
- Ball intersect: We need to solve the quadratic equation $\|x\|_2^2 + 2x^\intercal dt + t^2 = r_z^2$ for t. The expensive operations are finding $\|x\|$ ($2n$ flops), and the *div* and *sqrt* for solving the quadratic equation. In total, we have $2n + 15$ flops.
- Update sample point: The single coodinate direction update requires only one call to the random number generator and one flop.
- Determine the zone: The libc *log* is the costliest call, and implemented with multiple but constantly many flops, thus still pales in comparison with the other parts.

Note, that we do not state a single number of flops required. As the polytope and ellipsoid intersects are the most expensive, and their flop count directly depends on the composition of the body, the total number of flops can be computed from the stated components above.

3. IMPLEMENTATION AND OPTIMIZATION

In the following we describe several optimization steps that we performed. We consider three major representations of the polytope body and one for the ellipsoid. With these, we explore optimizations on the algorithm level (caching, sparse representation, multiple Markov Chains) and instruction level (SIMD vectorization, *div* precomputation).

Polytope: Baseline. As evident from above, the bulk of the work is calculating $t_i = \frac{b_i - a_i^\intercal x}{a_i^\intercal d}$ for all i, i.e. for each sample point we compute m dot products. To get the largest negative and the smallest positive t_i we perform m *comp*s and in total m *max*s or *min*s in each iteration. Note that we access the rows of A one by one, thus the row-major format is ideal for spatial locality.

Polytope: Caching dot-products. Instead of calculating all m dot-products in each step s, we cache the numerator $c_s^i := b - a_i^\intercal x$ for $i \in [m]$ and update it together with x_s. When updating $x_{s+1} \leftarrow x_s + \tilde{t}d$, we also update every element of the cache $c_{s+1}^i \leftarrow c_s^i - a_{i,j}\tilde{t}$ for all $i \in [m]$, with 2 flops each. Whereas previously, we had $2nm + 6m$ flops for the *intersect*, we now only have $5m$ flops for the *intersect* (3 *cmp*, *min* or *max* and one *div*). Additionally, we now do a *cacheUpdate* with $2m$ flops. In total, this gives a speedup of $\Theta(n)$. Furthermore, the number of memory accesses in A has decreased to one entry per row - all in the same column. We can now optimize spatial locality by choosing column-major order for A.

Polytope: Precompute Inverse. Out of the 5 flops per constraint in *intersect*, *div* is the bottleneck with its large gap and latency. We divide only with elements $a_{i,j}$ of A which remain constant throughout the algorithm. Thus we can precompute and store the inverses $\frac{1}{a_{i,j}}$ in an additional matrix and replace the *div* with a *mul*. This comes at the cost of almost doubling the storage for the polytope data.

Polytope: Vectorized. Previously we relied on branching to update either t^+ or t^-. In the vectorized case we maintain vector accumulators and apply masking and the *blendv* instruction to update them. While the flop count remains the same, we got rid of the branching overhead.

Ellipsoid. Also for ellipsoids, *intersect* benefits from caching. The intersection points are given by the roots of the quadratic equation: $\mathtt{a}t^2 + \mathtt{b}t + \mathtt{c} = 0$ with:

$$\mathtt{a} = d^\intercal E d,\ \mathtt{b} = 2d^\intercal E(x-e) \text{ and } \mathtt{c} = (x-e)^\intercal E(x-e) - 1$$

Here we cache $E(x-e)$ and $\mathtt{c}$; everything else is accessible in constant time. $\mathtt{a}$ is simply an entry of E and the direction vector d is given as input to the intersection function, so the computation of an intersection takes constant time. The *cacheUpdate* requires a constant number of operations to update $\mathtt{c}$ and $2n$ operations for the update of $E(x-e)$.

Optimizing the *cacheUpdate* using *FMA*s and vectorization improves its performance up to the memory bound, as the operational intensity approaches $1/8$. The new *intersect* function can be optimized with instruction reordering and *FMA*s. However, *intersect* has an inherent performance bound, since the quadratic formula depends on a *div* and a *sqrt*, which are executed on the same port. Nevertheless, this can be mitigated by increasing instruction level parallelism. We compute the inverse of the denominator of $\frac{-\mathtt{b} \pm \sqrt{\mathtt{b}^2 - 4\mathtt{ac}}}{2\mathtt{a}}$ in parallel to the discriminant. Once the *sqrt* is done we can

multiply it with this inverse. The new costs of our cached functions are 11 flops for the intersection and $2n + 5$ flops for the cache update. This is now linear instead of quadratic.

Polytope: Sparse Constraints. If $A_{i,j} = 0$ the constraint i is parallel to the unit vector in coordinate direction j and can be ignored when walking in this direction. Thus if A is sparse a lot of entries are ignored and we can get a speedup by storing only non-zero entries in the compressed sparse columns (CSC) storage format (c.f. Fig. 2).

