Computing with an algebraic-perturbation variant of Barvinok’s algorithm

Jon Lee\textsuperscript{a,}\textsuperscript{*}, Daphne Skipper\textsuperscript{b}

\textsuperscript{a}Department of I&OE, University of Michigan, Ann Arbor, MI
\textsuperscript{b}Department of Mathematics, Augusta University, Augusta, GA

Abstract

We present a variant of Barvinok’s algorithm for computing a short rational generating function for the integer points in a nonempty pointed polyhedron

\[ P := \{ x \in \mathbb{R}^n : Ax \leq b \} \]

given by rational inequalities. A main use of such a rational generating function is to count the number of integer points in \( P \).

Our variant is based on making an algebraic perturbation of the right-hand side \( b \in \mathbb{Q}^m \) of the inequalities, replacing each \( b_i \) with \( b_i + \tau^i \), where \( \tau \) is considered to be an arbitrarily small positive real indeterminate. Hence, elements of our right-hand side vector become elements of the ordered ring \( \mathbb{Q}[\tau] \) of polynomials in \( \tau \). Denoting the algebraically-perturbed polyhedron as \( P(\tau) \subset \mathbb{R}[\tau]^n \), we readily see that: (i) \( P(\tau) \) is always full dimensional, (ii) \( P(\tau) \) is always simple, and (iii) \( P(\tau) \) contains the same integer points as \( P \). Unlike other versions of Barvinok’s algorithm, we will have to do some arithmetic in \( \mathbb{Q}[\tau] \). However, because of (i) we will not need to preprocess our input polyhedron if it is not full dimensional, and because of (ii) we will not need to triangulate tangent cones at the vertices of the polyhedron.

We give the details of our perturbation variant of Barvinok’s algorithm, describe a proof-of-concept implementation developed in \textit{Mathematica}, and present results of computational experiments.

Introduction

Let \( P := \{ x \in \mathbb{R}^n : Ax \leq b \} \) be a nonempty pointed polyhedron given by \( m \) rational inequalities. In celebrated work, Barvinok gave an algorithm,
which is polynomial time when the dimension $n$ is fixed, for computing a short rational generating function for the integer points in $P$ (see [1]). One main use of such a short rational generating function is to count the number of integer points in $P$. This of course also determines whether or not $P$ contains an integer point — a key concern in integer-linear optimization (for both checking integer feasibility of an inequality system and for verifying optimality of a feasible candidate solution).

Throughout, our viewpoint is that we are thinking about what is efficient/practical when the dimension $n$ is thought of as fixed, and the rational data $A$ and $b$ and number of constraints $m$ can be considered varying.

Some familiarity with Barvinok’s algorithm will help in appreciating our variant, but it is still possible to follow the presentation as a first view of how Barvinok’s algorithm works. A detailed and fairly recent exposition of the underlying theory is [2] (also see the older but still very nice survey [3]). Another entry point is the recent book by De Loera, Hemmecke and Köppe; see their description of Algorithm 7.1 (primal partially-open variant of Barvinok’s algorithm; [4, p. 134]). Their variant “contains algorithmic refinements upon Barvinok’s original algorithm, which improves the theoretical and practical efficiency of the algorithm”.

We note that similar ideas to ours have been described, for example, in [5, 6, 7, 8, 9]. Using various combinations of cone polarization and perturbation, different algorithms can be developed. We emphasize that our perturbation framework, while mathematically equivalent to triangulating the polar cones (of tangent cones) and then polarizing the child cones, leads to a significantly-different concrete and general computational paradigm. In contrast to some other works, we do not make an explicit rational perturbation, but rather we proceed symbolically. Furthermore, we do not explicitly polarize or triangulate any cones. We do not claim that any one version of Barvinok’s algorithm is best. Our goal is mainly to computationally demonstrate the viability of our approach.

In §1 we describe our perturbation variant of Barvinok’s algorithm. In §2 we describe a proof-of-concept implementation using Mathematica. In §3 we report on a numerical example, exhibiting how the algorithm behaves on an example that has degeneracy and is not full dimensional. In §4 we report on computational results obtained on a set of 30 new test instances. For a non full-dimensional instance, under our perturbation, many vertices have closely-related tangent cones — differing only by multiplying some generators by $-1$. In §5 we report on some computational experiments on algorithmic
improvements related to this observation. In §6 we mention a few possible extensions of our work and make some concluding remarks.

A preliminary report of our work, without any description of an implementation, nor examples, nor computational results, appeared as [10].

1. Perturbation variant

Before presenting our algorithm, we describe the setting to which we are adapting Barvinok’s algorithm. Next, we review the geometric results (concerning indicator functions of polyhedra and generating functions for the contained integer points) that Barvinok’s algorithm rests upon. Then, we present each step of our perturbation variant, as we describe its adaptation from a vanilla Barvinok algorithm.

1.1. The perturbation

We make an algebraic perturbation of the right-hand side \( b \in \mathbb{Q}^m \) of the inequalities, replacing each \( b_i \) with \( b_i + \tau_i \), for \( i = 1, 2, \ldots, m \), where \( \tau \) is an arbitrarily small positive real indeterminate. Hence, components of \( b \) become elements of the ordered ring \( \mathbb{Q}[\tau] \) of polynomials in \( \tau \) with rational coefficients. The ordering of polynomials in \( \mathbb{Q}[\tau] \) is the usual one: 

\[
p(\tau) := \sum_{l=1}^{k} p_l \tau^l <_\tau q(\tau) := \sum_{l=1}^{k} q_l \tau^l
\]

if the least \( l \) for which \( p_l \neq q_l \) has \( p_l < q_l \). Denoting the algebraically-perturbed polyhedron as \( P(\tau) := \{ x \in \mathbb{R}^n : Ax \leq \tau b + \vec{\tau} \} \), where \( \vec{\tau} := (\tau, \tau^2, \ldots, \tau^m) \), it is easy to see that:

(i) \( P(\tau) \) is always full dimensional,
(ii) \( P(\tau) \) is always simple, and
(iii) \( P(\tau) \) contains the same integer points as \( P \).

Regarding (i), we assume that \( P \) is nonempty, so there is an \( \hat{x} \in P \). Therefore, \( A_i \hat{x} < b_i + \tau_i \), for \( i = 1, 2, \ldots, m \). So, \( P(\tau) \) is full dimensional.

