`barvinok`: User Guide

Version: barvinok-0.41.8

Sven Verdoolaege

September 4, 2024

Contents

List of Figures

1 `isl` interface

1.1 Library

The `barvinok` library currently supports only a few functions that interface with the `isl` library. In time, this interface will grow and is set to replace the `PolyLib` interface. For more information on the `isl` data structures, see the `isl` user manual.

```
__isl_give isl_pw_qpolynomial *isl_basic_set_card(
        __isl_take isl_basic_set *bset);
__isl_give isl_pw_qpolynomial *isl_set_card(__isl_take isl_set *set);
__isl_give isl_union_pw_qpolynomial *isl_union_set_card(
        __isl_take isl_union_set *uset);
```

Compute the number of elements in an `isl_basic_set`, `isl_set` or `isl_union_set`. The resulting `isl_pw_qpolynomial` or `isl_union_pw_qpolynomial` has purely parametric cells.

```
__isl_give isl_pw_qpolynomial *isl_basic_map_card(
        __isl_take isl_basic_map *bmap);
__isl_give isl_pw_qpolynomial *isl_map_card(__isl_take isl_map *map);
__isl_give isl_union_pw_qpolynomial *isl_union_map_card(
        __isl_take isl_union_map *umap);
```

Compute a closed form expression for the number of image elements associated to any element in the domain of the given `isl_basic_map`, `isl_map` or `isl_union_map`. The union of the cells in the resulting `isl_pw_qpolynomial` is equal to the domain of the input `isl_map`.

```
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_sum(
        __isl_take isl_pw_qpolynomial *pwqp);
__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_sum(
        __isl_take isl_union_pw_qpolynomial *upwqp);
```

Compute the sum of the given piecewise quasipolynomial over all integer points in the domain. The result is a piecewise quasipolynomial that only involves the parameters. If, however, the domain of the piecewise quasipolynomial wraps a relation, then the sum is computed over all integer points in the range of that relation and the domain of the relation becomes the domain of the result.

```
__isl_give isl_pw_qpolynomial *isl_set_apply_pw_qpolynomial(
        __isl_take isl_set *set, __isl_take isl_pw_qpolynomial *pwqp);
__isl_give isl_union_pw_qpolynomial *isl_union_set_apply_union_pw_qpolynomial(
        __isl_take isl_union_set *uset,
        __isl_take isl_union_pw_qpolynomial *upwqp);
```

Compute the sum of the given piecewise quasipolynomial over all integer points in the intersection of the domain and the given set.

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```

Figure 1.1: A loop nest

```
__isl_give isl_pw_qpolynomial *isl_map_apply_pw_qpolynomial(
        __isl_take isl_map *map, __isl_take isl_pw_qpolynomial *pwqp);
__isl_give isl_union_pw_qpolynomial *isl_union_map_apply_union_pw_qpolynomial(
        __isl_take isl_union_map *umap,
        __isl_take isl_union_pw_qpolynomial *upwqp);
```

Compose the given map with the given piecewise quasipolynomial. That is, compute the sum over all elements in the intersection of the range of the map and the domain of the piecewise quasipolynomial as a function of an element in the domain of the map.

1.2 Calculator

The `iscc` calculator offers an interface to some of the functionality provided by the `isl` and `barvinok` libraries. The language used by `iscc` is extremely simple. The calculator supports operations on constants and dynamically typed variables and assignments (`:=`) to those variables. If the result of an expression is not used inside another expression and not assigned to a variable, then this result is printed on the screen. The operators are overloaded based on the types of the arguments, which may be sets, relations, piecewise quasipolynomials, piecewise quasipolynomial folds, lists, strings or booleans. The supported operations are shown in Table 1. Note that when an operation requires an argument of a certain type, a binary list with the first element of the required type may also be used instead. For a detailed description of some of the concepts behind `isl` and `iscc`, refer to Verdoolaege (2016).

1.2.1 Sets and Iteration Domains

Within the polyhedral model for analysis and transformation of static affine programs, the most basic kind of set is the *iteration domain*. The iteration domain represents the iterations of a statement in a loop nest. Take, for example, the loop nest in Figure 1.1 and assume first that `n` has a fixed value, say 5. The pairs of values of `i` and `j` for which statement S is executed are shown graphically in Figure 1.2. Mathematically, this set of pairs can be represented as

$$\{ (i, j) \in \mathbb{Z}^2 \mid 1 \le i \le 5 \wedge 1 \le j \le i \}$$

and the `isl` notation is very similar:

```
{ [i,j] : 1 <= i <= 5 and 1 <= j <= i }
```

In this notation, the coordinates are separated by commas and enclosed in square brackets. This description of the space in which the set lives is followed by a colon and the

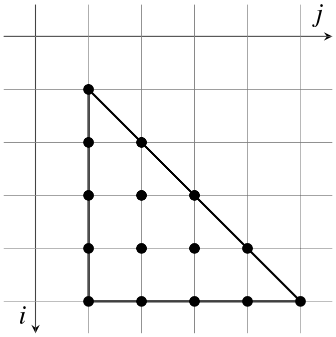


Figure 1.2: The iteration domain of the loop nest in Figure 1.1

constraints on the coordinates. Assuming the iterators are incremented by one in every iterations of the loop, a lower and upper bound on each loop iterator can be read off from the initialization and the test. Note that in an `iscc` set, the coordinates are assumed to be integer by default. For an iteration domain to be representable by such a set, the iterators therefore need to be integers.

The constraints of a set need to be affine, i.e., linear plus constant term. These affine constraint may be combined through conjunctions (`and`), disjunctions (`or`), projections (`exists`) and negations (`not`). Note that the formula is immediately converted into Disjunctive Normal Form (DNF), so it may sometimes be more efficient to construct a set from smaller sets by applying basic operations such as intersection (`*`), union (`+`) and difference (`-`). For example, the following square with its diagonal removed,

$$\{ (i, j) \mid 0 \le i, j \le 10 \wedge \neg(i = j) \}$$

can be constructed as

```
{ [i,j] : 0 <= i,j <= 10 } - { [i,i] }
```

or as

```
{ [i,j] : 0 <= i,j <= 10 and not (i = j) }
```

Note that an occurrence of a relational operator in a set description may define several constraints, one for each pair of arguments. The elements in a list of arguments are separated by a comma. If there are no constraints on the coordinates, i.e., in case of a universe set, the colon may be omitted as well. For example

```
{ [] }
```

represents the entire (unnamed) zero-dimensional space, and should not be confused with

```
{ }
```

which represents the empty set.

Returning to the iteration domain of the loop nest in Figure 1.1, we usually do not want to analyze such a program for a specific value of `n`, but instead for all possible values of `n` at once. A generic description of the iteration domain can be obtained through the introduction of a (free) parameter, as in

```
[n] -> { [i,j] : 1 <= i <= n and 1 <= j <= i }
```

The optional parameters should be declared by placing them in a comma delimited list inside `[]` (followed by an "`->`") in front of the main set description. The parameters are global and are identified by their names, so the order inside the list is arbitrary. This should be contrasted to the coordinates of a space, the names of which are only relevant within the description of the set and which are instead identified by their positions. That is,

```
[n] -> { [i,j] : 1 <= i <= n and 1 <= j <= i }
```

is equal to

```
[n] -> { [a,b] : 1 <= a <= n and 1 <= b <= a }
```

but it is not equal to

```
[n] -> { [j,i] : 1 <= i <= n and 1 <= j <= i }
```

(because the order of the coordinates has changed) or

```
[m] -> { [i,j] : 1 <= i <= m and 1 <= j <= i }
```

(because it involves a different parameter).

It is sometimes convenient to represent constraints that only involve parameters and that are not tied to any particular space. To construct such a parameter domain, the list of coordinates should simply be omitted. Note that the colon is required in this case, even if there are no constraints. In particular,

```
{ : }
```

represents the universal parameter domain, which is very different from the empty set.

