An implementation of exact mixed volume computation

Anders Nedergaard Jensen

Technische Universität Kaiserslautern, Germany
http://home.math.au.dk/jensen/

Abstract. Mixed volumes of lattice polytopes play a central role in numerical and tropical algebraic geometry. We present an implementation of a new algorithm for their computation based on tropical homotopy continuation, which is a combinatorial procedure using ideas from numerical algebraic geometry. While the mathematical aspects of the algorithm are presented elsewhere, here we mainly address technical details of the implementation, in particular how it was made fast and reliable. The implementation is distributed as part of the library gfanlib.

Keywords: Tropical geometry, mixed volumes, numerical algebraic geometry

1 Introduction

Given n convex bounded sets $P_1, \ldots, P_n \subseteq \mathbb{R}^n$ the function $f : \mathbb{R}^n_{\geq 0} \to \mathbb{R}_{\geq 0}$ given by $(\lambda_1, \ldots, \lambda_n) \mapsto \text{Volume}_n(\lambda_1 P_1 + \cdots + \lambda_n P_n)$ turns out to be a polynomial function and the coefficient of $\lambda_1 \cdots \lambda_n$ in the corresponding polynomial is called the *mixed volume* of $P_1, \ldots, P_n$. Mixed volumes were studied by Minkowski, but due to their appearance in the BKK Theorem, they have in recent years played important roles in enumerative, tropical and numerical algebraic geometry.

Theorem 1 (Bernstein, Khovanskii, Kushnirenko, 1975). *Let $f_1, \ldots, f_n \in \mathbb{C}[x_1, \ldots, x_n]$ be polynomials. Then the number of isolated solutions to the polynomial system $f_1(x) = \cdots = f_n(x) = 0$ with $(x_1, \ldots, x_n) \in (\mathbb{C} \setminus \{0\})^n$ is (counting multiplicities) bounded by the mixed volume of the Newton polytopes of $f_1, \ldots, f_n$.*

There have been many attempts to implement effective[1] algorithms for mixed volume computation, i.e. [6–8]. Indeed the available software is capable of computing mixed volumes of systems whose numerical solutions cannot be determined — simply because there are too many. Thus for the immediate purpose of solving, faster algorithms may not seem particularly important. However, for the human interaction process, being able to quickly determine mixed volumes will allow choosing the right approach more quickly. There are other cases where speed of mixed volume computation is important. For example, when we don't need one big, but rather many small mixed volumes, such as when we wish to

[1] Mixed volume computation, just as volume computation, is #P-hard [2].

```
****************** CLING ******************
* Type C++ code and press enter to run it *
*             Type .q to exit             *
*******************************************
[cling]$ #include "gfanlib_circuittableint.cpp"
[cling]$ #include "gfanlib_paralleltraverser.cpp"
[cling]$ #include "gfanlib_mixedvolume.cpp"
[cling]$ using namespace std;
[cling]$ using namespace gfan;
[cling]$ using namespace gfan::MixedVolumeExamples;
[cling]$ auto s={cyclic(2),cyclic(3),cyclic(5),cyclic(7)};
[cling]$ for(auto v:s)cout<<mixedVolume(v)<<" ";cout<<endl;
2 6 70 924
[cling]$ .q
```

Fig. 1. A session with the C++ "interpreter" CLING demonstrating how we can turn gfanlib into a simple interactive computer algebra system capable of computing mixed volume. We ran CLING with options `-l /usr/lib/libgmp.so.3.5.2 -std=c++0x`.

determine the multiplicities in a stable intersection of a set of tropical hypersurfaces.

Correctness is another issue. In [6] the authors write:

> "However, for many polynomial systems in real applications, listed in Table B1 and Table B2 in §4, MixedVol-2.0 produced the same mixed volume for all different sets of random liftings, whereas DEMiCs failed to provide a unique mixed volume with respect to different liftings."

comparing the implementation in [6] to that of [8]. However, it seems that there has been no serious attempt in recent implementations to eliminate the possibility of round-off errors. This may in practise not be a problem for many systems, but it is a problem if the software is used in a mathematical proof.