Regarding (ii), a vertex of \( P(\tau) \) is the unique solution \( v \in \mathbb{R}[\tau]^n \) of

\[
A_l v = b_l + \tau^l, \quad \text{for} \ l \in \beta,
\]

where \( A_l \) is the \( l^{\text{th}} \) row of \( A \), \( \beta \subset \{1, 2, \ldots, m\} \) and \( |\beta| = n \), provided that such a unique solution exists and that it is in \( P(\tau) \). Note that the components of \( v \) are polynomials in \( \tau \) and that all components have no terms \( \tau^j \) with nonzero coefficient having \( j \notin \beta \). Therefore, \( A_j v \neq b_j + \tau^j \) for \( j \notin \beta \). Hence, \( P(\tau) \) is always simple. Furthermore, \( v \in \mathbb{Q}[\tau]^n \) when \( A \) and \( b \) are rational.
Regarding (iii), we wish to emphasize that the set of integers in \( \mathbb{R}[\tau] \) is precisely the ordinary set of integers \( \mathbb{Z} \).

Although we consider \( \tau \) to be an arbitrarily small positive \textit{real indeterminate}, statements (i-iii) above, and the comments following them, can be interpreted geometrically in \( \mathbb{R}^n \) as holding for every sufficiently small positive \textit{real number}. Nonetheless, in our algorithm and implementation, we work with \( \tau \) as an indeterminate.

Unlike other versions of Barvinok’s algorithm, we will have to do some arithmetic in \( \mathbb{Q}[\tau] \). However, because of (i) we will not need to preprocess our input polyhedron if it is not full dimensional, and because of (ii) we will not need to triangulate tangent cones at the vertices of the polyhedron.

We note that this kind of algebraic perturbation of \( b \) is one possible way of interpreting the “lexicographic tie-breaking rule” for the simplex algorithm of linear optimization (see [11], for example, where this kind of interpretation is made explicit). In fact, algebraic perturbation has a very general place in breaking degeneracy in geometric and algebraic algorithms (see [12]).

1.2. The algorithm

Before getting started, we lay out the fundamental geometric results that Barvinok’s algorithm is based on. In what follows, \([\cdot]\) is the indicator function, and \(\text{tcone}(P,F)\) denotes the tangent cone of polyhedron \(P\) from a face \(F\).

**Theorem 1** (Brianchon [13], Gram [14]). Let \(P \subset \mathbb{R}^n\) be a polyhedron. Then

\[
[P] = \sum \{(-1)^{\dim F}[\text{tcone}(P,F)] : F \text{ is a nonempty face of } P \}. 
\]

All of the summands for non-vertex faces of \(P\) are indicator functions of non-pointed (tangent) cones. Suppressing those terms, we have

**Corollary 2** (Brion’s Theorem). Let \(P \subset \mathbb{R}^n\) be a polyhedron. Then

\[
[P] \equiv \sum \{[\text{tcone}(P,v)] : v \text{ is a vertex of } P \} 
\]

(modulo indicator functions of non-pointed cones).

While this does not give us an exact identity for indicator functions, we do obtain an exact identity for the associated generating functions. We use the standard monomial notation that for an \(n\)-vector of variables \(z\) and an
n-vector of integers \( a \), we write \( z^a := \prod_{i=1}^{n} z_i^{a_i} \). The generating function for the integer points of \( P \) is defined as

\[
g(P; z) := \sum_{a \in P \cap \mathbb{Z}^n} z^a.
\]

**Theorem 3** (Brion’s Theorem for generating functions). Let \( P \subset \mathbb{R}^n \) be a polyhedron. Then

\[
g(P; z) = \sum \{ g(tcone(P, v); z) : v \text{ is a vertex of } P \}.
\]

We do not seek to calculate an exact formula for \([P]\). Rather, we want a representation of the generating function \( g(P; z) \) as a short rational function, and this is where Barvinok comes in.

**APB: Algebraic-Perturbation Barvinok**

1: **input** a rational pointed polyhedron \( P := \{ x \in \mathbb{R}^n : Ax \leq b \} \).

2: **output** a rational generating function for \( P \cap \mathbb{Z}^n \) in the form

\[
g(P; z) := \sum_{i \in I} \epsilon_i \sum_{a \in A_i} z^a \prod_{j=1}^{n} (1 - z^{b_{ij}}),
\]

with \( \epsilon_i \in \{\pm 1\} \), \( b_{ij} \in \mathbb{Z}^n \), and \( A_i \subset \mathbb{Z}^n \) for \( i \in I \), where \( I \) and the \( A_i \) are finite.

3: **compute** all vertices \( v \) and corresponding tangent cones \( C \) of \( P(\tau) \). Because \( P(\tau) \) is simple, each tangent cone \( C \) is simplicial in our setting. We organize this information as a set \( \mathcal{T} \) of triples \( (C, v_C, \epsilon_C) \) (that we will update). In each triple, \( C \) is a pointed (simplicial) cone rooted at a vertex \( v_C \) of \( P(\tau) \), and we store an associated sign \( \epsilon_C \in \{\pm 1\} \). Initially, we let

\[
\mathcal{T} := \{ (C, v, +1) : v \text{ is a vertex of } P(\tau), \text{ and } C \text{ is the associated (simplicial) tangent cone of } P(\tau) \}.
\]

To compute the vertices, we can simply check all possible \( \binom{m}{n} = O(m^n) \) choices of sets of \( n \) basic rows from \( A \) (recall our assumption that \( n \) is fixed). If \( v \in \mathbb{Q}[\tau]^n \) is the unique solution \( x \in \mathbb{R}[\tau]^n \) of

\[
A_l x = b_l + \tau^l, \text{ for } l \in \beta,
\]

5
then the simplicial cone $C$ is the solution set in $\mathbb{R}[\tau]^n$ of
\[
A_l x \leq 0, \text{ for } l \in \beta.
\]

We emphasize that each $C$ is rational — i.e., its generators are in $\mathbb{Q}^n$.

4: the next step would normally be to triangulate each of the tangent cones $C$, but in our setting each $C$ is simplicial, so there is nothing to do here. Of course, our perturbation created more vertices for $P(\tau)$ than for $P$, effectively perturbing non-simple vertices to create many simple vertices. So we do not have to triangulate tangent cones, but we have more of them up front.