To plug in a particular value for a parameter, the user should take the intersection (`*`) with a parameter domain assigns a particular value to the parameter. For example,

```
S := [n] -> { [i,j] : 1 <= i <= n and 1 <= j <= i };
S * [n] -> { : n = 5 };
```

It should be noted, though, that the result is not the same as simply replacing `n` by 5 as the result of the above sequence will still have the global parameter `n` set to 5. To avoid this assignment, the user should instead compute the gist (`%`) of the original set in the context of setting `n` to 5. That is, the result of the sequence below is `True`.

```
S1 := { [i,j] : 1 <= i <= 5 and 1 <= j <= i };
S2 := [n] -> { [i,j] : 1 <= i <= n and 1 <= j <= i };
(S2 % [n] -> { : n = 5}) = S1;
```

If a loop has a non-unit stride as in Figure 1.3 then affine constraints on the coordinates and the parameters are not sufficient to represent the iteration domain. What is needed is a way to express that the value of `i` is equal to 1 plus 3 times some integer and this is where existentially quantified variables can be used. Existentially quantified

```
for (i = 1; i <= n; i += 3)
    /* S */
```

Figure 1.3: A loop with non-unit stride

```
for (i = 1; i <= n; ++i)
    if ((i + 1) % 5 <= 2)
        /* S */
```

Figure 1.4: A loop with a modulo condition

variables are introduced by the `exists` keyword and followed by a colon. They can only be used within a single disjunct. As an example, the iteration domain of the loop in Figure 1.3 can be represented as

```
[n] -> { [i] : exists a : 1 <= i <= n and i = 1 + 3 a }
```

Existentially quantified variables are also useful to represent modulo constraints. Consider for example the loop in Figure 1.4. The iterator values `i` for which the statement `S` is executed lie between 1 and `n` and are such that the remainder of `i + 1` on division by 5 is less than or equal to 2. The constraint $(i + 1) \bmod 5 \le 2$ can be written as $(i + 1) - 5 \left\lfloor \frac{i+1}{5} \right\rfloor \le 2$, where $f = \left\lfloor \frac{i+1}{5} \right\rfloor$ is the greatest integer part of $\frac{i+1}{5}$. That is, f is the unique integer value satisfying the constraints $5f \le i + 1$ and $5f \ge (i + 1) - 4$. The iteration domain of the statement in Figure 1.4 can therefore be represented as

```
[n] -> { [i] : exists f : 1 <= i <= n and (i + 1) - 5 f <= 2 and
       (i + 1) - 4 <= 5 f <= i + 1 }
```

Since greatest integer parts occur relatively frequently, there is a special notation for them in `isl` using `[]`. The above set can therefore also be represented as

```
[n] -> { [i] : 1 <= i <= n and (i + 1) - 5 * [(i + 1)/5] <= 2 }
```

Actually, since modulos are pretty common too, `isl` also has a special notation for them and the set can therefore also be respresented as

```
[n] -> { [i] : 1 <= i <= n and (i + 1) % 5 <= 2 }
```

It should be noted that `[]` always rounds down (towards $-\infty$), while integer division in C truncates (towards 0). When translating conditions in C containing integer divisions and/or modulos to `isl` constraints, the user should therefore make sure the sign of the dividend is positive. If not, the integer division needs to be translated differently for positive and negative values.

Most programs involve more than one statement. Although it is possible to work with different sets, each representing the iteration domain of a statement, it is usually more convenient to collect all iteration domains in a single set. To be able to differentiate between iterations of different statements with the same values for the iterators, `isl` allows spaces to be named. The name is placed in front of the `[]` enclosing the

```
for (i = 0; i < n; ++i)
T:  t[i] = a[i];
for (i = 0; i < n; ++i)
    for (j = 0; j < n - i; ++j)
F:      t[j] = f(t[j], t[j+1]);
for (i = 0; i < n; ++i)
B:  b[i] = t[i];
```

Figure 1.5: A program with three loop nests

iterators. Consider for example the program in Figure 1.5. The program contains three statements which have been labeled for convenience. The iteration domain of the first statement (`T`) can be represented as

```
[n] -> { T[i] : 0 <= i < n }
```

The union of all iteration domains can be represented as

```
[n] -> {
   T[i] : 0 <= i < n;
   F[i,j] : 0 <= i < n and 0 <= j < n - i;
   B[i] : 0 <= i < n
}
```

The semicolon `;` is used to express a disjunction between spaces. This should be contrasted with the `or` keyword which expresses a disjunction between conjunctions of constraints. For example, the result of the following `iscc` statement is `True`.

```
{ [i] : i = 3 or i = 5 } = { [3]; [5] };
```

1.2.2 Maps and Access Relations

A second important concept in the polyhedral model is that of an *access relation*. An access relation connects iterations of a statement to the array elements accessed by those iterations. Such a binary relation can be represented by a map in `isl`. Maps are defined in similar way to sets, except that the single space is replaced by a pair of spaces separated by `->`. As an example, consider once more the program in Figure 1.5. In particular, consider the access `t[j+1]` in statement `F`. The index expression maps the pair of iterations `i` and `j` to `t[j+1]`, i.e., element `j+1` of a space with name `t`. Ignoring the loop bound constraints, this access relation can be represented as

```
{ F[i,j] -> t[j + 1] }
```

It is however customary to include the constraints on the iterators in the access relation, resulting in

```
[n] -> { F[i,j] -> t[j + 1] : 0 <= i < n and 0 <= j < n - i }
```

The constraints can be added by intersecting the domains of the access relations with the iteration domains. For example, the following sequence constructs the access relation for all reads in the program.

```
D := [n] -> {
    T[i] : 0 <= i < n;
    F[i,j] : 0 <= i < n and 0 <= j < n - i;
    B[i] : 0 <= i < n
};
Read := {
    T[i] -> a[i];
    F[i,j] -> t[j];
    F[i,j] -> t[j + 1];
    B[i] -> t[i]
};
Read := Read * D;
```

In this sequence, the * operator, when applied to a map and a set, intersects the domain of the map with the set.

The notation `R(S)` can be used to compute the image of the set `S` under the map `R`. For example, given the sequence above, `Read({F[0,1]})` computes the array elements read in iteration (0, 1) of statement `F` and is equal to

```
[n] -> { t[2] : n >= 2; t[1] : n >= 2 }
```

That is, elements 1 and 2 of the `t` array are read, provided `n` is at least 2.

Maps need not be single-valued. As an example, assume that `A` is a two-dimensional array of size `n` in both dimensions. Iterations `i` of a statement `S` may pass a pointer to an entire row of `A` to a function as in `f(A[i])`. Without knowing anything about `f`, we would have to assume that this function may access any element of the row. The access relation corresponding to `A[i]` is therefore

```
[n] -> { S[i] -> A[i,j] : 0 <= i,j < n }
```

This map associates `n` elements of `A` to each iteration of `S`.

1.2.3 Nested Spaces

Each space may contain a nested pair of spaces. Such nested spaces are extremely useful in more advanced applications. As an example, suppose that during equivalence checking (Verdoolaege et al. 2009) of two programs the iterations of `S1` in one program are supposed to produce the same results as the same iterations of `SA` in the other program, which may be described using the following map

```
[n] -> { S1[i] -> SA[i] : 0 <= i <= n }
```

If the iterations of `S1` depend on the same iterations of `S2`, i.e., `{S1[i]->S2[i]}`, while those of `SA` depend on the next iteration of `B`, i.e., `{SA[i]->SB[i+1]}`, then we can apply the cross product of these two dependence maps, i.e.,

```
{ [S1[i] -> SA[i']] -> [S2[i] -> SB[1+i']] }
```

to the original map to find out which iterations of `S2` should correspond to which iterations of `SB`.

1.2.4 Basic Operations

Basic operations on sets and maps include intersection (`*`), union (`+`), difference (`-`), cross product (`cross`), sampling (`sample`), affine hull (`aff`), lexicographic optimization (`lexmin` or `lexmax`), subset (`<=`) and equality (`=`) tests, code generation (`codegen`) and the cardinality (`card`). Additional operations on maps include computing domain (`dom`) and range (`ran`), differences between image and domain (`deltas`), join (`.`), inverse (`^-1`) and transitive closure (`^+`). The latter may result in an overapproximation.