Finally, memory usage is an issue – not so much because we will run out of RAM on conventional machines, but rather because memory accesses slow down computation and limits what exotic architectures our software may run on.

We address these three issues with our new implementation. Distributed as part of the C++ library gfanlib [4], other systems can benefit from it. How to use the library is demonstrated in Figure 1 and 2. The algorithm itself, which uses no linear programming, was proposed at MEGA 2015 and is a variant of the algorithm in [7]. The mathematical details are described in [5], while we here address some software engineering aspects. We hope that a general audience will find the discussion interesting.

2 A brief description of the algorithm

We briefly describe the tropical homotopy approach to finding mixed volumes. The main object of study is overlays of tropical hypersurfaces.

```
#include <iostream>
#include "gfanlib.h"
using namespace std;
using namespace gfan;

int main(){
    try{
        cout << mixedVolume(MixedVolumeExamples::cyclic(12),16) << endl;
        cout << mixedVolume(MixedVolumeExamples::gaukwa(7),8) << endl;
    }
    catch (...){
        cerr << "Error - most likely an integer overflow." << endl;
        return 1;
    }
    return 0;
}
```

Fig. 2. A sample C++ program demonstrating how to run the mixed volume computation with 16 or 8 threads and catch possible exceptions.

Definition 1. *Given a matrix $A \in \mathbb{Z}^{n\times m}$ and a vector $\omega \in \mathbb{R}^m$, the tropical hypersurface $T(A,\omega)$ is defined as*

$$T(A,\omega) := \{x \in \mathbb{R}^n : \max_i(\omega_i + A_{1i}x_1 + \cdots + A_{ni}x_n) \text{ is attained at least twice}\}.$$

A tropical hypersurface is a polyhedral complex of codimension 1. We will consider n such hypersurfaces with a total of $m = m_1 + \cdots + m_n$ terms. We let $\mathcal{A} = (A_1, \ldots, A_n)$ be the tuple of matrices with $A_i \in \mathbb{Z}^{n\times m_i}$ and $\omega \in \mathbb{R}^m = \mathbb{R}^{m_1} \times \cdots \times \mathbb{R}^{m_n}$ be the concatenated vector of coefficients.

Example 1. The tropical hypersurfaces defined by $\omega = ((0,-1,0,0),(0,0,1,0))$ and

$$\mathcal{A} = \left(\begin{pmatrix} 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 1 & 2 \\ 0 & 1 & 1 & 1 \end{pmatrix}\right)$$

are shown in Figure 3 (left). The right picture is for $\omega' = ((0,-1,0,\frac{3}{2}),(0,0,1,0))$.

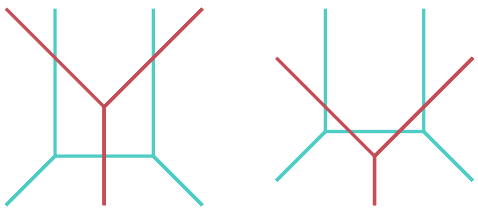

Fig. 3. Different overlays of tropical hypersurfaces defined by the same pair of matrices. As coefficients change in the tropical homotopy one intersection point splits into two.

In the case of generic coefficients, given an intersection point, for each hypersurface, the maximum in Definition 1 is attained exactly twice. That is, to each intersection point p we may associate a tuple $M = ((a_1, b_1), \ldots, (a_n, b_n)) \in \{1, \ldots, m_1\}^2 \times \cdots \times \{1, \ldots, m_n\}^2$ containing the indices to the columns of the A_i where the maximum was attained. We call M the mixed cell dual to p.