5: the index of a simplicial rational cone $C := \text{cone}(\{h_1, h_2, \ldots, h_n\})$, where each $h_i$ is a primitive vector of the integer lattice $\mathbb{Z}^n$ and $H := [h_1, h_2, \ldots, h_n]$ is $\text{ind } C := |\det H|$. In this step, we apply “signed decomposition”, possibly repeatedly, to any “high-index” cone $C$ from a triple $(C, v_C, \epsilon_C)$ on the list $T$, replacing the associated triple with $\leq n$ child triples (all with the same vertex $v$ as the mother) — we need to give a recipe for calculating (the signs and cones of) the children. The least index of a cone is 1 (a unimodular cone), but we may be satisfied with cones with index below some threshold.

One iteration of Barvinok’s signed decomposition (see [4, p. 132]) works as follows on a triple $(C, v_C, \epsilon_C)$: we decompose the simplicial rational cone $C := \text{cone}(\{h_1, h_2, \ldots, h_n\})$, based on a a nonzero point $w \in \mathbb{Z}^n$ such that $\text{cone}(\{h_1, h_2, \ldots, h_n, w\})$ is pointed. Let $C_j := \text{cone}(\{h_1, h_2, \ldots, h_{j-1}, w, h_{j+1}, \ldots, h_n\})$, Then there exist $\epsilon_j \in \{0, \pm 1\}$, such that $[C] \equiv \sum_{j=1}^{n} \epsilon_j[C_j]$.

$\epsilon_j = 0$ if $C_j$ is not full dimensional.
We need to explain how to get $w$ and the $\epsilon_j$. We will get $w$ by finding a $u$ related to a short nonzero vector in a particular lattice. Then we let

$$w := Hu.$$  

If

$$\text{cone}(\{h_1, h_2, \ldots, h_n, w\})$$

is not pointed, then we simply substitute $w$ with $-w$, so it is easy to get the pointedness to hold.

How to compute the $\epsilon_j \in \{0, \pm 1\}$ is not completely explained in [4], but it is actually quite easy. We let $\epsilon_j := \text{sign}(u_j)$, for $j = 1, 2, \ldots, n$.

We have essentially specified how to get everything needed. Now we replace the mother $(C, v_C, \epsilon_C)$ with the set of children

$$\{ (C_j, v_C, \epsilon_C \cdot \epsilon_j) : \epsilon_j \neq 0 \}.$$

Finally, we explain how we get $u$. The index of the cone $C_j$ is

$$\lambda_j := \det[h_1, h_2, \ldots, h_{j-1}, w, h_{j+1}, \ldots, h_n],$$

for $j = 1, 2, \ldots, n$. Employing Cramer’s rule, we have $u = \lambda / \det H$, and so

$$\lambda = \text{adj}H \cdot w,$$

where $\text{adj}H := \det H \cdot H^{-1}$ is the adjugate (or classical adjoint, or transposed matrix of cofactors) of $H$. Note that $\text{adj}H$ is integer when $H$ is.

So $\lambda$ is a nonzero vector in the lattice $L := \text{adj}H \cdot \mathbb{Z}^n$. Our goal is to have the cones $C_j$ all have low index. So it is natural to choose $\lambda$ to be a short nonzero vector in the lattice $L$. Any norm will do, as long as we can reliably get within a constant factor of the shortest. So employing the $\ell_2$-norm and using the LLL algorithm (see [15], for example) is a natural choice. From $\lambda$ we get $u = \lambda / \det H$, and from $u$ we get $w = Hu$.

We end up with our list of triples $T$ having all cones with low index. We also have

$$g(P; z) = \sum_{(C, v_C, \epsilon_C) \in T} \epsilon_C \cdot g(v_C + C; z),$$

starting from Brion’s theorem for generating functions, and (repeatedly) substituting according to Barvinok’s signed decomposition, as we refine $T$.  

7
6: now, for each \((C, v, \cdot) \in T\), suppose that the simplicial cone \(C\) has the representation \(C := \{Hu : u \in \mathbb{R}^n_+\}\), with \(H \in \mathbb{Z}^{n \times n}\) and the columns of \(H\) being primitive integer vectors. We also have the associated point \(v \in \mathbb{Q}[\tau]^n\). We define the semi-open parallelepiped 
\[
\Pi(v, H) := v + \{Hu : 0 \leq u < e\} \subset \mathbb{R}[\tau]^n.
\]
We want to enumerate the integer points in \(\Pi(v, H)\). Consider the Smith Normal Form of \(H\). That is, \(HU = WS\), where \(U\) and \(W\) are integer unimodular matrices and \(S = \text{diag}(s)\) is a diagonal integer matrix. Then 
\[
\Pi(v, H) \cap \mathbb{Z}^n = \{Ww - H[-H^{-1}(v - Ww)] : 0 \leq w < s, \ w \in \mathbb{Z}^n\}
\]
(cf. [16, Lemma 11]). Note that the round-down function \(\lfloor \cdot \rfloor\) is applied to polynomials in \(\tau\), since \(v \in \mathbb{R}[\tau]^n\). We have no difficulty in applying \(\lfloor \cdot \rfloor\), as the ordered ring \(\mathbb{Z}\) is a subset of the ordered ring \(\mathbb{R}[\tau]\). It is worth emphasizing that finally \(\Pi(v, H) \cap \mathbb{Z}^n\) is a finite set of integer points — no presence of \(\tau\).

7: write down the formula of step 2.
\[
g(P; z) := \sum_{(C, v_C, \epsilon_C) \in T} \epsilon_C \cdot \sum_{a \in \Pi(v_C, H^C) \cap \mathbb{Z}^n} z^a \frac{1}{\prod_{j=1}^n (1 - z^{h^C_j^a})},
\]
where \(H^C := [h^C_1, h^C_2, \ldots, h^C_n]\) is the matrix with columns generating \(C\). We have really calculated \(g(P(\tau); z)\), but observing that finally this expression does not depend on \(\tau\), we have as well calculated \(g(P; z)\).