The `card` operation computes the number of elements in a set or the number of elements in the image of a domain element of a map. The operation is performed exactly and completely symbolically and the result is a piecewise quasipolynomial, i.e., a subdivision of one or more spaces, with a quasipolynomial associated to each cell in the subdivision. As a trivial example, the result of

```
card { A[i] -> B[j] : 0 <= j <= i }
```

is `{ A[i] -> (1+i) : i >= 0 }`. Operations on piecewise quasipolynomials include sum (`+`) and difference (`-`) and the computation of an upper bound over the domain. If the domain contains a pair of nested spaces, then the upper bound is computed over the nested range. As another trivial example, the result of

```
ub{ [[i] -> [j]] -> j^2 : -i <= j <= i }
```

is `({ [i] -> max(i^2) : i >= 0 }, True)`. The first element in this list is the actual bound in the form of a piecewise quasipolynomial fold, i.e., a maximum of quasipolynomials defined over cells. The second indicates whether the bound is tight, i.e., whether a maximum has been computed.

1.2.5 Advanced Operations

While the basic `card` operation simply counts the number of elements in an affine set, it is also possible to assign a weight to each element of the set and to compute the sum of those weights over all the points in the set. The syntax for this weighted counting is to compute the `sum` of a piecewise quasipolynomial over its domain. As in the case of the `ub` operator, if the domain contains a pair of nested space, the sum is computed over the range. As an example, the result of

```
sum{ [[i] -> [j]] -> i*j : 0 <= j <= i }
```

is `{ [i] -> (1/2*i^2+1/2*i^3) : i >= 0 }`.

After the computation of some sum or bound, the result may have to be reformulated in terms of other variables. For example, during inter procedural analysis, a result computed in terms of the formal parameters may have to be reformulated in terms of the actual parameters. `iscc` therefore allows maps and piecewise quasipolynomials or folds to be composed. If the map is multi-valued, then the composition maps each domain element to the sum or upper bound of the values at its image elements.

Finally, because of its high-level representation, `iscc` can provide a dependence analysis operation taking only three maps as input, the sink accesses, the potential source accesses and a schedule. The result is a single dependence map.

1.2.6 More Examples

```
P := [n, m] -> { [i,j] : 0 <= i <= n and i <= j <= m };
card P;

f := [n,m] -> { [i,j] -> i*j + n*i*i*j : i,j >= 0 and 5i + 27j <= n+m };
sum f;
s := sum f;
s @ [n,m] -> { : 0 <= n,m <= 20 };

f := [n] -> { [i] -> 2*n*i - n*n + 3*n - 1/2*i*i - 3/2*i-1 :
                (exists j : 0 <= i < 4*n-1 and 0 <= j < n and
                            2*n-1 <= i+j <= 4*n-2 and i <= 2*n-1 ) };
ub f;
u := (ub f)[0];
u @ [n] -> { : 0 <= n <= 10 };

m := [n] -> { [i,j] -> [i+1,j+1] : 1 <= i,j < n;
              [i,j] -> [i+1,j-1] : 1 <= i < n and 2 <= j <= n };
m^+;
(m^+)[0];

codegen [N] -> { A[i] -> [i,0] : 0 <= i <= N; B[i] -> [i,1] : 1 <= i <= N };

{ [k] -> [i] : 1 <= i <= k } . { [n] -> 2 * n : n >= 1 };

{ [m] -> [c] : 1 <= c <= m } . { [k] -> max((3 * k + k^2)) : k >= 1 };
```

1.2.7 Comparison to other Interactive Polyhedral Tools

Two related interactive polyhedral tools are the Omega calculator (Kelly et al. 1996) and `SPPoC` (Boulet and Redon 2001). The syntax of `iscc` was very much inspired by that of the Omega calculator. However, the Omega calculator only knows sets and maps. In particular, it does not perform any form of counting. An earlier version of `barvinok` came with a modified version of the Omega calculator that introduced an operation for counting the number of elements in a set, but it would simply print the result and not allow any further manipulations. `SPPoC` does support counting, but only the basic operation of counting the elements in a set. In particular, it does not support weighted counting, nor the computation of upper bounds. It also only supports (single-valued) functions and not generic relations like the Omega calculator and `iscc`. Internally, `SPPoC` uses `PolyLib`, `PipLib` and `omega` to perform its operations. Although the first two of these libraries may have been state-of-the-art at the time `SPPoC` was created, they are no longer actively maintained and have been largely superseded by more recent libraries. In particular, `PipLib` effectively only supports a single operation, which is now also available in both `isl` and `PPL`. The operations on rational polyhedra in `PolyLib` are also available in `PPL`, usually through a cleaner interface and with a more efficient implementation. As to counting the elements in a parametric polytope, Barvinok's algorithm, implemented in the `barvinok` library, is usually much faster than the algorithm implemented in `PolyLib` (Verdoolaege et al. 2007b). Furthermore, the ability to work with named and nested spaces and the ability of sets and maps to contain (pairs of) elements from different spaces are not available in the Omega calculator and `SPPoC`.

Table 1: `iscc` operations. The variables have the following types, s: set, m: map, q: piecewise quasipolynomial, f: piecewise quasipolynomial fold, t: schedule (tree), l: list, i: integer, b: boolean, S: string, o: object of any type

Syntax	Meaning
s_2 := `aff` s_1	affine hull of s_1
m_2 := `aff` m_1	affine hull of m_1
q := `card` s	number of elements in the set s
q := `card` m	number of elements in the image of a domain element
s_2 := `coalesce` s_1	simplify the representation of set s_1 by trying to combine pairs of basic sets into a single basic set
m_2 := `coalesce` m_1	simplify the representation of map m_1 by trying to combine pairs of basic maps into a single basic map
q_2 := `coalesce` q_1	simplify the representation of q_1 by trying to combine pairs of basic sets in the domain of q_1 into a single basic set
f_2 := `coalesce` f_1	simplify the representation of f_1 by trying to combine pairs of basic sets in the domain of f_1 into a single basic set
`codegen` t	generate code for the given schedule.
`codegen` m	generate code for the given schedule.
`codegen` m_1 `using` m_2	generate code for the schedule m_1 using the options m_2.
s_2 := `coefficients` s_1	The set of coefficients of valid constraints for s_1
s_2 := `solutions` s_1	The set of elements satisfying the constraints with coefficients in s_1
s_3 := s_1 `cross` s_2	Cartesian product of s_1 and s_2
m_3 := m_1 `cross` m_2	Cartesian product of m_1 and m_2
m_3 := m_1 `cross_range` m_2	Cartesian product of the ranges of m_1 and m_2 for their shared domain
s := `deltas` m	the set $\{\, y - x \mid x \to y \in m \,\}$
m_2 := `deltas_map` m_1	the map $\{\, (x \to y) \to y - x \mid x \to y \in m_1 \,\}$
s := `dom` m	domain of map m
s := `dom` q	domain of piecewise quasipolynomial q
s := `dom` f	domain of piecewise quasipolynomial fold f
s := `dom` t	domain of schedule t
s := `domain` m	domain of map m
s := `domain` q	domain of piecewise quasipolynomial q
s := `domain` f	domain of piecewise quasipolynomial fold f
s := `domain` t	domain of schedule t
m_2 := `domain_map` m_1	a map from a wrapped copy of m_1 to the domain of m_1