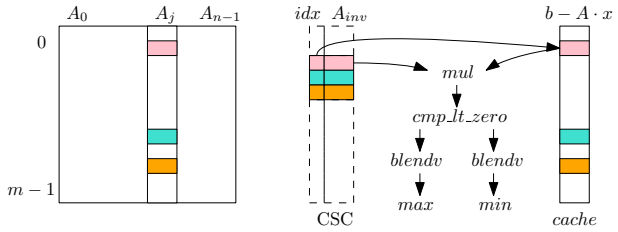

Fig. 2: Inverses of non-zero entries of the columns of A are stored contiguously in memory in A_{inv} together with their row indices (in color for col j). The cache $b - A \cdot x$ is stored as before, thus we have to do non-linear access on it.

Let nz_A denote the number of non-zeros in A and by nz_j the number of non-zeros in column j of A. Fig. 2 visualizes the SIMD computation and memory reads performed in *intersect* for coordinate direction j. We read $2nz_j$ doubles and nz_j ints and we do $4nz_j$ flops (*mul, cmp, min, max*). Compared to the dense implementation we do less work per non-zero entry (we don't have to check if an entry is 0) and we read more data (the row-indices). Thus the operational intensity is lower and the performance is more memory bound. Additionally we now need to gather non-contiguous entries of the cache in one SIMD vector and lose spatial locality. For the *cacheUpdate* we both read and write scattered locations and also need to read the row-indices and perform $2nz_j$ flops. Again, we reduce work by only considering non-zero entries.

A problem is that the preprocessing will modify the input matrix A, to be precise, preprocessing will update $A \leftarrow AL$ where L is a lower triangular matrix [3]. Even if A is sparse, AL may be less sparse. But because L is a triangular matrix we can hope that some of the sparsity of A will be retained.

Polytope: Just-in-time code generation. In the sparse case we do only little work and the runtime is dominated by the latency of the flops and data look-ups. Note, that whether t_i is negative or positive only depends on $a_{i,j}$. Thus the case distinction of taking *min* or *max* is determined by the input. These values are only known at runtime, but we can generate a function (section of machine-code) per coordinate direction at runtime to compute the intersection with the polytope constraints that hardcodes this decision. With this approach the flops per non-zero entry decrease to a *mul* and either a *max* or a *min* (c.f. Fig. 3). This code is put in executable memory via *mmap*. A function table maps dimensions $j \in [n]$ to the respective code section.

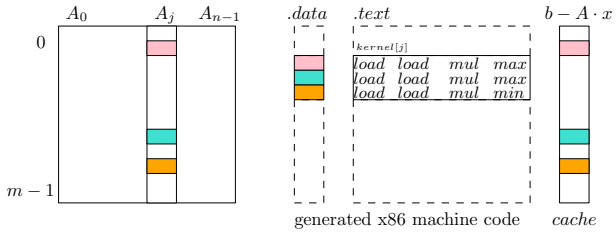

Fig. 3: Just-in-time compilation approach: For each coordinate direction a section of code is generated, proportional to the number of non-zeros in the column of A. Only two *load*s one *mul* and either a *min* or *max* are emitted.

This approach should reduce the latency because the number of flops are reduced and the row-indices do not have to be loaded. However, this approach increases the code size which may slow down the instruction load and scheduling process if the instruction cache is exceeded.

To vectorize this approach without needing to resort back to case distinctions, we have to ensure that only positive or negative $a_{i,j}$ are loaded into one vector. We could achieve this by gathering $a_{i,j}$ of the same sign from scattered memory locations into one vector. But this leads to an undesirable increase of instructions required, leading to a blow-up in the machine-code size. Instead, we only load adjacent packed double vectors. We greedily try to minimize the number of loads. For vectors containing mixed signs or zero entries we then invalidate (set to $\pm\infty$) all but the desired entries. In some cases this means we need to process a vector twice, once for each sign. In the best case, vectorization could lead to a 4x speedup. In the worst case with only one valid entry per vector, we the load overhead will make it slower than the unvectorized approach. To make speedup more likely, we shuffle the rows of the constraint matrix A with a best-effort algorithm. The algorithm greedily places entries of the same sign next to each other.

Improvements on randomness. Two random numbers are needed in each update of x: an integer to select a coordinate direction and a double to chose a point on the segment. In the sparse setting, the C standard library *rand* function incurs relatively high cost, compared to *intersect*.

A possible approach is to precompute chunks of random numbers and then load them from memory on demand. Unfortunately, the *rand* function is mostly limited by its latency, therefore precomputation only shifts the time of execution without any benefit. Furthermore, maintaining the chunk imposes some overhead. Another approach is to implement a pseudorandom number generator (PRNG) ourselves. We first consider a straightforward implementation of a mersenne twister [9]. Unfortunately, optimizing and vectorizing this PRNG is quite complex. Instead we opt for a simple shift register as proposed by Marsaglia [10]. Since

it consists only of a few integer operations (3 *shift*, 3 *xor*, 1 *and*), this is easily vectorizable with SIMD instructions.