— End APB Algorithm —

**Theorem 4.** APB runs in polynomial time, in fixed dimension \(n\).

**Proof.** The proof rests on the proof that Barvinok’s algorithm runs in polynomial time, in fixed dimension \(n\). We simply also observe that for the arithmetic that we do in \(\mathbb{R}[\tau]\), all of the polynomials arising have at most \(n\) nonzero terms, and the coefficients are (by Cramer’s rule) rational numbers having numerators and denominators bounded in bit size by a polynomial in the sizes of numbers in the input matrix \(A\). \(\square\)
2. Proof-of-concept implementation

Implementations of versions of Barvinok’s algorithm were available soon after the algorithm was announced (see [17]). With the goal of developing a platform for convenient experimentation and extension, rather than to build the fastest implementation possible, we implemented our APB algorithm with Mathematica, version 10.0 [18]. One advantage of using Mathematica is its huge user community and low barrier to entry. In fact, Mathematica is bundled with the “Raspberry Pi” — “a low cost [$35], credit-card sized computer that plugs into a computer monitor or TV, and uses a standard keyboard and mouse” (see [19]). Our use of Mathematica means for many people: immediate ease of use, integration and modification. Furthermore, as Mathematica functions used by our implementation of APB are improved, our implementation immediately benefits.

On the negative side, Mathematica functions such as LatticeReduce are opaque to and cannot be modified by users (see [20] for comments about this issue and related remarks about the hazards of relying on such computer-algebra systems). Therefore a reasonable alternative could be to work instead with the open-source alternative Sage.

Because, of course, our input data is rational, all of our arithmetic in \( \mathbb{R} \) (respectively, \( \mathbb{R}[\tau] \)) is really done in \( \mathbb{Q} \) (\( \mathbb{Q}[\tau] \)) with exact arithmetic. Crucially, checking if one polynomial is less-than-or-equal to another in \( \mathbb{Q}[\tau] \), is easily accomplished in Mathematica:

(* Return True if p1 <= p2 *)
(* p1 and p2 are polynomials in t *)

PolyLEQ[p1_, p2_] := Module[{difference},
  difference = CoefficientList[p1 - p2, t];
  If[difference == {}, True,
    If[Drop[difference, LengthWhile[difference, # == 0 &]][[1]] < 0,
      True, False]
  ];
]

Additionally, we take advantage of several Mathematica functions:

- The function IntegerSmithNormalForm (available as a downloadable Mathematica package) calculates the Smith normal form of an integer matrix \( A \) (employed in step 7). See [21].
The Mathematica function \texttt{LatticeReduce} seeks a \emph{reduced basis} (in the sense of Lovász) of a lattice specified by its generators using the LLL algorithm (employed in step 5).

Unfortunately, the $w$ found using the result $\lambda$ of \texttt{LatticeReduce} need not always decrease the index of the child cones $C_j$ (this only happens when the index is very small). There are many versions of LLL, and these typically have parameters trading off quality of the basis reduction against running time. We do not have access to such parameters in \texttt{LatticeReduce} which uses Storjohann’s variant of the LLL (see \[22, 23\]) and only seeks a \textquote{semi-reduced basis}. When \texttt{LatticeReduce} fails, we may proceed as follows.

Referring to step 5, child cone $C_j$ has $\text{ind } C_j = |u_j| (\text{ind } C)$. So if $|u_j| \leq (\text{ind } C)^{-1/n}$, then $\text{ind } C_j \leq (\text{ind } C)^{(n-1)/n} < \text{ind } C$. In other words, if $\|u\|_\infty \leq (\text{ind } C)^{-1/n}$, the decomposition of cone $C$ by primitive integer vector $w = Hu$ is successful.

We can find such a $w$ directly using a method similar to the one suggested in \[1\, p. 775\]. Of course $\text{ind } C = \det H$. Let $\gamma$ be an $n$-vector with all entries equal to $\lfloor \det H \rfloor (n-1)/n$. The desired vector $w$ is any primitive solution to the following system for some $j \in \{1, ..., n\}$, the existence of which is guaranteed by Minkowski’s theorem:

$$-\gamma \leq \text{adj } H \cdot w \leq \gamma,$$

$$w_j \geq 1,$$

$$w \in \mathbb{Z}^n.$$  \hfill (\Phi_j)

The Mathematica function \texttt{Solve} finds a solution to an expression. We may use it to solve the integer-linear inequality system $\Phi_j$, and we only consider using it when \texttt{LatticeReduce} fails. As \texttt{Solve} does not always return a primitive vector, we reduce the solution vector, as needed, before continuing.

To count the number of integer points in the polytope using its rational generating function $g(P; z)$, we need to evaluate this rational generating function at $z = e$. But $z = e$ is a pole of all the fractions in our formula for $g(P; z)$. Known approaches for dealing with this are: (i) numerical perturbation and rounding (see \[24\]), and (ii) a \textquote{residue} approach (see \[25\] and \[1\]). Following our spirit of algebraic perturbation and exploiting Mathematica functionality, we use the Mathematica function \texttt{Limit} to calculate the limit of $g(P; z - \delta')$, as $\delta \to 0$, where $\delta' := (\delta, \delta^2, \ldots, \delta^n)$.
3. Example

We constructed a 4-dimensional polytope to illustrate the behavior of our algorithm. Our example, constructed to illustrate several features of the algorithm, has the following properties:

- it is not full dimensional;
- it has a non-simple vertex;
- it has many integer points;
- it has a vertex with a tangent cone of very high index.

The first two properties play to how our perturbation variant differs from the usual approach of Barvinok’s algorithm. The last two properties simply make sure that the example is not completely trivial.

Our example was constructed via its extreme points. Then we passed to an inequality representation (for input to our algorithm) using the rational-arithmetic version (\texttt{cddr+}) of the “double-description” software \texttt{cdd+} [20].