The purpose of the tropical homotopy algorithm is to keep track of the mixed cells as ω changes along a straight line ℓ towards some $\omega' \in \mathbb{R}^m$. To prevent ℓ from passing through low dimensional cones, ω is perturbed symbolically, by letting $\omega \in (\mathbb{R}(\varepsilon))^m$ with $\varepsilon > 0$ symbolic and small. We give the specifications:

Algorithm 2 (Tropical homotopy)
Input: *A tuple $\mathcal{A}$, coefficients ω and ω' and the mixed cells dual to $\cap_i T(A_i, \omega_i)$.*
Output: *The mixed cells dual to $\cap_i T(A_i, \omega_i' + \varepsilon\omega_i)$.*

2.1 A mixed cell cone

The tropical homotopy attempts to treat each mixed cell independently. Therefore it is relevant to know the set of ω's for which a particular cell appears.

Theorem 3. *The set C_M of all $\omega \in \mathbb{R}^m$ giving rise to a particular M appearing as a dual mixed cell is an m-dimensional cone with $m - 2n$ facets.*

The facet normals are in the null space of the *Cayley matrix* of $\mathcal{A}$. In Example 1

$$\text{Cayley}(A_1, A_2) = \begin{pmatrix} 0 & 1 & 2 & 1 & 1 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

and the four marked columns constitute the center mixed cell in Figure 3 (left). Each of the other columns together with those indexed by M forms a matrix of rank $2n$. Each non-zero vector v in the null space is a facet normal of C_M. In the indicated case $v = (1, 0, 1, -2, 0, -2, 0, 2, 0)$, and we may check that v separates ω and ω'. What happens to M after ω crosses the facet is encoded in v.

We may assign multiplicities to each intersection point, namely the absolute value of the determinant of the submatrix of the Cayley matrix indexed by the mixed cell. The multiplicities happen to sum to $\text{MixVol}(\text{New}(f_1), \ldots, \text{New}(f_n))$. Consequently, the mixed volume can be found by finding any generic tropical intersection $\bigcap_i T(A_i, \omega_i)$. Next we discuss how to build up such intersection.

2.2 Tropical regeneration

In numerical regeneration [3], to solve $f_1 = \cdots = f_n = 0$, one starts with the solution to a linear system consisting of n equations in n unknowns. Successively each linear equation is first replaced by a product of linear equations and thereafter the desired f_i. During this process, the solutions to the intermediate systems are tracked. The tropical regeneration process is its *tropicalisation*.

We start with the configuration $(L_1, \ldots, L_n)$ where the L_i denote $n \times (n+1)$ matrices with a zero vector and the standard basis vectors as columns. Then we replace L_1 by the configuration B_1 which is just L_1 scaled so that the columns of A_1 are in the convex hull of the columns of B. Solutions to a system with matrices

$$((A_1, B_1), L_2, \ldots, L_n)$$

are known for coordinates of the coefficients associated to A_1 being very small. As we instead make coefficients of L_1 small via tropical homotopy, we obtain a set of mixed cells for the tuple (after deleting solutions going to infinity):

$$(A_1, L_2, \ldots, L_n).$$

Repeating this process for the other A_i we eventually reach the set of solutions to a system with exponent matrices $(A_1, \ldots, A_n)$. The mixed volume is obtained by summing the multiplicities, i.e. the absolute values of the determinants.

Starting out from one solution to the linear system, the solution point splits up into many that may later join, go to infinity or split up further as the regeneration proceeds. Using reverse search [1], the graph of all these homotopy paths may be turned into a tree, allowing easy organisation of a memoryless traversal.

3 Technical implementation details

A comparison to other implementations can be found in Figure 4. Note that those are running on different hardware and using different algorithms, with the algorithm of [7] begin closest to ours. Our timings are roughly 50 times faster than the first arXiv.org version of [5]. Moreover, the new version achieves this while still catching overflows, should any arise. In this section we will describe how this performance gain was obtained and address a few other technical details.

Profiling To improve performance of software it is important to measure any possible progress. We found the Linux command line tool `perf` convenient, alternatingly doing `perf record ./a.out` and `perf report` to find the hotspots.