There is a small additional overhead when generating a random double in a range for choosing a random point on a segment. This requires 4 additional flops.

Cache vector norm for ball-intersect. In the sparse case also the ball-intersect becomes significant as it requires $2n$ flops for the vector norm of x. In step s we update $x_{s+1} = x_s + \tilde{t}d$. We cache the vector norm as $c_s = \|x_s\|$ and update $c_{s+1} = c_s - (x_s)_j^2 + (x_{s+1})_j^2$ with 6 flops.

Multiple Markov Chains. So far all our implementations have been memory bound. In the table below we list the work W and memory IO induced by computing one *intersect* and performing the corresponding *cacheUpdate*.

Representation	W [Flops]	IO [Bytes]	I [Flops/Bytes]
Dense	$7m$	$40m$	0.175
CSC	$6nz_j$	$48nz_j$	0.125
JIT	$4nz_j$	$40nz_j$	0.100
Ellipsoid	$2n$	$24n$	0.083

Explanations for these numbers are given in the paragraphs where we discuss the respective implementations.

To increase operational intensity we run multiple Markov chains at once. This means maintaining $k > 1$ independent sample points and updating each one in every step. To do one update step, we choose a single coordinate direction for all k Markov chains, calculate the intersections and update to independent points on the respective segments. This approach has two potential benefits: First, the computational intensity is increased. While the whole work is multiplied by k, only parts of the IO are multiplied by k. Because we update all Markov chains in the same coordinate direction, the corresponding column of A needs to be read only once for all chains. E.g. the intensity for the dense implementation is $\frac{7k}{16+24k}$. While this increases the performance, unfortunately, the computation is still memory bound (c.f. Sec. 4). A second benefit of this approach is that the latency of one *intersect* becomes less significant, especially in the sparse setting where latencies dominate the runtime. If we can perform multiple intersections at once, with little more than the latency of a single one, this can lead to significant speedup.

Cousins et al. [5] also run multiple Markov chains at a time, though they use multiple threads, one per chain. The benefit or running k chains in parallel is that one has to run each of them for only $\frac{1}{k}$'th of the number of steps. However, their theoretical and experimental analysis shows that the MCMC method requires a certain number of steps, to ensure the chain has mixed sufficiently, thus one cannot scale the number of chains arbitrarily; they recommend $5 - 10$.

We use 4 or 8 sample points at a time, this is easily implementable with SIMD vector instructions and is justified by the findings of Cousins et al. [5]. We implement this approach for all polytope representations (dense, CSC, JIT).

Body	Dim	Max dim	Constrs	nz_A	Vol
cubeRot$_n$	n	100	$2n$	$2n^2$	2^n
cross$_n$	n	13	2^n	$n2^n$	$\frac{2^n}{n!}$
dens$_d$	$2 \cdot 10^2$	200	$2 \cdot 10^3$	$4d \cdot 10^3$	-
2var$_n$	n	200	$10n$	$20n$	-
Birkhoff$_n$	$(n-1)^2$	324	n^2	-	-
ellipsoid$_n$	n	200	n	n	-

Fig. 4: Body types with their dimension, the highest dimension we used in our experiments, number of constraints, number of nonzeros and volume (if known).

4. EXPERIMENTAL RESULTS

In this section, we discuss the experimental setup and input data for our benchmarks. Later, we present our experimental results in a similar order as above: First we show the results for the ellipsoids and dense polytopes. Second, we compare the performance of the sparse implementations with the best dense one. Then, investigate the impact of further optimizations for sparse implementations. Next, we compare the operational intensity of the dense and sparse implementations and show that we can increase operational intensity by running multiple Markov chains at once. Finally, we see how our implementation compares to two other implementations that we found online.

Experimental setup. Unless otherwise mentioned, we ran all of our experiments on an Intel Haswell i7-4870HQ, with base frequency 2.50 GHz, and 128kB L1d and L1i cache each, 256KB L2 and 6MB L3 cache. We compiled with gcc version 9.3.0 using the flags *-march=native -mfma -ffast-math -O3*. The stream benchmark [11], configured for data sizes of around 64KB, relevant for our input data, reaches about 10 Bytes/Cycle.