Our example is the convex hull in $\mathbb{R}^4$ of the five points:

\begin{align*}
  v_1 &= (10164, 11033, 27993, 7605)', \\
  v_2 &= (36012, 47384, 115014, 20160)', \\
  v_3 &= (11268, 23713, 52953, -963)', \\
  v_4 &= (141636/5, 230516/5, 536646/5, 43164/5)', \\
  v_5 &= (331211254/10485, 654394862/10485, 1467352282/10485, 227317156/10485)'.
\end{align*}

This polytope is only 3-dimensional — it lives in the subspace $\mathcal{L}$ that is the solution set of

$$105x_1 + 267x_2 - 142x_3 - 5x_4 = 0.$$ 

The points $v_1, v_2, v_3, v_4$ are the vertices of a (2-dimensional) quadrilateral $Q$, satisfying the further equation

$$-192x_1 - 489x_2 + 260x_3 + 9x_4 = 0.$$ 

Our polytope $P \subset \mathbb{R}^4$ is the 3-dimensional pyramid on base $Q$ and apex $v_5$. So $v_5$ is a non-simple vertex of $P$. An inequality description of $P$ is given by:
\[ A = \begin{pmatrix}
209906 & 534406 & -284349 & -8563 \\
7461256299194 & 18989267297577 & -10098183517286 & -353585936493 \\
138 & 352 & -187 & -7 \\
-697 & -1773 & 943 & 33 \\
-192 & -489 & 260 & 9 \\
105 & 267 & -142 & -5 \\
-105 & -267 & 142 & 5
\end{pmatrix} \]

\[ b = \begin{pmatrix}
4682810 \\
-22677127241406 \\
6490 \\
11286 \\
0 \\
0 \\
0
\end{pmatrix} \]

The first four rows induce the facets of \( P \) meeting at the apex \( v_5 \). The fifth row induces the facet of \( P \) that is the quadrilateral \( Q \). The sixth and seventh rows together describe the 3-dimensional subspace \( L \) containing \( P \).

With the standard approach of working in the subspace \( L \) and triangulating the tangent cone associated with \( v_5 \) (into a pair of simplicial cones), we would have 6 cones to decompose.

For our example, the polytope \( P(\tau) \) has 12 vertices — see the Appendix. Because \( P \) lives in \( L \), each simple vertex of \( P \) leads to a pair of vertices of \( P(\tau) \). That accounts for 8 vertices of \( P(\tau) \). The vertex \( v_5 \) of \( P \) leads to 4 vertices of \( P(\tau) \). It can be helpful to think of these 4 vertices as arising by first perturbing the nonsimple \( v_5 \) within the subspace \( L \) to make a pair of simple vertices, and then each of those gives rise to a pair when we perturb to break out of \( L \). Because the 12 vertices of \( P(\tau) \) come in pairs, with the only distinguishing feature between the tangent cones of each element in a pair being the presence of the sixth or seventh inequality, which have opposite normals, we really have again 6 different cones to decompose.

The different cones to decompose (labeled with their associated vertex of \( P \)) have indexes tabulated in Figure 1.

We ran our code on a Apple MacBook Air. For this example, we attempted to repeatedly decomposed any cone having index exceeding unity. The maximum number of decomposition iterations was 9. LatticeReduce
failed on 226 index-2 cones, which we then decomposed with Solve (LatticeReduce did succeed on 402 index-2 cones) The number of cones in the final decomposition was 3,380. The number of integer points is 10,826,268,359.

We carried out a few preliminary experiments, varying the maximum index (1, 2 and 100) for cones that we do not attempt to decompose. Detailed timing results, per algorithmic step, are tabulated in Figure 2.

The row labeled ‘count’ refers to calculating the number of integer points from the rational generating function. We can see that Solve is very inefficient for our purposes, and it is far better to not decompose very low-index cones (agreeing with results of others with respect to other variants of Barvinok’s algorithm). On the other hand, decomposing all cones with index greater than 2 leads to a shorter generating function than decomposing all cones with index greater than 100. So, for this example, the best of the three choices we tried is to decompose all cones with index greater than 2.
4. Experiments

We carried out computational experiments on a shiny new Apple MacBook Air, Intel Core i5 (2.7GHz), with 4GB of memory.

4.1. Test instances

We generated 10 of each of three types of test instances. Key information about each is summarized in Tables 1, 2, and 3. All of these test instances are full dimensional, and have some non-simple vertices.

For each polytope $P$, $n$ is the ambient dimension, ‘simp vert’ is the number of simple vertices, ‘deg vert’ in the number of non-simple (i.e., degenerate) vertices, ‘pert vert’ is the number of vertices of $P(\tau)$, ‘max index’ is the maximum index over all supporting cones of vertices of $P(\tau)$, and ‘integer’ is the number of integer points in $P$ (presented when we were able to calculate it; ‘Null’ if we were not able to). Note that we did not take care to insure that instances have very large numbers of integer points; rather, we use those counts as a reality check, and we mainly concentrated on producing instances having high-index cones to decompose. Furthermore, our instance do not have very high ambient dimension nor numbers of vertices. Our belief is that our results should reasonably scale with the number of vertices, and we were more interested in testing other behaviors.

Bipyramid instances have all but two points on a hyperplane, and then a pair of points on each side of the hyperplane, each on a line orthogonal to the hyperplane. Parallel Hyperplane instances are each the convex hull of points that are all on either of two parallel hyperplanes (In Table 2, $v_1$ refers to the number of points on one hyperplane and $v_2$ the other). Random instances are each the convex hull of randomly generated points.

4.2. Performance vs. decomposition level

For all 30 test instances, we ran our APB implementation, decomposing all cones with LatticeReduce until their indexes were below a threshold: $k_{\text{max}} = 1, 20$ or 100. If LatticeReduce failed to reduce the index of a cone having index above $k_{\text{max}}$, then we simply stopped decomposing such a cone (for these experiments, we did not make use of Solve; see §2). Results are depicted in Figures 3, 4, and 5. In these tables, for each instance, the ordinate is the % CPU time relative to the CPU time for $k_{\text{max}} = 100$. So for each triple of runs, the ordinate for a threshold of $k_{\text{max}} = 100$ is always 100%. What we can see is that there is usually not a significant difference between
Table 1: Bipyramid instances

<table>
<thead>
<tr>
<th>inst</th>
<th>n</th>
<th>simp</th>
<th>vert</th>
<th>deg vert</th>
<th>pert</th>
<th>max index</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>bp1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td></td>
<td>13132800</td>
<td>360</td>
</tr>
<tr>
<td>bp2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td></td>
<td>2479559149454590000</td>
<td>112038221</td>
</tr>
<tr>
<td>bp3</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>12</td>
<td></td>
<td>5241760819200</td>
<td>10836</td>
</tr>
<tr>
<td>bp4</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>12</td>
<td></td>
<td>1829798530560000</td>
<td>588509</td>
</tr>
<tr>
<td>bp5</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>16</td>
<td></td>
<td>2326879170000</td>
<td>27642</td>
</tr>
<tr>
<td>bp6</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>20</td>
<td></td>
<td>1721834827776</td>
<td>7477</td>
</tr>
<tr>
<td>bp7</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td></td>
<td>2350080</td>
<td>90</td>
</tr>
<tr>
<td>bp8</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td></td>
<td>1010693376</td>
<td>1343</td>
</tr>
<tr>
<td>bp9</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>30</td>
<td></td>
<td>1057678840373460000</td>
<td>Null</td>
</tr>
<tr>
<td>bp10</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>22</td>
<td></td>
<td>281925000</td>
<td>Null</td>
</tr>
</tbody>
</table>