continued on next page

Syntax	Meaning
s := `ran` m	range of map m
s := `range` m	range of map m
m_2 := `range_map` m_1	a map from a wrapped copy of m_1 to the range of m_1
m := `identity` s	identity relation on s
q := `lattice_width` s	lattice width of the set s
l := `lb` q	compute a lower bound on the piecewise quasipolynomial q over all integer points in the domain of q and return a list containing the lower bound and a boolean that is true if the lower bound is known to be tight. If the domain of q wraps a map, then the lower bound is computed over all integer points in the range of the wrapped map instead.
s_2 := `lexmin` s_1	lexicographically minimal element of s_1
m_2 := `lexmin` m_1	lexicographically minimal image element
s_2 := `lexmax` s_1	lexicographically maximal element of s_1
m_2 := `lexmax` m_1	lexicographically maximal image element
s_2 := `lift` s_1	lift s_1 to a space with extra dimensions corresponding to the existentially quantified variables in s_1 such that `domain(unwrap(lift S))` is equal to `S`
m := `map` t	convert schedule t to a map representation
s_2 := `params` s_1	parameter domain of set s_1
s := `params` m	parameter domain of map m
l := `parse_file` S	parse the file names S and return a list consisting of the iteration domain, the must write access relation, the may write access relation, the read access relation and the original schedule. This operation is only available if `pet` support was compiled in.
s_2 := `poly` s_1	polyhedral hull of s_1
m_2 := `poly` m_1	polyhedral hull of m_1
q_2 := `poly` q_1	polynomial approximation of q_1
q_2 := `lpoly` q_1	polynomial underapproximation of q_1
q_2 := `upoly` q_1	polynomial overapproximation of q_1
l := `pow` m	compute an overapproximation of the power of m and return a list containing the overapproximation and a boolean that is true if the overapproximation is known to be exact
`print` o	print object
o := `read` "*filename*"	read object from file
s_2 := `sample` s_1	a sample element of the set s_1
m_2 := `sample` m_1	a sample element of the map m_1
s_2 := `scan` s_1	the set s_1 split into individual elements, provided s_1 contains a finite number of elements

continued on next page

Syntax	Meaning
m_2 := `scan` m_1	the map m_1 split into individual elements, provided m_1 contains a finite number of elements
`source` "*filename*"	read commands from file
q_2 := `sum` q_1	sum q_1 over all integer points in the domain of q_1, or, if the domain of q_1 wraps a map, over all integer points in the range of the wrapped map.
S := `typeof` o	a string representation of the type of o
l := `ub` q	compute an upper bound on the piecewise quasipolynomial q over all integer points in the domain of q and return a list containing the upper bound and a boolean that is true if the upper bound is known to be tight. If the domain of q wraps a map, then the upper bound is computed over all integer points in the range of the wrapped map instead.
l := `vertices` s	a list of vertices of the rational polytope defined by the same constraints as s
s := `wrap` m	wrap the map in a set
m := `unwrap` s	unwrap the map from the set
`write` "*filename*" o	write object to file
m_2 := `zip` m_1	the cross product of domain and range of m_1, i.e., $\{ (\mathbf{w} \to \mathbf{y}) \to (\mathbf{x} \to \mathbf{z}) \mid (\mathbf{w} \to \mathbf{x}) \to (\mathbf{y} \to \mathbf{z}) \in m_1 \}$
m_3 := `any` m_1 `before` m_2 `under` t	compute a map from any element x in the domain of m_1 to any element y in the domain of m_2 such that their images $m_1(x)$ and $m_2(y)$ overlap and such that x is scheduled before y by t.
m_4 := `any` m_1 `before` m_2 `under` m_3	same as the previous operation, with the schedule represented by a map.
l := `last` m_1 `before` m_2 `under` t	compute a map that contains for any element y in the domain of m_2 a mapping from the last element x in the domain of m_1 (according to the schedule t) to y such that $m_1(x)$ and $m_2(y)$ overlap and such that x is scheduled before y by t. Return a list containing this map and the subset of m_2 for which there is no corresponding element in the domain of m_1.
l := `last` m_1 `before` m_2 `under` m_3	same as the previous operation, with the schedule represented by a map.
m_4 := `any` m_1 `last` m_2 `before` m_3 `under` t	compute a map that contains for any element y in the domain of m_3 a mapping from the last element x in the domain of m_2 (according to the schedule t) to y such that $m_2(x)$ and $m_3(y)$ overlap and such that x is scheduled before y by t as well as from any element z in the domain of m_1 such that $m_1(z)$ and $m_3(y)$ overlap, z is scheduled before y by t and such that there is no x in the domain of m_2 with $m_2(x) \cap m_3(y) \neq \emptyset$ and x scheduled between z and y by t.

continued on next page

Syntax	Meaning
$m_5 := \mathtt{any}\ m_1\ \mathtt{last}\ m_2\ \mathtt{before}\ m_3\ \mathtt{under}\ m_4$	same as the previous operation, with the schedule represented by a map.
$t := \mathtt{schedule}\ s\ \mathtt{respecting}\ m_1\ \mathtt{minimizing}\ m_2$	compute a schedule for the domains in s that respects all dependences in m_1 and tries to minimize the dependences in m_2.
$b_3 := b_1 + b_2$	or
$i_3 := i_1 + i_2$	sum
$s_3 := s_1 + s_2$	union
$m_3 := m_1 + m_2$	union
$q_3 := q_1 + q_2$	sum
$f_2 := f_1 + q$	sum
$f_2 := q + f_1$	sum
$S_3 := S_1 + S_2$	concatenation
$S_2 := o + S_1$	concatenation of stringification of o and S_1
$S_2 := S_1 + o$	concatenation of S_1 and stringification of o
$i_3 := i_1 - i_2$	difference
$s_3 := s_1 - s_2$	set difference
$m_3 := m_1 - m_2$	set difference
$m_2 := m_1 - s$	subtract s from the domain of m_1
$m_2 := m_1\ \mathtt{->-}\ s$	subtract s from the range of m_1
$q_3 := q_1 - q_2$	difference
$b_3 := b_1 * b_2$	and
$i_3 := i_1 * i_2$	product
$s_3 := s_1 * s_2$	intersection
$m_3 := m_1 * m_2$	intersection
$q_2 := i * q_1$	product
$q_2 := q_1 * i$	product
$q_3 := q_1 * q_2$	product
$f_2 := i * f_1$	product
$f_2 := f_1 * i$	product
$m_2 := m_1 * s$	intersect domain of m_1 with s
$q_2 := q_1 * s$	intersect domain of q_1 with s
$f_2 := f_1 * s$	intersect domain of f_1 with s
$m_2 := m_1\ \mathtt{->}{*}\ s$	intersect range of m_1 with s
$s_2 := m(s_1)$	apply map m to set s_1
$q_2 := q_1(s)$	apply q_1 to each element in s and compute the sum of the results
$l := f(s)$	apply f to each element in s, compute a bound of the results and return a list containing the bound and a boolean that is true if the bound is known to be tight.

continued on next page

Syntax	Meaning
$m_3 := m_1$ `.` m_2	join of m_1 and m_2
$m_3 := m_1$ `before` m_2	join of m_1 and m_2
$m_3 := m_2(m_1)$	join of m_1 and m_2
$m_3 := m_2$ `after` m_1	join of m_1 and m_2
$f_3 := f_1$ `.` f_2	join of f_1 and f_2, combining the lists of quasipolynomials over shared domains
$q_2 := m$ `.` q_1	join of m and q_1, taking the sum over all elements in the intersection of the range of m and the domain of q_1
$q_2 := m$ `before` q_1	$q_2 := m$ `.` q_1
$q_2 := q_1(m)$	$q_2 := m$ `.` q_1
$q_2 := q_1$ `after` m	$q_2 := m$ `.` q_1
$l := m$ `.` f	join of m and f, computing a bound over all elements in the intersection of the range of m and the domain of f and returning a list containing the bound and a boolean that is true if the bound is known to be tight.
$l := m$ `before` f	$l := m$ `.` f
$l := f(m)$	$l := m$ `.` f
$l := f$ `after` m	$l := m$ `.` f
$m := s_1$ `->` s_2	universal map with domain s_1 and range s_2
$q_2 := q_1$ `@` s	evaluate the piecewise quasipolynomial q_1 in each element of the set s and return a piecewise quasipolynomial mapping each of the individual elements to the resulting constant
$q := f$ `@` s	evaluate the piecewise quasipolynomial fold f in each element of the set s and return a piecewise quasipolynomial mapping each of the individual elements to the resulting constant
$s_3 := s_1$ `%` s_2	simplify s_1 in the context of s_2, i.e., compute the gist of s_1 given s_2
$m_3 := m_1$ `%` m_2	simplify m_1 in the context of m_2, i.e., compute the gist of m_1 given m_2
$m_2 := m_1$ `%` s	simplify m_1 in the context of the domain s, i.e., compute the gist of m_1 given domain s
$q_2 := q_1$ `%` s	simplify q_1 in the context of the domain s, i.e., compute the gist of q_1 given s
$f_2 := f_1$ `%` s	simplify f_1 in the context of the domain s, i.e., compute the gist of f_1 given s
$m_2 := m_1$`^i`	the ith power of m_1; if i is negative then the result is the $(-i)$th power of the inverse of m_1