Input data. In Table 4 we list the bodies used in our experiments together with their key parameters. cubeRot$_n$ is the n-dimensional cube with sidelength 2 that was transformed by a random rotation matrix. We expect this matrix to have no zero entries because we rotate the cube in each dimension around a random angle. cross$_n$ is the n-dimensional cross polytope. dens$_d$ is a polytope whose constraint matrix consists of $d\%$ non-zeros (random values at random positions) and $(100-d)\%$ zeros. 2var$_n$ is a polytope in dimension n with $10n$ constraints where each constraint contains exactly 2 variables. Birkhoff$_n$ is the n-th Birkhoff polytope proposed as a benchmark by Cousins et al. [5]. ellipsoid$_n$ is an n-dimensional ellipsoid that is centered at the origin. The principal semi-axes of each dimension are random.

Benefit of caching. Figure 5 shows the $\mathcal{O}(n)$ speedup we achieved on the algorithm level for dense implementations with caching. Asymptotic speedup is hard to represent in a plot, thus we present a table. The baseline implementation mostly computes dot-products, which are very efficient. Although the cached implementation had linearly less flops,

the remaining flops are the costly ones (c.f. discussion about *div* above). Thus the speedup with caching is not directly n. But together with inverse precomputation and vectorization we reach linear speedup with *inv-vec*.

n	baseline [cycles]	cached [cycles]	*inv-vec* [cycles]
10	$3.70 \cdot 10^2$	$3.89 \cdot 10^2$	$1.19 \cdot 10^2$
20	$1.35 \cdot 10^3$	$6.77 \cdot 10^2$	$1.56 \cdot 10^2$
40	$5.10 \cdot 10^3$	$1.39 \cdot 10^3$	$2.48 \cdot 10^2$
60	$1.25 \cdot 10^4$	$2.15 \cdot 10^3$	$3.47 \cdot 10^2$
100	$4.06 \cdot 10^4$	$3.57 \cdot 10^3$	$5.78 \cdot 10^2$

Fig. 5: Mean runtime of 10^4 runs for *intersect* on cubeRot$_n$.

Optimizations for dense polytopes. Fig. 6 shows the performance of different cached, dense implementations of *intersect*. Implementations *ref-impl-div* and *vectorized-div* both use the *div* instruction. In *vectorized-div* we do 20 flops per loop iteration: a *div*, two *cmp*s and a *max* and a *min*. Thus the performance roof is at $\frac{20}{27}$ flops/cycle in that case, given the 27 cycle gap of *div*. Replacing the *div* by a *mul* (*ref-impl-mul*) and vectorizing (*vectorized-mul*) gives about 3x speedup to around 2 flops/cycle. Without the *div* we have a gap of 4 cycles per loop iteration from the ops on port 2, thus the performance roof increases to 5 flops/cycle. *vectorized-mul* falls short of this roof. Since the amount of work is quite low we cannot amortize the overhead of function calls and memory latency. Using multiple Markov chains we raise the work per intersection call and thus also the operational intensity (*parallel-4* and *parallel-8*). This gets us to about 4 flops/cycle. In total we achieved a speedup of 10x from *ref-impl-div* to *parallel-8*. Other experiments showed that 5flops/cycle are almost reachable, if the columns of the constraint matrix are longer, e.g. in the cross polytope where $m = 2^n$.

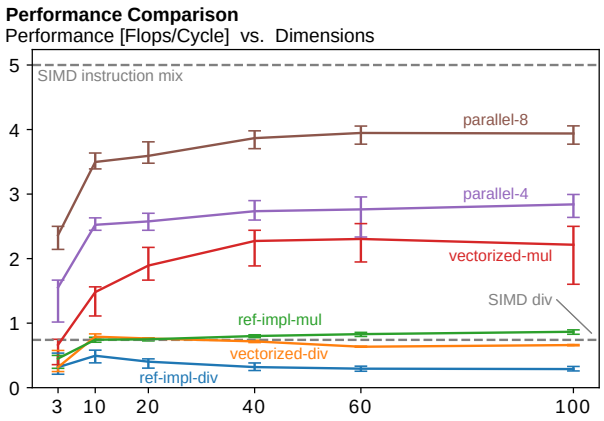

Fig. 6: Performance of a single *intersect* (without *cacheUpdate*) on $cubeRot_n$, with 10^5 repetitions and 95% CI. *parallel-4* and *parallel-8* perform 4 and 8 times the work, respectively, but are based on the same instruction mix.

Ellipsoid. Fig. 7 depicts the roofline plot for the *cacheUpdate* function of the ellipsoid. This function dominates the volume estimation for ellipsoids with its linear runtime. The plot shows, that the fastest vectorized implementations reach the memory roof of the machine for $n = 60$ and slide down the roof for bigger sizes which is also the case when running the entire MCMC function. For low n constant overhead disproportionally increases the operational intensity. For high n the intensity approaches 0.08. *ref-impl* gets vectorized by the compiler but does not scale well. Inspecting the generated asm shows that there is an inefficient data access scheme. *fma-impl* uses explicit *FMA*s and leads to unvectorized machine code.