Table 2: Parallel-hyperplane instances

<table>
<thead>
<tr>
<th>inst</th>
<th>n</th>
<th>v1</th>
<th>v2</th>
<th>simp</th>
<th>vert</th>
<th>deg vert</th>
<th>pert</th>
<th>max index</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ph1</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td></td>
<td>1672613935605960</td>
<td>409</td>
</tr>
<tr>
<td>ph2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td></td>
<td>212638588489166000</td>
<td>5879</td>
</tr>
<tr>
<td>ph3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td></td>
<td>7305438254383500</td>
<td>1008</td>
</tr>
<tr>
<td>ph4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>12</td>
<td></td>
<td>126699738680099000000000</td>
<td>7271</td>
</tr>
<tr>
<td>ph5</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td></td>
<td>54397463154943700000000000</td>
<td>565416</td>
</tr>
<tr>
<td>ph6</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>11</td>
<td></td>
<td>458481278400</td>
<td>43492</td>
</tr>
<tr>
<td>ph7</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td></td>
<td>21011602500</td>
<td>470</td>
</tr>
<tr>
<td>ph8</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>11</td>
<td></td>
<td>8202632483832290</td>
<td>1535</td>
</tr>
<tr>
<td>ph9</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>18</td>
<td></td>
<td>185728795397460</td>
<td>Null</td>
</tr>
<tr>
<td>ph10</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>29</td>
<td></td>
<td>1439886888272300000</td>
<td>Null</td>
</tr>
</tbody>
</table>

a $k_{\text{max}} = 20$ or 100, and both are typically much better than a $k_{\text{max}} = 1$ (for a few instances, we could not even solve with a threshold of 1, within our time and memory limits).

4.3. Decomposition depth

In Figure 6, we have depicted, on a per instance basis, the max recursion depth to which we needed to decompose a cone (using LatticeReduce). The
Table 3: Random instances

<table>
<thead>
<tr>
<th>inst</th>
<th>n</th>
<th>simp vert</th>
<th>deg vert</th>
<th>pert vert</th>
<th>max index</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>rp1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>231171</td>
<td>204</td>
</tr>
<tr>
<td>rp2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>12</td>
<td>76590650769226300</td>
<td>552642</td>
</tr>
<tr>
<td>rp3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>12</td>
<td>6859366900707900000000000</td>
<td>525087337</td>
</tr>
<tr>
<td>rp4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>16</td>
<td>67909270</td>
<td>3140</td>
</tr>
<tr>
<td>rp5</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>20</td>
<td>9149213360</td>
<td>2566</td>
</tr>
<tr>
<td>rp6</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>24</td>
<td>51868512268</td>
<td>4382</td>
</tr>
<tr>
<td>rp7</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>18</td>
<td>321801014480</td>
<td>4256</td>
</tr>
<tr>
<td>rp8</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>40</td>
<td>933042869756346000</td>
<td>212</td>
</tr>
<tr>
<td>rp9</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>38</td>
<td>92198897238</td>
<td>103</td>
</tr>
<tr>
<td>rp10</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>22</td>
<td>224882809741760</td>
<td>Null</td>
</tr>
</tbody>
</table>

Figure 3: % CPU time relative to the CPU time for $k_{\text{max}} = 100$

abscissa is the maximum index of a supporting cone of $P(\tau)$ (see Tables 1, 2, and 3), on a logarithmic scale. We see that this depth is never very high, even though our cones have very high maximum index. For each instance, we can
Figure 4: % CPU time relative to the CPU time for $k_{\text{max}} = 100$

Parallel Hyperplane

Figure 5: % CPU time relative to the CPU time for $k_{\text{max}} = 100$

Random

see the extent to which decreasing $k_{\text{max}}$ increases the recursion depth. The best $k_{\text{max}}$ should depend on whether any parallelization can be employed.
4.4. Distribution of time

Figures 7, 8 and 9 tell us where CPU time is spent in the algorithm, for each instance and choice of $k_{\text{max}}$.

Bipyramid instances: We always see a significant amount of CPU time spent in steps 6/7. Time spent in step 5 is well limited by having a modestly large value for $k_{\text{max}}$. Often the CPU time in step 3 is modest, but this is not always the case, and it is often not strongly related to the choice of $k_{\text{max}}$.

Parallel-Hyperplane instances: For $k_{\text{max}} = 20$ and 100, the CPU time is mostly spent in steps 6/7. But as the instances get harder, when $k_{\text{max}} = 1$ we have an increasing large fraction of the CPU time spent in step 5. Over all of these instances, step 3 does not take a significant portion of the CPU time.

Random instances: As for the other types of instances, CPU time spent in step 5 is well limited by having a modestly large value for $k_{\text{max}}$. Beyond that observation, the remaining CPU time can be unpredictably split between step 3 and steps 6/7.

5. Repeated cones

In APB v1.0, when the input polyhedron $P$ is not full dimensional, there are sometimes “repeated cones” — i.e., tangent cones rooted at vertices that come from the same unperturbed vertex and differ only by multiplying some generators by $-1$. Indeed, we saw this in the example of §3. Conceptually, it is an easy matter to take the economy of decomposing each such family of cones only once, but there are many details to consider to fully exploit the potential for efficiency gains. We have implemented this as APB v2.0.
If the dimension of $P$ is $n - d$, then of course there are $d$ linearly inde-
pendent equations satisfied by all points in $P$:
\[
\tilde{A}_i x = \tilde{b}_i, \text{ for } i = 1, 2, \ldots, d,
\]
we can explicitly expose the dimension deficit by including the $2d$ inequalities
\[
\begin{align*}
\tilde{A}_i x &\leq \tilde{b}_i, \text{ for } i = 1, 2, \ldots, d, \\
-\tilde{A}_i x &\leq -\tilde{b}_i, \text{ for } i = 1, 2, \ldots, d
\end{align*}
\]
into the inequality description of $P$. Repeated cones arise from these pairs of opposite sign inequalities. Alternatively, when $d > 1$, we could attempt to be more parsimonious, and instead use the $d + 1$ inequalities
\[
\begin{align*}
\tilde{A}_i x &\leq \tilde{b}_i, \text{ for } i = 1, 2, \ldots, d, \\
-\sum_{i=1}^d \tilde{A}_i x &\leq -\sum_{i=1}^d \tilde{b}_i.
\end{align*}
\]
This formulation results in fewer cones overall, but none are repeated.