Example	n	Mixed volume	1 thread	16 threads	Ratio	[7] 8 threads	[6] 1 thread
Cyclic-15	15	35243520	461.3	35.8	12.9	4070	36428
Noon-20	20	3486784361	59.0	4.8	12.4	6460	1109
Chandra-21	21	1048576	151.6	11.5	13.2	7580	1067
Katsura-17	18	131070	4.5	0.5	9.5	5310	75619
Eco-22	22	1048576	102.7	8.5	12.1	8750	

Fig. 4. Timings in seconds for our implementation compared to timings reported in [7] and [6]. Our timings are for a dual Intel Xeon E2670 CPU system, while a "2.4GHz Intel Core 2 Quad CPU" was used in [6] and a "SGI Altix ICE 8400" system in [7]. For Gaukwa-7 our implementation had integer overflows, while [7] and [6] had no problems.

3.1 The circuit table

Given a tuple $(A_1, \ldots, A_n)$ and a mixed cell $M = ((a_1, b_1), \ldots, (a_n, b_n))$, we may use Theorem 3 to find out which circuits of the Cayley matrix give the irredundant inequalities of C_M. The tuple M selects columns of the Cayley matrix that form an invertible $(2n) \times (2n)$ matrix D. Our first attempt was to keep D^{-1} around to easily produce the desired circuits. The drawback of this approach is that D need not be invertible over $\mathbb{Z}$ and we had to resolve to floating point arithmetic, cast results to integers and verify them in exact arithmetic – a costly procedure.

The solution is to not store D^{-1}, but instead update the circuits as M changes.

Definition 2. *Given $\mathcal{A}$ and M, their circuit table is an $m \times m$ matrix, with each row indexed by a $a \notin M$ containing a non-zero kernel vector of the Cayley matrix of $\mathcal{A}$ with support contained in $M \cup \{a\}$.*

This leaves $2n$ rows of the circuit matrix undefined. We will fix the scaling of each row by letting its coordinates be the maximal minors of the $2n \times (2n+1)$ submatrix of $\text{Cayley}(A_1, \ldots, A_n)$ indexed by $M \cup \{a\}$. Therefore our rows are not circuits in the strict sense, as they need not be primitive.

The most important feature of the circuit matrix is that if we substitute an entry of M, it is possible to keep the matrix updated using integer row operations, i.e. cancelling out a particular coordinate common between two circuits.

Cache-friendly memory layout While matrices on which we want to do row operations are conveniently stored row-wise in memory, the circuit matrix is quite sparse. In a typical situation with $n = 10$ it may be of size 100×100, while having at most $2n + 1 = 21$ non-zero entries per row. Moreover, most of the entries appear in pairs with opposite sign due to the last n rows of the Cayley matrix. As all but one entry of a row appears in a column indexed by M, we may therefore store the matrix compactly as a 100×10 matrix and an additional vector of entries. It however turns out that many of the operations required to update the circuit matrix can conveniently and cache-friendly be performed as multiply-add row operations of the packed 10×100 transposed matrix of type

$$\text{row}_i := \frac{1}{\gamma}(\alpha \text{row}_i + \beta \text{row}_j) \tag{1}$$

where α, β and γ are integers. While our implementation treat generic integer implementations, we should imagine that the entries above are 32 bit ints.

Eliminating integer divisions The update (1) above is done roughly for every row of the packed circuit matrix and requires an exact division for each entry. This is costly compared to the addition and multiplications. A common trick for low-level optimisation is to replace the division by multiplication. If γ is odd, then $\gcd(\gamma, 2^{32}) = 1$ and therefore $[\gamma]$ has a multiplicative inverse in $\mathbb{Z}/2^{32}\mathbb{Z}$.

On the other hand, if γ is even, we can reduce to the odd case by binary right shifts. The γ remains constant for the update of the full matrix as it happens to be the volume of the mixed cell in question, i.e. the determinant of D.

Vectorisation In the update (1) each entry of a row gets the same treatment. This allows the use of the by now common 128 bit SIMD vector instructions. They can do 4 integer operations simultaneously. While we use a highlevel C++ class to encapsulate matrix data, to use the vector instructions we need to pass the restrict keyword, telling for example that the data pointed to by a, respectively b, can only be accessed through a, respectively b:

```
void muladd(int* __restrict__ a, int* __restrict__ b, int alph, int len);
```

Passing compiler options `-O3 -mavx -msse2 -finline-limit=1000`, gcc will emit the desired SIMD instructions. It is possible that the code would be more efficient if we forced data to be aligned at 128 bit boundaries and `len` to be divisible by 4. However, as we would like the code to automatically exploit 256 bit instructions in the future we decided not to make the code 128 bit specific.