continued on next page

Syntax	Meaning
$l := m\hat{}+$	compute an overapproximation of the transitive closure of m and return a list containing the overapproximation and a boolean that is true if the overapproximation is known to be exact
$o := l[i]$	the element at position i in the list l
$b := q_1 == q_2$	is q_1 obviously equal to q_2?
$b := f_1 == f_2$	is f_1 obviously equal to f_2?
$b := s_1 = s_2$	is s_1 equal to s_2?
$b := m_1 = m_2$	is m_1 equal to m_2?
$b := S_1 = S_2$	is S_1 equal to S_2?
$b := s_1 <= s_2$	is s_1 a subset of s_2?
$b := m_1 <= m_2$	is m_1 a subset of m_2?
$b := s_1 < s_2$	is s_1 a proper subset of s_2?
$b := m_1 < m_2$	is m_1 a proper subset of m_2?
$b := s_1 >= s_2$	is s_1 a superset of s_2?
$b := m_1 >= m_2$	is m_1 a superset of m_2?
$b := s_1 > s_2$	is s_1 a proper superset of s_2?
$b := m_1 > m_2$	is m_1 a proper superset of m_2?
$m := s_1 << s_2$	a map from s_1 to s_2 of those elements that live in the same space and such that the elements of s_1 are lexicographically strictly smaller than those of s_2.
$m_3 := m_1 << m_2$	a map from the domain of m_1 to the domain of m_2 of those elements such that their images live in the same space and such that the images of the elements of m_1 are lexicographically strictly smaller than those of m_2.
$m := s_1 <<= s_2$	a map from s_1 to s_2 of those elements that live in the same space and such that the elements of s_1 are lexicographically smaller than those of s_2.
$m_3 := m_1 <<= m_2$	a map from the domain of m_1 to the domain of m_2 of those elements such that their images live in the same space and such that the images of the elements of m_1 are lexicographically smaller than those of m_2.
$m := s_1 >> s_2$	a map from s_1 to s_2 of those elements that live in the same space and such that the elements of s_1 are lexicographically strictly greater than those of s_2.
$m_3 := m_1 >> m_2$	a map from the domain of m_1 to the domain of m_2 of those elements such that their images live in the same space and such that the images of the elements of m_1 are lexicographically strictly greater than those of m_2.

continued on next page

Syntax	Meaning
$m := s_1 >>= s_2$	a map from s_1 to s_2 of those elements that live in the same space and such that the elements of s_1 are lexicographically greater than those of s_2.
$m_3 := m_1 >>= m_2$	a map from the domain of m_1 to the domain of m_2 of those elements such that their images live in the same space and such that the images of the elements of m_1 are lexicographically greater than those of m_2.

2 `PolyLib` interface of the `barvinok` library (obsolescent)

Although `barvinok` currently still uses `PolyLib` internally, this is likely to change in the not too distant future. Consider using `isl` based alternatives for the functions in this section as the latter are likely to be removed in future releases.

Our `barvinok` library is built on top of `PolyLib` (Wilde 1993; Loechner 1999). In particular, it reuses the implementations of the algorithm of Loechner and Wilde (1997) for computing parametric vertices and the algorithm of Clauss and Loechner (1998) for computing chamber decompositions. Initially, our library was meant to be a replacement for the algorithm of Clauss and Loechner (1998), also implemented in `PolyLib`, for computing quasi-polynomials. To ease the transition of application programs we tried to reuse the existing data structures as much as possible.

2.1 Existing Data Structures

Inside `PolyLib` integer values are represented by the `Value` data type. Depending on a configure option, the data type may either by a 32-bit integer, a 64-bit integer or an arbitrary precision integer using `GMP`. The `barvinok` library requires that `PolyLib` is compiled with support for arbitrary precision integers.

The basic structure for representing (unions of) polyhedra is a `Polyhedron`.

```
typedef struct polyhedron {
  unsigned Dimension, NbConstraints, NbRays, NbEq, NbBid;
  Value **Constraint;
  Value **Ray;
  Value *p_Init;
  int p_Init_size;
  struct polyhedron *next;
} Polyhedron;
```

The attribute `Dimension` is the dimension of the ambient space, i.e., the number of variables. The attributes `Constraint` and `Ray` point to two-dimensional arrays of constraints and generators, respectively. The number of rows is stored in `NbConstraints` and `NbRays`, respectively. The number of columns in both arrays is equal to `1+Dimension+1`. The first column of `Constraint` is either 0 or 1 depending on whether the constraint is an equality (0) or an inequality (1). The number of equalities is stored in `NbEq`. If the constraint is $\langle \mathbf{a}, \mathbf{x} \rangle + c \geq 0$, then the next columns contain the coefficients a_i and the final column contains the constant c. The first column of `Ray` is either 0 or 1 depending on whether the generator is a line (0) or a vertex or ray (1). The number of lines is stored in `NbBid`. Let d be the least common multiple (lcm) of the denominators of the coordinates of a vertex $\mathbf{v}$, then the next columns contain dv_i and the final column contains d. For a ray, the final column contains 0. The field `next` points to the next polyhedron in the union of polyhedra. It is `0` if this is the last (or only) polyhedron in the union. For more information on this structure, we refer to Wilde (1993).

Quasi-polynomials are represented using the `evalue` and `enode` structures.

```
typedef enum { polynomial, periodic, evector } enode_type;

typedef struct _evalue {
  Value d;              /* denominator */
  union {
    Value n;            /* numerator (if denominator != 0) */
    struct _enode *p;   /* pointer   (if denominator == 0) */
  } x;
} evalue;

typedef struct _enode {
  enode_type type;      /* polynomial or periodic or evector */
  int size;             /* number of attached pointers */
  int pos;              /* parameter position */
  evalue arr[1];        /* array of rational/pointer */
} enode;
```

If the field `d` of an `evalue` is zero, then the `evalue` is a placeholder for a pointer to an `enode`, stored in `x.p`. Otherwise, the `evalue` is a rational number with numerator `x.n` and denominator `d`. An `enode` is either a `polynomial` or a `periodic`, depending on the value of `type`. The length of the array `arr` is stored in `size`. For a `polynomial`, `arr` contains the coefficients. For a `periodic`, it contains the values for the different residue classes modulo the parameter indicated by `pos`. For a polynomial, `pos` refers to the variable of the polynomial. The value of `pos` is 1 for the first parameter. That is, if the value of `pos` is 1 and the first parameter is p, and if the length of the array is l, then in case it is a polynomial, the `enode` represents

$$\texttt{arr[0]} + \texttt{arr[1]}p + \texttt{arr[2]}p^2 + \cdots + \texttt{arr[l-1]}p^{l-1}.$$

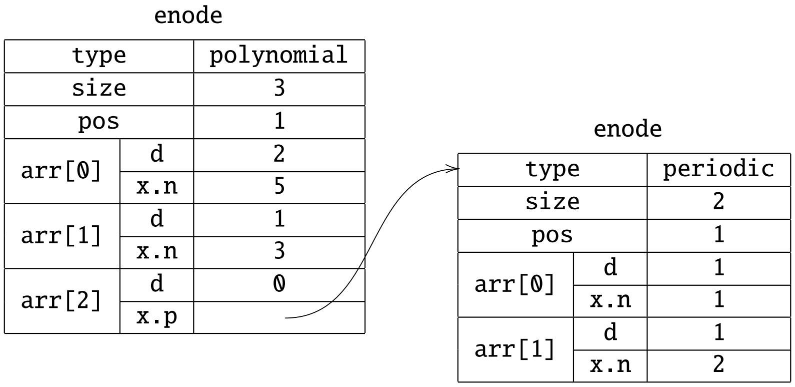
If it is a periodic, then it represents

$$[\texttt{arr[0]}, \texttt{arr[1]}, \texttt{arr[2]}, \ldots, \texttt{arr[l-1]}]_p .$$