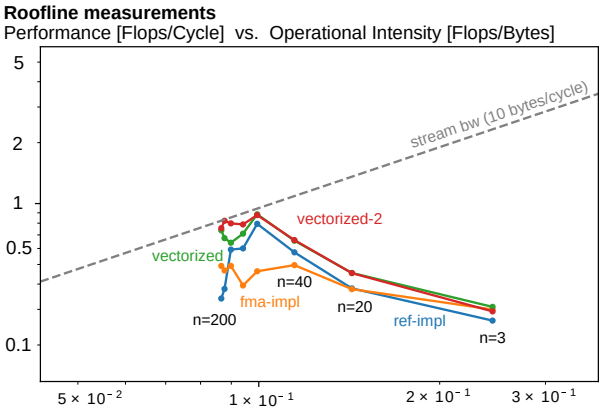

Fig. 7: Roofline plot for the *cacheUpdate* of an $ellipsoid_n$ for ($n = 3 \ldots 200$). *vectorized-2* uses loop-unrolling.

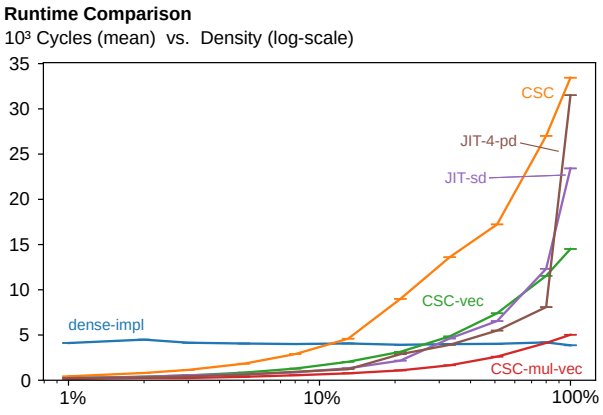

Fig. 8: Runtime of a single *intersect*, on dens$_d$ with 2000 repetitions. Error bars are omitted, as there is an inherent variability of work (non-zeros per column), due to the random body generation.

Benefit of CSC and JIT for bodies of varying density. Fig. 8 compares how our sparse implementations perform on bodies of varying density w.r.t. the dense implementation. The dense implementation always performs the same work, no matter the density. The sparse implementations CSC

and JIT do work proportional to the number of non-zeros. Thus they are much faster up to a certain density when their overhead outweighs the benefits. For CSC, we show three implementations: a non-vectorized one, a vectorized one that is significantly faster, and one that replaces the *div* with the *mul* (precomputation). Compared to *dense-impl*, *csc-mul-vec* only has the overhead of reading the row indices. It is always faster than *dense-impl*, except for the densest cases. For JIT, we show two code generators: *JIT-sd* uses scalar doubles and *JIT-4-pd* uses 4-packed-doubles. The latter can be faster because of SIMD parallelism. However, it can also be slower if the non-zero entries of the same sign are not adjacent, thus the performance depends on the structure of the input. For the JIT implementations, we experience a heavy performance deterioration at high densities. This is because for dense cases, we exceed the L3 cache size. For 100% density, the 4-packed double version generates 5.5MB instruction code and has a working set of over 6MB, in total almost double the 6MB L3 cache size.

Very sparse cases. Fig. 9 shows the runtime of the intersect function of several *CSC* and *JIT* implementations. Here we consider the very sparse 2var_n bodies. Note that these polytopes have on average 20 non-zeros per column, independent of n. Indeed, the plot confirms that the runtime of our sparse implementations depend mostly on the number of non-zero elements and not on n or m. Because the matrix entries are random, the actual number of non-zero elements can vary greatly between columns which explains the wide confidence intervals. Nevertheless, the tendency seems to be that the runtime of all sparse implementations increases slightly with increasing dimension. This cannot be due to cache misses, given that all the data and instructions fit in the respective L1 caches. A possible explanation is that in higher dimensions we access memory, namely the dot-product cache, from more scattered locations. All *CSC* implementations use SIMD vectorization. *CSC-nogather* achieves a 25% speedup by replacing *vgatherdpd* with explicit *vmovhpd* and *vinsertf128* to load scattered data and avoids creating an index vector. *CSC-mul* additionally gets rid of *div* by storing inverse elements of A and achieves another speedup of 30%. *JIT* versions differ in their use of instruction types (scalar,2/4-packed). It seems in this very sparse case best effort sorting of the constraints still manages to mitigate the potential overhead of using packed doubles. JIT implementations achieve a speedup compared to CSC because they perform less flops and need not read row indices.