Overflow checking If we know bounds on the entries of the ith and jth rows in (1) then we may be able to conclude that the operation does not overflow. However, a bound would have to be computed again for the next update. Therefore we compute the minimum and maximum entry of the new row_i in the same loop that performs (1) and store the results for later. The code will translate into two vector instructions (max and min) per loop iteration.

In the implementation there are actually four different loops for doing (1) depending on the bounds on the arguments and whether a shift is needed. In the worst case the operation needs to be done in 64 bit to catch the overflow. To actually perform this checking without overflows, we put further restrictions on the entries of $\mathcal{A}$ and ω' (must fit in 16 bit) and on m ($m < 2^{15}$). These restrictions are checked automatically in the initialisation step of the algorithm.

3.2 Templates

Our description above has mainly considered the case where the integers are of 32 bit. With the use of C++ templates, we have written the code for updating the packed circuit table generically, so that the type can be replaced at compile time. For example one may choose to make an integer type with arbitrary precision, which should be called in case of an overflow.

3.3 Parallelisation

The key to easy parallelisation on multicore architectures is abstraction. We make a class representing the concept of an enumeration tree traverser. It will be located at a vertex of the tree and has a simple interface for asking for the number of children and moving up and down in the tree. With this class

at hand we make, say 16, instances of such objects and hand them and the actual parallelisation over to an abstract tree traversal library. Such library was contributed to the gfan project by Bjarne Knudsen. It uses the C++11 standard for dealing with threads.

The only other issue to take care of for good performance is not stressing the memory allocator, which means that matrices whose sizes are known at initialisation are never reallocated. With our 16 core machine we obtain speedups of a factor 13.5 in the best cases. This is a reasonable speedup as the machine clocks down by roughly 10 percent when many cores are in use.

3.4 Exceptions

We use the C++ exceptions to handle issues with overflows. If a subroutine called by the enumeration graph traverser overflows, it will throw an exception. We do not count on C++ handling exceptions across threads and therefore catch the exception in the abstract traverser, which will then make sure that the computation is aborted in all threads. The library then throws a new exception indicating that an overflow occurred. We then have the option to redo the computation with higher precision.

For this strategy to work it is important that our code is exception safe. To be that, the most important feature is that we design our classes so that allocated memory is freed in destructors following the RAII principle (Resource Acquisition Is Initialization). In particular we avoid `malloc` and `new` but use STL containers instead (Standard Template Library).

References

1. David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Appl. Math.*, 65(1-3):21–46, 1996. First International Colloquium on Graphs and Optimization (GOI), 1992 (Grimentz).
2. Martin Dyer, Peter Gritzmann, and Alexander Hufnagel. On The Complexity of Computing Mixed Volumes. *SIAM Journal on Computing*, 27(2):356–400, 1998.
3. Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler. Regeneration homotopies for solving systems of polynomials. *Math. Comp.*, 80(273):345–377, January 2011.
4. Anders Nedergaard Jensen. Gfan, a software system for Gröbner fans. Available at `http://home.imf.au.dk/ajensen/software/gfan/gfan.html`.
5. Anders Nedergaard Jensen. Tropical homotopy continuation, 2016. arXiv:1601.02818.
6. Tsung-Lin Lee and Tien-Yien Li. Mixed volume computation in solving polynomial systems. *Contemporary Mathematics*, 556:97–112, 2011.
7. Gregorio Malajovich. Computing mixed volume and all mixed cells in quermassintegral time. 2014, arXiv:1412.0480.
8. Tomohiko Mizutani, Akiko Takeda, and Masakazu Kojima. Dynamic enumeration of all mixed cells. *Discrete & Computational Geometry*, 37(3):351–367, 2007.