Note that the elements of a `periodic` may themselves be other `periodic`s or even `polynomial`s. In our library, we only allow the elements of a `periodic` to be other `periodic`s or rational numbers. The chambers and their corresponding quasi-polynomial are stored in `Enumeration` structures.

```
typedef struct _enumeration {
  Polyhedron *ValidityDomain; /* constraints on the parameters */
  evalue EP;                  /* dimension = combined space    */
  struct _enumeration *next;  /* Ehrhart Polynomial,
                                 corresponding to parameter
                                 values inside the domain
                                 ValidityDomain above          */
} Enumeration;
```

For more information on these structures, we refer to Loechner (1999).



enode

type		polynomial
size		3
pos		1
arr[0]	d	2
	x.n	5
arr[1]	d	1
	x.n	3
arr[2]	d	0
	x.p	

enode

type		periodic
size		2
pos		1
arr[0]	d	1
	x.n	1
arr[1]	d	1
	x.n	2

Figure 2.2: The quasi-polynomial $[1, 2]_p p^2 + 3p + \frac{5}{2}$.

Example 2.1 *Figure 2.2 is a skillful reconstruction of Figure 2 from Loechner (1999). It shows the contents of the* `enode` *structures representing the quasi-polynomial* $[1, 2]_p p^2 + 3p + \frac{5}{2}$.

2.2 Options

The `barvinok_options` structure contains various options that influence the behavior of the library.

```
struct barvinok_options {
    struct barvinok_stats   *stats;

    /* PolyLib options */
    unsigned    MaxRays;

    /* NTL options */
                /* LLL reduction parameter delta=LLL_a/LLL_b */
    long        LLL_a;
    long        LLL_b;

    /* barvinok options */
    #define BV_SPECIALIZATION_BF 2
    #define BV_SPECIALIZATION_DF 1
    #define BV_SPECIALIZATION_RANDOM 0
    #define BV_SPECIALIZATION_TODD 3
    int         incremental_specialization;

    unsigned long           max_index;
    int                     primal;
    int                     lookup_table;
    int                     count_sample_infinite;

    int                     try_Delaunay_triangulation;

    #define     BV_APPROX_SIGN_NONE     0
    #define     BV_APPROX_SIGN_APPROX   1
    #define     BV_APPROX_SIGN_LOWER    2
    #define     BV_APPROX_SIGN_UPPER    3
    int                     polynomial_approximation;
    #define     BV_APPROX_NONE          0
    #define     BV_APPROX_DROP          1
    #define     BV_APPROX_SCALE         2
    #define     BV_APPROX_VOLUME        3
    #define BV_APPROX_BERNOULLI 4
    int                     approximation_method;
    #define     BV_APPROX_SCALE_FAST    (1 << 0)
    #define     BV_APPROX_SCALE_NARROW  (1 << 1)
    #define     BV_APPROX_SCALE_NARROW2 (1 << 2)
    #define     BV_APPROX_SCALE_CHAMBER (1 << 3)
    int                     scale_flags;
    #define     BV_VOL_LIFT             0
    #define     BV_VOL_VERTEX           1
    #define     BV_VOL_BARYCENTER       2
    int                     volume_triangulate;

    /* basis reduction options */
    #define     BV_GBR_GLPK     1
    #define     BV_GBR_CDD      2
    int         gbr_lp_solver;

    #define     BV_LP_POLYLIB           0
    #define     BV_LP_GLPK              1
    #define     BV_LP_CDD               2
    #define     BV_LP_CDDF              3
    int         lp_solver;

    #define     BV_HULL_GBR             0
    #define     BV_HULL_HILBERT         1
    int         integer_hull;
};
```

```
struct barvinok_options *barvinok_options_new_with_defaults();
```

The function `barvinok_options_new_with_defaults` can be used to create a `barvinok_options` structure with default values.

- `PolyLib` options

  - `MaxRays`

    The value of `MaxRays` is passed to various `PolyLib` functions and defines the maximum size of a table used in the double description computation in the `PolyLib` function `Chernikova`. In earlier versions of `PolyLib`, this parameter had to be conservatively set to a high number to ensure successful operation, resulting in significant memory overhead. Our change to allow this table to grow dynamically is available in recent versions of `PolyLib`. In these versions, the value no longer indicates the maximal table size, but rather the size of the initial allocation. This value may be set to `0` or left as set by `barvinok_options_new_with_defaults`.

- `NTL` options
  - `LLL_a`
  - `LLL_b`

    The values used for the reduction parameter in the call to `NTL`'s implementation of Lenstra, Lenstra and Lovasz' basis reduction algorithm (LLL).

- `barvinok` specific options
  - `incremental_specialization`

    Selects the specialization algorithm to be used. If set to `0` then a direct specialization is performed using a random vector. Value `1` selects a depth first incremental specialization, while value `2` selects a breadth first incremental specialization. For more information we refer to Verdoolaege (2005, Section 4.4.3).

2.3 Data Structures for Quasi-polynomials

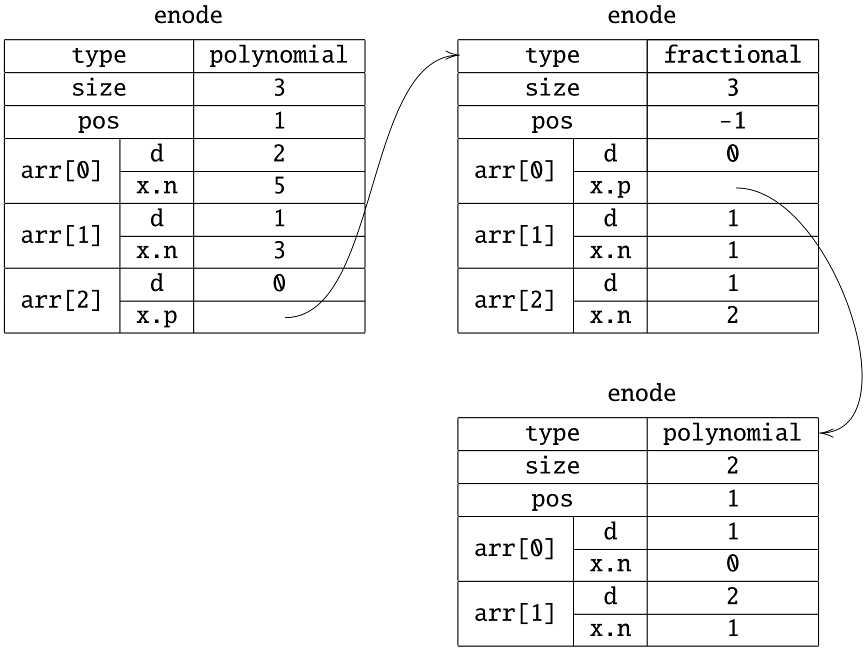
Internally, we do not represent our quasi-polynomials as step-polynomials, but instead as polynomials of fractional parts of degree-1 polynomials. However, we also allow our quasi-polynomials to be represented as polynomials with periodic numbers for coefficients, similarly to Loechner (1999). By default, the current version of `barvinok` uses `fractional`s, but this can be changed through the `--disable-fractional` configure option. When this option is specified, the periodic numbers are represented as an explicit enumeration using the `periodic` type. A quasi-polynomial based on fractional parts can also be converted to an actual step-polynomial using `evalue_frac2floor`, but this is not fully supported yet.