We conducted experiments to compare three scenarios:

1. Not exposing the dimension deficit (i.e., using the $d + 1$ inequalities);
2. Exposing the dimension deficit (i.e., using the $2d$ inequalities), but not attempting to take advantage of repeated cones.
3. Exposing the dimension deficit (i.e., using the $2d$ inequalities), and attempting to take advantage of repeated cones.

When the dimension deficit is explicitly exposed, we can avoid certain computations for repeated cones in steps 3, 5 and 6.

In step 3, we identify indices of pairs of duplicate (opposite sign) inequalities, $E := \{(j, k) : A_j = -A_k, b_j = -b_k, 1 \leq j < k \leq m\}$. We restrict our focus to non-duplicate inequalities to find the “primary” vertices of $P(\tau)$. More precisely, primary vertices arise from $n$-element subsets of $\{1, ..., m\} \setminus K$, where $K := \{k : (j, k) \in E\}$. For primary vertex $v$ associated with $I_v \subseteq \{1, ..., m\} \setminus K$, there are up to $2^{|I_v \cap J|} - 1$ duplicates of $v$: one per nonempty subset of $I_v \cap J$, where $J := \{j : (j, k) \in E\}$.

For nonempty $D \subseteq I_v \cap J$, rather than solving a system of $n$ inequalities, we build potential duplicate vertex $v_D$ by replacing $\tau^j$ with $-\tau^k$ in $v$ for each pair $(j, k) \in E$ for which $j \in D$. We cannot skip the step of testing feasibility of $v_D$, but if we find $v_D$ is indeed a vertex of $P(\tau)$ we can easily recover the generators of the cone $C_D$ associated with $v_D$ by flipping signs.
on the appropriate generators of cone $C$ of $v$. Also, if $H$ and $H_D$ are the matrices of generators of $C$ and $C_D$, respectively, $\det H_D = (-1)^D \det H$.

In step 5, we calculate a single $w$ to decompose a family of repeated cones. We do have to check that cone(\{h_1, h_2, \ldots, h_n, w\}) is pointed for each individual cone with generators $h_1, h_2, \ldots, h_n$, but otherwise careful bookkeeping of signs of generators and determinants replaces calculations associated with cone decomposition for all but primary cones.

In step 6, we apply the formula to enumerate the integer points of the fundamental parallelepiped associated with every cone. However, the components of the formula can be recovered from that of a primary cone. The same $W$ and $S$ are appropriate for every member of a family of repeated cones. We recover $H^{-1}v$ of a repeated cone from that of a primary cone by selectively flipping row signs and replacing $\tau_j$ with $-\tau^k$ as described above.

Our non-full dimensional instances have dimension deficits $d$ ranging from 1 to 3, and ambient dimensions from $\mathbb{R}^4$ to $\mathbb{R}^6$. There are only two runs for examples 1 and 2; these are the only example with $d = 1$, so $2d = d + 1$.

The examples were constructed using three basic techniques. One method was to start with a full-dimensional polytope then increase the ambient space, adding simple equations involving the extra variables. We used unimodular matrices to translate and rotate the polytope away from new axes. This method was used for our original example described in §3, which appears here as example 1, as well as for examples 3, 4, 5, and 6. The second method, used for examples 2 and 7, was to start with the inequality description of a full-dimensional polytope, then to decrease the dimension of the polytope by choosing some of the inequalities to make equations. For example 8, we started with a cube in $\mathbb{R}^4$ bounded in each dimension by $[-100, 100]$. We included two randomly generated non-parallel equations with solutions in the cube to decrease the dimension of the polytope to 2.

Our computational results are presented in Figure 10. It is natural to expect that when $d > 1$, the $d + 1$ formulation is more efficient than the $2d$ formulation because it results in fewer vertex/cone pairs in $P(\tau)$. Indeed, in all but one of our examples, the $d + 1$ formulation resulted in a faster run time than the $2d$ formulation when both were run using APB v1.0. On average, the $d + 1$ formulation run times were 79% those of the $2d$ formulation.

When the equations are explicitly exposed, there are exactly as many primary cones in $P(\tau)$ as there are vertex/cone pairs in the unperturbed polytope $P$ — roughly 1/3 the number of cones as in the $d + 1$ formulation when $d = 2$ (a smaller fraction when $d > 2$). In all of our examples, the
efficiency gained by exploiting the repeated cones was enough to offset the overall increase in cones of the 2d formulation. The run times for APB v2.0 applied to the 2d formulation were roughly half (51% on average) that of APB v1.0 when applied to the \(d + 1\) formulation.

To see the raw improvement of APB v2.0 over v1.0 when repeated cones are involved, we compare when both are applied to the 2d formulation. In this case, APB v2.0 takes 47% of the time required by APB v1.0 on average.

6. Conclusion

Our Mathematica implementations, APB v1.0 and v2.0, are freely available to others for easy experimentation and extension. We have tested them on an Apple MacBook Air running OS X and on the $35 Raspberry Pi 2 (900MHz quad-core ARM Cortex-A7 CPU, 1GB LPDDR2 SDRAM) running ARMv6 Raspbian (a port of Debian).

We conclude by mentioning some possible extensions to the APB algorithm, and we hope that our Mathematica implementations can serve as a basis for testing these and other related ideas.
6.1. Basis reduction

It is convenient to use LLL to find a short vector because it is available in software like Mathematica. In [4, p. 133] it is suggested that the $\ell_\infty$-norm makes more sense than the $\ell_2$-norm that LLL works with. But, as pointed out in [4], when $n$ is fixed the approximation already made by LLL in only guaranteeing a short (rather than a shortest) vector is only magnified by a further constant factor in using a different norm. Still, there are alternatives to the LLL/$\ell_2$-norm that could be implemented and explored. For example, there is the so-called “generalized basis reduction algorithm” of [27].