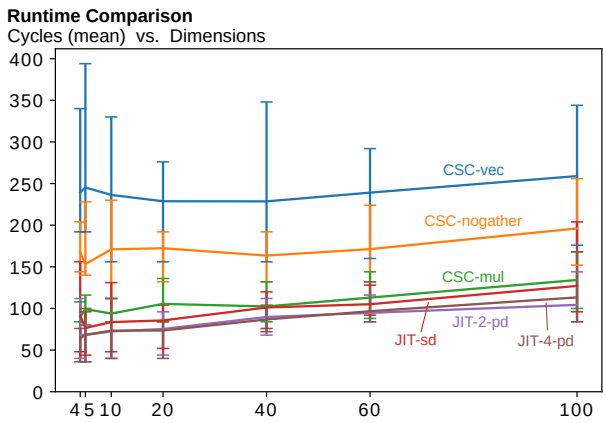

Fig. 9: Runtime of a single intersection computation on 2var_n bodies with 10^5 runs. Errorbars show the 95% CI.

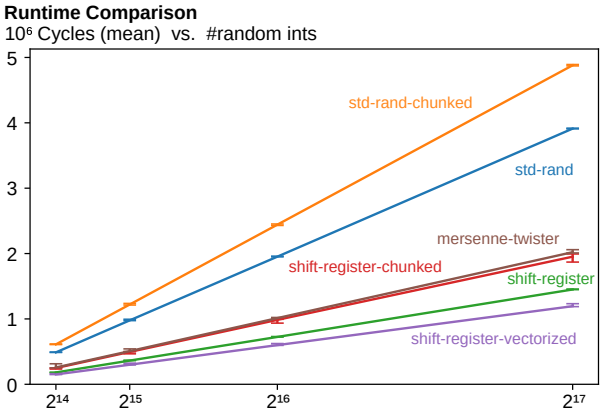

Fig. 10: Runtime plot for PRNGs (95% CI).

Improvements on randomness. Fig. 10 shows all our approaches for our PRNG. We can see that precomputing chunks imposes an overhead of 20-25% for both the standard C and the shift register implementation. The use of a straightforward mersenne twister leads to a speedup of 2x over *std-rand* whereas the use of a shift register results in a speedup of 2.75x. The *shift-register-vectorized* reaches a speedup of 3.25x despite the inherent chunking overhead.

Optimizations relevant for sparse bodies. For sparse bodies *intersect* approaches constant cost and computing $\|x\|^2$ for the ball intersect and generating random numbers becomes relatively costly. Fig. 11 shows both the *CSC* and *JIT* implementations on a very sparse body, where caching the squared-norm gives about a 30% speedup. Additionally using the less costly shift-register PRNG we achieve a total speedup of almost 2.

Operational intensity. Fig. 12 compares operational intensity of *dense*, *CSC* and *JIT* implementations. The plot confirms that all our implementations are memory bound for sufficiently large n. For all implementations $I(n)$ is decreasing in n. As expected $I(n)$ approaches the operational intensity given by the *intersect* and *cacheUpdate* computations for large n because their cost and memory traffic scale with n whereas most other costs stay constant. As detailed in Sec. 3, the sparse implementations have much lower operational intensity than the dense implementations.

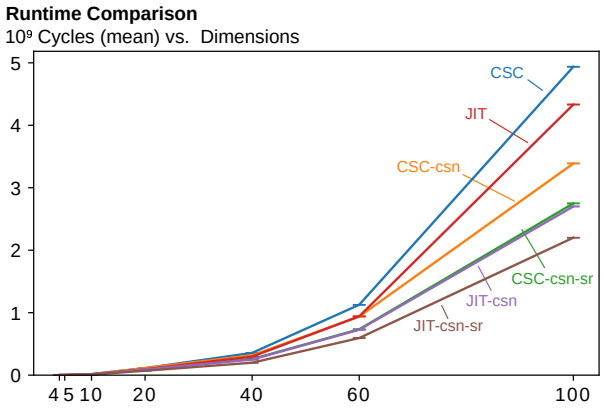

Fig. 11: Runtime of volume estimation with 2var_n ($\varepsilon = 0.8$). One run suffices for stable measurement due to long runtime. *csn*: cached-squared-norm. *sr*: shift-register.

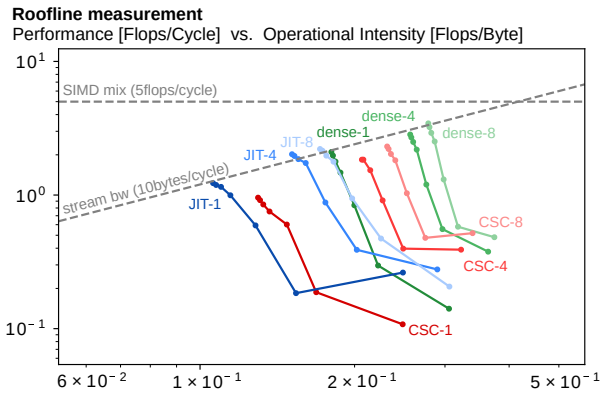

Fig. 12: Roofline plot for volume estimation of $cubeRot_n$ ($n = 3 \ldots 100$). 10 runs per experiment (sufficient for stable measurement due to long runtime). Number suffixes describe the number of Markov chains run in parallel.