For reasons of compatibility,[1] we shoehorned our representations for piecewise quasi-polynomials into the existing data structures. To this effect, we introduced four new types, `fractional`, `relation`, `partition` and `flooring`.

```
typedef enum { polynomial, periodic, evector, fractional,
               relation, partition, flooring } enode_type;
```

---

[1] Also known as laziness.



enode

type		polynomial
size		3
pos		1
arr[0]	d	2
	x.n	5
arr[1]	d	1
	x.n	3
arr[2]	d	0
	x.p	

enode

type		fractional
size		3
pos		-1
arr[0]	d	0
	x.p	
arr[1]	d	1
	x.n	1
arr[2]	d	1
	x.n	2

enode

type		polynomial
size		2
pos		1
arr[0]	d	1
	x.n	0
arr[1]	d	2
	x.n	1

Figure 2.4: The quasi-polynomial $\left(1 + 2\left\{\frac{p}{2}\right\}\right)p^2 + 3p + \frac{5}{2}$.

The field `pos` is not used in most of these additional types and is therefore set to `-1`.

The types `fractional` and `flooring` represent polynomial expressions in a fractional part or a floor respectively. The generator is stored in `arr[0]`, while the coefficients are stored in the remaining array elements. That is, an `enode` of type `fractional` represents

$$\texttt{arr[1]} + \texttt{arr[2]}\{\texttt{arr[0]}\} + \texttt{arr[3]}\{\texttt{arr[0]}\}^2 + \cdots + \texttt{arr[l-1]}\{\texttt{arr[0]}\}^{l-2}.$$

An `enode` of type `flooring` represents

$$\texttt{arr[1]} + \texttt{arr[2]}\lfloor\texttt{arr[0]}\rfloor + \texttt{arr[3]}\lfloor\texttt{arr[0]}\rfloor^2 + \cdots + \texttt{arr[l-1]}\lfloor\texttt{arr[0]}\rfloor^{l-2}.$$

Example 2.3 *The internal representation of the quasi-polynomial*

$$\left(1 + 2\left\{\frac{p}{2}\right\}\right)p^2 + 3p + \frac{5}{2}$$

is shown in Figure 2.4.

The `relation` type is used to represent strides. In particular, if the value of `size` is 2, then the value of a `relation` is (in pseudo-code):

```
(value(arr[0]) == 0) ? value(arr[1]) : 0
```

If the size is 3, then the value is:

```
(value(arr[0]) == 0) ? value(arr[1]) : value(arr[2])
```

The type of `arr[0]` is typically `fractional`.

Finally, the `partition` type is used to represent piecewise quasi-polynomials. We prefer to encode this information inside `evalue`s themselves rather than using `Enumeration`s since we want to perform the same kinds of operations on both quasi-polynomials and piecewise quasi-polynomials. An `enode` of type `partition` may not be nested inside another `enode`. The size of the array is twice the number of "chambers". Pointers to chambers are stored in the even slots, whereas pointer to the associated quasi-polynomials are stored in the odd slots. To be able to store pointers to chambers, the definition of `evalue` was changed as follows.

```
typedef struct _evalue {
  Value d;              /* denominator */
  union {
    Value n;            /* numerator (if denominator > 0) */
    struct _enode *p;   /* pointer   (if denominator == 0) */
    Polyhedron *D;      /* domain    (if denominator == -1) */
  } x;
} evalue;
```

Note that we allow a "chamber" to be a union of polyhedra as discussed in Verdoolaege (2005, Section 4.5.1). Chambers with extra variables, i.e., those of Verdoolaege (2005, Section 4.6.5), are only partially supported. The field `pos` is set to the actual dimension, i.e., the number of parameters.

2.4 Operations on Quasi-polynomials

In this section we discuss some of the more important operations on `evalue`s provided by the `barvinok` library. Some of these operations are extensions of the functions from `PolyLib` with the same name.

Most of these operation are also provided by `isl` on `isl_pw_qpolynomial`s, which are set to replace `evalue`s. Use `isl_pw_qpolynomial_from_evalue` to convert from `evalue`s to `isl_pw_qpolynomial`s.

```
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_from_evalue(
        __isl_take isl_space *dim, const evalue *e);
```

```
void eadd(const evalue *e1,evalue *res);
void emul(const evalue *e1, evalue *res);
```

The functions `eadd` and `emul` takes two (pointers to) `evalue`s `e1` and `res` and computes their sum and product respectively. The result is stored in `res`, overwriting (and deallocating) the original value of `res`. It is an error if exactly one of the arguments of `eadd` is of type `partition` (unless the other argument is `0`). The addition and multiplication operations are described in Verdoolaege (2005, Section 4.5.1) and Verdoolaege (2005, Section 4.5.2) respectively.

The function `eadd` is an extension of the function `new_eadd` from Seghir (2002). Apart from supporting the additional types from Section 2.3, the new version also additionally imposes an order on the nesting of different `enode`s. Without such an ordering, `evalue`s could be constructed representing for example

$$(0y^0 + (0x^0 + 1x^1)y^1)x^0 + (0y^0 - 1y^1)x^1,$$

which is just a funny way of saying 0.

```
void eor(evalue *e1, evalue *res);
```

The function `eor` implements the union operation from Verdoolaege (2005, Section 4.5.3). Both arguments are assumed to correspond to indicator functions.

```
evalue *esum(evalue *E, int nvar);
evalue *evalue_sum(evalue *E, int nvar, unsigned MaxRays);
```

The function `esum` has been superseded by `evalue_sum`. The function `evalue_sum` performs the summation operation from Verdoolaege (2005, Section 4.5.4). The piecewise step-polynomial represented by `E` is summated over its first `nvar` variables. Note that `E` must be zero or of type `partition`. The function returns the result in a newly allocated `evalue`. Note also that `E` needs to have been converted from `fractional`s to `flooring`s using the function `evalue_frac2floor`.

```
void evalue_frac2floor(evalue *e);
```

This function also ensures that the arguments of the `flooring`s are positive in the relevant chambers. It currently assumes that the argument of each `fractional` in the original `evalue` has a minimum in the corresponding chamber.

```
double compute_evalue(const evalue *e, Value *list_args);
Value *compute_poly(Enumeration *en,Value *list_args);
evalue *evalue_eval(const evalue *e, Value *values);
```

The functions `compute_evalue`, `compute_poly` and `evalue_eval` evaluate a (piecewise) quasi-polynomial at a certain point. The argument `list_args` points to an array of `Value`s that is assumed to be long enough. The `double` return value of `compute_evalue` is inherited from `PolyLib`.

```
void print_evalue(FILE *DST, const evalue *e, char **pname);
```

The function `print_evalue` dumps a human-readable representation to the stream pointed to by `DST`. The argument `pname` points to an array of character strings representing the parameter names. The array is assumed to be long enough.

```
int eequal(const evalue *e1, const evalue *e2);
```

The function `eequal` return true (1) if its two arguments are structurally identical. I.e., it does *not* check whether the two (piecewise) quasi-polynomial represent the same function.

```
void reduce_evalue (evalue *e);
```

The function `reduce_evalue` performs some simplifications on `evalue`s. Here, we only describe the simplifications that are directly related to the internal representation. Some other simplifications are explained in Verdoolaege (2005, Section 4.7.2). If the highest order coefficients of a `polynomial`, `fractional` or `flooring` are zero (possibly after some other simplifications), then the size of the array is reduced. If only the constant term remains, i.e., the size is reduced to 1 for `polynomial` or to 2 for the other types, then the whole node is replaced by the constant term. Additionally, if the argument of a `fractional` has been reduced to a constant, then the whole node is replaced by its partial evaluation. A `relation` is similarly reduced if its second branch or both its branches are zero. Chambers with zero associated quasi-polynomials are discarded from a `partition`.