6.2. Stopped decomposition

We decompose cones having index greater than an input parameter $k_{\text{max}}$. If LatticeReduce fails on a cone with index $k > k_{\text{max}}$, then we may decompose further using Solve. If Solve fails, or if we choose not to use it, we simply stop decomposing (step 6 still applies to simplicial cones having index greater than unity). But another possibility is to observe that a $\lambda \neq 0$ related to a decomposition has each component being an integer with absolute value no more than $\bar{k} := \lfloor k^{(n-1)/n} \rfloor \leq k - 1$. So, we could consider enumerating the $(2\bar{k} + 1)^n - 1$ possibilities for $\lambda$, and if $w := \frac{1}{\det H} H \lambda$ is an integer vector, then we have a valid decomposition. Of course this will only make sense if $(2\bar{k} + 1)^n - 1$ is reasonably small, but also note that:

- we already know $\det H$ at this point, so there is little work to do to calculate the associated $w$'s;
- we could choose to check these in an optimistic order, e.g., sorting trial $\lambda$ by 1-norm;
- we could abort any calculation of $w := \frac{1}{\det H} H \lambda$ early if we find a fractional component in a $w$;
- such an enumeration is trivial to parallelize.

6.3. Other variants

A variant of Barvinok’s algorithm opens up some facets of the cones in the signed decomposition. By doing this in a particular way, we can turn a “mod equivalence” of indicator functions (up to lower-dimensional cones) into a legitimate equation (see [4, Theorem 6.4.1]). A detailed treatment of this is in [16], and our approach can also be applied to that algorithmic variation. Many other variants exist (see [28], for example), and it could well be fruitful to integrate several other ideas.
6.4. Double description

We did not take any care to implement step 3 efficiently, and there are certainly some cases (e.g., some Random instances) where this would help. Ideally, we could make an exact-rational version of \texttt{cddr+} that could work with our algebraically-perturbed $P(\tau)$. This is not a mathematical challenge, but rather a significant software-engineering challenge. Then, of course, we would like to have this interfaced with \texttt{Mathematica} — why not dream?

Acknowledgments

The authors gratefully acknowledge Jesús De Loera for his many answers produced under relentless questioning. J. Lee was partially supported by NSF grant CMMI–1160915 and ONR grant N00014-14-1-0315.

References


Appendix

The 12 rows of this $12 \times 4$ table gives the vertices of $P(\tau) \subset \mathbb{R}^4$. 

\[
\begin{array}{cccc}
21t^6 - 1021097t^5 & + 34178412t^4 & - 459541 + 10164 \\
-21t^7 & + 1009984t^6 & - 33417862 + 3459940 & + 10164 \\
21t^6 & - 144935t^5 & + 28722t^4 & + 22988 + 36012 \\
-21t^7 & + 100051t^6 & + 28722t^5 & + 22988 + 36012 \\
21t^6 & - 10653117t^5 & + 136t^4 & + 11268 \\
-21t^7 & + 1000203t^6 & + 136t^5 & + 11268 \\
21t^6 & - t^5 & + 146t^4 & + 11266 \\
-21t^7 & - t^5 & + 146t^4 & + 11266 \\
21t^6 & + 213006674t^5 & + 27684321209t^4 & + 4690968724t^3 & + 312121254 \\
-21t^7 & + 231643741t^6 & - 262784212698t^5 & + 4690968724t^4 & + 312121254 \\
21t^6 & + 223441714t^5 & + 5118573354t^4 & + 102416302000 + 312121254 \\
-21t^7 & + 223441714t^5 & + 5118573354t^4 & + 102416302000 + 312121254 \\
51t^6 & - 1173815677t^5 & - 1150611t^4 & - 1038960 + 277993 \\
-51t^7 & + 1373851t^6 & - 1150611t^5 & - 1038960 + 277993 \\
51t^6 & - 216467t^5 & + 96999t^4 & + 19981 + 15014 \\
-51t^7 & - 216467t^5 & + 96999t^4 & + 19981 + 15014 \\
51t^6 & + 91216877t^5 & + 12103t^4 + 59692 + 52953 \\
-51t^7 & + 91216877t^5 & + 12103t^4 + 59692 + 52953 \\
51t^6 & - t^5 & + 161t^4 & + 11266 \\
-51t^7 & - t^5 & + 161t^4 & + 11266 \\
51t^6 & + 91245072t^5 & + 146732515t^4 & + 16279467512 \\
-51t^7 & + 91245072t^5 & + 146732515t^4 & + 16279467512 \\
51t^6 & + 10123523t^5 & + 114129521t^4 & + 10165 \\
-51t^7 & + 10123523t^5 & + 114129521t^4 & + 10165 \\
51t^6 & - 86953535t^5 & + 560688765t^4 & + 146732515 \\
-51t^7 & - 86953535t^5 & + 560688765t^4 & + 146732515 \\
51t^6 & - 100051t^5 & + 102416302000 + 312121254 \\
-51t^7 & - 100051t^5 & + 102416302000 + 312121254 \\
7t^6 & - 147578311t^5 & - 1660052 + 3556638 & + 7605 \\
-7t^7 & - 147578311t^6 & - 1660052 + 3556638 & + 7605 \\
7t^6 & - 479t^5 & + 19911 + 29971 + 20160 \\
-7t^7 & - 479t^6 & + 19911 + 29971 + 20160 \\
7t^6 & - 331211t^5 & - 10713 + 10712 + 963 \\
-7t^7 & - 331211t^6 & - 10713 + 10712 + 963 \\
7t^6 & - 41t^5 & + 19023113745 & + 11266 \\
-7t^7 & - 41t^6 & + 19023113745 & + 11266 \\
7t^6 & - 41t^5 & + 19023113745 & + 11266 \\
-7t^7 & - 41t^6 & + 19023113745 & + 11266 \\
51t^6 & - 223441741t^5 & + 6778361320005 + 10485 \\
-51t^7 & + 223441741t^6 & + 6778361320005 + 10485 
\end{array}
\]