For CSC this holds due to memory overheads, for JIT because it performs less flops. This plot clearly shows that the operational intensity can be increased for all implementations by sampling multiple Markov chains. Running the sparse implementations cubeRot_n gives an upper bound on the achievable performance for sparse bodies where base latency is not amortized and discontiguous memory accesses incur further overhead.

Comparison to alternative Implementations. Cousins et al. [5] propose the Birkhoff polytope as a benchmark for volume estimation algorithms and implementations. Fig. 13 shows a runtime comparison to both Polyvest [3] (C++) and the implementation of Cousins et al. [5] (Matlab). In this benchmark we ran the whole application, including the preprocessing. PolyVest [3] can run on multiple threads, but this unfortunately slowed down its execution, so we ran it only on a single thread. PolyVest did not terminate within an hour for bodies larger than $\mathcal{B}_{11}$. We ran Cousins et al. [5] with 8 Markov chains, one per thread. This implementation is already considerably faster, and terminates after 20 min. for $\mathcal{B}_{15}$ (196 dimensions). However, the code asserted on larger bodies. In [5] they also only report results up to $\mathcal{B}_{15}$. Our *dense-8* implementation is even faster. Representing the determinant of matrices in the preprocessing and the telescoping product in the volume estimation with the logarithm extended our the numerical range beyond 10^{308} of *doubles*. Now our code can handle $\mathcal{B}_{19}$ (324 dimensions) within 14 minutes. For larger bodies, we encountered further numerical issues in the preprocessing, most likely the shallow-beta cuts are not performed with enough accuracy.

Body	PolyVest [3]	Cousins et al. [5]	*dense-8*	Vol Estim.
$\mathcal{B}_3$	$7.57 \cdot 10^{-2}$	$2.95 \cdot 10^{-1}$	$3.84 \cdot 10^{-2}$	1.13
$\mathcal{B}_4$	$3.45 \cdot 10^{-1}$	$6.50 \cdot 10^{-1}$	$8.65 \cdot 10^{-2}$	$5.95 \cdot 10^{-2}$
$\mathcal{B}_5$	1.88	1.59	$1.75 \cdot 10^{-1}$	$1.37 \cdot 10^{-4}$
$\mathcal{B}_6$	7.44	3.22	$4.70 \cdot 10^{-1}$	$7.11 \cdot 10^{-9}$
$\mathcal{B}_7$	$2.72 \cdot 10^{1}$	$1.06 \cdot 10^{1}$	1.20	$5.47 \cdot 10^{-15}$
$\mathcal{B}_8$	$7.68 \cdot 10^{1}$	$2.05 \cdot 10^{1}$	2.73	$4.48 \cdot 10^{-23}$
$\mathcal{B}_9$	$2.13 \cdot 10^{2}$	$3.71 \cdot 10^{1}$	6.07	$2.61 \cdot 10^{-33}$
$\mathcal{B}_{10}$	$5.19 \cdot 10^{2}$	$9.63 \cdot 10^{1}$	$1.22 \cdot 10^{1}$	$7.97 \cdot 10^{-46}$
$\mathcal{B}_{11}$	$2.76 \cdot 10^{3}$	$1.61 \cdot 10^{2}$	$2.23 \cdot 10^{1}$	$1.33 \cdot 10^{-60}$
$\mathcal{B}_{12}$	Timeout	$2.76 \cdot 10^{2}$	$3.96 \cdot 10^{1}$	$7.25 \cdot 10^{-78}$
$\mathcal{B}_{13}$	-	$4.50 \cdot 10^{2}$	$6.73 \cdot 10^{1}$	$1.17 \cdot 10^{-97}$
$\mathcal{B}_{14}$	-	$7.41 \cdot 10^{2}$	$1.12 \cdot 10^{2}$	$5.14 \cdot 10^{-120}$
$\mathcal{B}_{15}$	-	$1.22 \cdot 10^{3}$	$1.74 \cdot 10^{2}$	$5.10 \cdot 10^{-145}$
$\mathcal{B}_{16}$	-	Exception	$2.62 \cdot 10^{2}$	$8.60 \cdot 10^{-173}$
$\mathcal{B}_{17}$	-	-	$3.93 \cdot 10^{2}$	$2.58 \cdot 10^{-203}$
$\mathcal{B}_{18}$	-	-	$5.72 \cdot 10^{2}$	$1.24 \cdot 10^{-236}$
$\mathcal{B}_{19}$	-	-	$8.22 \cdot 10^{2}$	$6.18 \cdot 10^{-273}$
$\mathcal{B}_{20}$	-	-	Num. issues	-

Fig. 13: Runtime volume estimation of Birkhoff bodies in seconds for $\varepsilon = 0.2$. Time limit: 1h. Run on an 8-core Intel Haswell i7-4720HQ 2.60GHz base frequency and MatlabR2019b. Vol Estim. is output of our algorithm.

5. CONCLUSIONS

We implemented an optimized MCMC algorithm for estimating the volume of arbitrary intersections of polytopes and ellipsoids in high dimensions. We managed to get an $\Theta(n)$ speedup with a caching approach and applied further optimizations such as precomputation, vectorization and using multiple Markov chains. In the case of sparse polytopes, we achieved additional speedup using specialized representations, namely compressed sparse column format and just-in-time compilation. The latter reduces branching and flops.

Given that the computation is memory bound, we increased performance by raising the operational intensity with multiple Markov chains. We were able to outperform previous implementations both in terms of runtime and the number of dimensions we can handle. In particular, our algorithm provides estimates of the volume of the 324-dim Birkhoff polytope.

6. CONTRIBUTIONS OF TEAM MEMBERS

We present the contribution of all team members:

Jonathan. Helped implement part of the PRNG optimizations (C). Vectorized several functions for dense bodies (C). Implemented the generation of a family of convex bodies required for benchmarking and analysis (C++). Participated in the joint performance counting and evaluation.

Silvan. Tried different PRNGs (C). Tried out chunking for PRNGs (C). Vectorization of shift register (C). Caching mechanism for ellipsoid (C). Optimization of intersection and cache update for ellipsoids (reordering including finding/implementing lower latency bounds, vectorization) (C). Worked together with Emanuel to improve and extend the benchmarking facility (C++). Setup of plotting code including roofs (performance roofs for roofline and performance plot, memory roofs for roofline plot and I/O-plot) (Python). Participated in the joint performance counting and evaluation.

Manuel. Implemented the preprocessing (C). Extended the "embedded language" used for benchmark specification and plotting code (Python). Implemented sparse column format CSC (C). Implemented caching and inverse trick for CSC. SIMD-optimizations for CSC, in particular trying out various random-memory access schemes (AVX2 gatherpd vs. AVX vinsertf128 and unpack) and reducing *cmp*s by using *blendv* instead. Participated in the joint performance counting and evaluation.

Emanuel. Set up facilities to allow for selection of one version of multiple function implementations (C++). Worked together with Silvan to improve and extend the benchmarking facility (C++). Set up a facility to simplify flop and byte counting and calculating performance (C++). Performed multiple VTune analyses to find bottlenecks not uncovered directly by the theoretical performance analysis. Implemented both column and row major format for dense polytope intersect (C). Designed and implemented the caching mechanism for the dot-product, used for dense and sparse implementations of the polytope intersect (C). Helped designing and implementing inverse precomputation for polytope intersect (C). Designed, implemented and optimized the *JIT* variant for sparse polytopes (C, AVX, asm, machine-code). Participated in the joint performance counting and evaluation.

7. REFERENCES

[1] Alfredo Braunstein, Roberto Mulet, and Andrea Pagnani, "Estimating the size of the solution space of metabolic networks," *BMC Bioinformatics*, vol. 9, no. 1, pp. 240, May 2008.

[2] L. T. Cook, P. N. Cook, K. Rak Lee, S. Batnitzky, B. Y. S. Wong, S. L. Fritz, J. Ophir, S. J. Dwyer, L. R. Bigongiari, and A. W. Templeton, "An algorithm for volume estimation based on polyhedral approxi mation," *IEEE Transactions on Biomedical Engineering*, vol. BME-27, no. 9, pp. 493–500, 1980.

[3] Cunjing Ge and Feifei Ma, "A fast and practical method to estimate volumes of convex polytopes," in *Frontiers in Algorithmics*. 2015, pp. 52–65, Springer International Publishing.

[4] Simonovits Marklos, "How to compute the volume in high dimension?," *Math. Program., Ser. B 97*, pp. 337–374, 2003.

[5] Ben Cousins and Santosh Vempala, "A practical volume algorithm," *Mathematical Programming Computation*, vol. 8, 2015.

[6] Ioannis Z. Emiris and Vissarion Fisikopoulos, "Efficient random-walk methods for approximating polytope volume," 2014, SOCG.

[7] Richard Bellman, "Dynamic programming.," Princeton, N. J.: Princeton University Press, 1957.

[8] Martin Grötschel, László Lovász, and Alexander Schrijver, *The Ellipsoid Method*, pp. 64–101, Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[9] Makoto Matsumoto and Takuji Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," .

[10] George Marsaglia, "Xorshift rngs," *Journal of Statistical Software, Articles*, vol. 8, no. 14, pp. 1–6, 2003.

[11] John D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.