2.5 Generating Functions

The representation of rational generating functions uses some basic types from the `NTL` library (Shoup 2004) for representing arbitrary precision integers (`ZZ`) as well as vectors (`vec_ZZ`) and matrices (`mat_ZZ`) of such integers. We further introduces a type `QQ` for representing a rational number and use vectors (`vec_QQ`) of such numbers.

```
struct QQ {
    ZZ n;
    ZZ d;
};

NTL_vector_decl(QQ,vec_QQ);
```

Each term in a rational generating function is represented by a `short_rat` structure.

```
struct short_rat {
    struct {
        /* rows: terms in numerator */
        vec_QQ  coeff;
        mat_ZZ  power;
    } n;
    struct {
        /* rows: factors in denominator */
        mat_ZZ  power;
    } d;
};
```

The fields `n` and `d` represent the numerator and the denominator respectively. Note that in our implementation we combine terms with the same denominator. In the numerator, each element of `coeff` and each row of `power` represents a single such term. The vector `coeff` contains the rational coefficients α_i of each term. The columns of `power` correspond to the powers of the variables. In the denominator, each row of `power` corresponds to the power $\mathbf{b}_{ij}$ of a factor in the denominator.

`short_rat`

`n.coeff`	3	2
	2	1
`n.power`	2	3
	5	-7
`d.power`	1	-3
	0	2

Figure 2.6: Representation of $\left(\frac{3}{2}\,x_0^2x_1^3 + 2\,x_0^5x_1^{-7}\right) / \left((1 - x_0x_1^{-3})(1 - x_1^2)\right)$.

Example 2.5 *Figure 2.6 shows the internal representation of*

$$\frac{\frac{3}{2}\,x_0^2x_1^3 + 2\,x_0^5x_1^{-7}}{(1 - x_0x_1^{-3})(1 - x_1^2)}.$$

The whole rational generating function is represented by a `gen_fun` structure.

```
typedef std::set<short_rat *,
                 short_rat_lex_smaller_denominator > short_rat_list;

struct gen_fun {
    short_rat_list term;
    Polyhedron *context;

    void add(const QQ& c, const vec_ZZ& num, const mat_ZZ& den);
    void add(short_rat *r);
    void add(const QQ& c, const gen_fun *gf,
                barvinok_options *options);
    void substitute(Matrix *CP);
    gen_fun *Hadamard_product(const gen_fun *gf,
                              barvinok_options *options);
    void print(std::ostream& os,
               unsigned int nparam, char **param_name) const;
    operator evalue *() const;
    ZZ coefficient(Value* params, barvinok_options *options) const;
    void coefficient(Value* params, Value* c) const;

    gen_fun(Polyhedron *C);
    gen_fun(Value c);
    gen_fun(const gen_fun *gf);
    ~gen_fun();
};
```

A new `gen_fun` can be constructed either as empty rational generating function (possibly with a given context `C`), as a copy of an existing rational generating function `gf`, or as constant rational generating function with value for the constant term specified by `c`. The first `gen_fun::add` method adds a new term to the rational generating function, described by the coefficient `c`, the numerator `num` and the denominator `den`. It makes all powers in the denominator lexico-positive, orders them in lexicographical order and inserts the new term in `term` according to the lexicographical order of the combined powers in the denominator. The second `gen_fun::add` method adds `c` times `gf` to the rational generating function.

The method `gen_fun::operator evalue *` performs the conversion from rational generating function to piecewise step-polynomial explained in Verdoolaege (2005, Section 4.5.5). The `Polyhedron context` is the superset of all points where the enumerator is non-zero used during this conversion, i.e., it is the set Q from Verdoolaege (2005, Equation 4.31). If `context` is `NULL` the maximal allowed context is assumed, i.e., the maximal region with lexico-positive rays.

The method `gen_fun::coefficient` computes the coefficient of the term with power given by `params` and stores the result in `c`. This method performs essentially the same computations as `gen_fun::operator evalue *`, except that it adds extra equality constraints based on the specified values for the power.

The method `gen_fun::substitute` performs the monomial substitution specified by the homogeneous matrix `CP` that maps a set of "compressed parameters" (Meister 2004) to the original set of parameters. That is, if we are given a rational generating function $G(\mathbf{z})$ that encodes the explicit function $g(\mathbf{i}')$, where $\mathbf{i}'$ are the coordinates of the transformed space, and `CP` represents the map $\mathbf{i} = A\mathbf{i}' + \mathbf{a}$ back to the original space with coordinates $\mathbf{i}$, then this method transforms the rational generating function to $F(\mathbf{x})$ encoding the same explicit function $f(\mathbf{i})$, i.e.,

$$f(\mathbf{i}) = f(A\mathbf{i}' + \mathbf{a}) = g(\mathbf{i}').$$

This means that the coefficient of the term $\mathbf{x}^{\mathbf{i}} = \mathbf{x}^{A\mathbf{i}'+\mathbf{a}}$ in $F(\mathbf{x})$ should be equal to the coefficient of the term $\mathbf{z}^{\mathbf{i}'}$ in $G(\mathbf{z})$. In other words, if

$$G(\mathbf{z}) = \sum_i \epsilon_i \frac{\mathbf{z}^{\mathbf{v}_i}}{\prod_j (1 - \mathbf{z}^{\mathbf{b}_{ij}})}$$

then

$$F(\mathbf{x}) = \sum_i \epsilon_i \frac{\mathbf{x}^{A\mathbf{v}_i + \mathbf{a}}}{\prod_j (1 - \mathbf{x}^{A\mathbf{b}_{ij}})}.$$

The method `gen_fun::Hadamard_product` computes the Hadamard product of the current rational generating function with the rational generating function `gf`, as explained in Verdoolaege (2005, Section 4.5.2).

2.6 Counting Functions

Our library provides essentially three different counting functions: one for non-parametric polytopes, one for parametric polytopes and one for parametric sets with existential variables. The old versions of these functions have a "`MaxRays`" argument, while the new versions have a more general `barvinok_options` argument. For more information on `barvinok_options`, see Section 2.2.

```
void barvinok_count(Polyhedron *P, Value* result,
                    unsigned NbMaxCons);
void barvinok_count_with_options(Polyhedron *P, Value* result,
                                 struct barvinok_options *options);
```

The function `barvinok_count` or `barvinok_count_with_options` enumerates the non-parametric polytope `P` and returns the result in the `Value` pointed to by `result`, which needs to have been allocated and initialized. If `P` is a union, then only the first set in the union will be taken into account. For the meaning of the argument `NbMaxCons`, see the discussion on `MaxRays` in Section 2.2.

The function `barvinok_enumerate` for enumerating parametric polytopes was meant to be a drop-in replacement of `PolyLib`'s `Polyhedron_Enumerate` function. Unfortunately, the latter has been changed to accept an extra argument in recent versions of `PolyLib` as shown below.

```
Enumeration* barvinok_enumerate(Polyhedron *P, Polyhedron* C,
                                unsigned MaxRays);
extern Enumeration *Polyhedron_Enumerate(Polyhedron *P,
    Polyhedron *C, unsigned MAXRAYS, char **pname);
```

The argument `MaxRays` has the same meaning as the argument `NbMaxCons` above. The argument `P` refers to the $(d + n)$-dimensional polyhedron defining the parametric polytope. The argument `C` is an n-dimensional polyhedron containing extra constraints on the parameter space. Its primary use is to indicate how many of the dimensions in `P` refer to parameters as any constraint in `C` could equally well have been added to `P` itself. Note that the dimensions referring to the parameters should appear *last*. If either `P` or `C` is a union, then only the first set in the union will be taken into account. The result is a newly allocated `Enumeration`. As an alternative we also provide a function (`barvinok_enumerate_ev` or `barvinok_enumerate_with_options`) that returns an `evalue`.

```
evalue* barvinok_enumerate_ev(Polyhedron *P, Polyhedron* C,
                              unsigned MaxRays);
evalue* barvinok_enumerate_with_options(Polyhedron *P,
        Polyhedron* C, struct barvinok_options *options);
```

For enumerating parametric sets with existentially quantified variables, we provide two functions: `barvinok_enumerate_e`, and `barvinok_enumerate_isl`.

```
evalue* barvinok_enumerate_e(Polyhedron *P,
          unsigned exist, unsigned nparam, unsigned MaxRays);
evalue* barvinok_enumerate_e_with_options(Polyhedron *P,
          unsigned exist, unsigned nparam,
          struct barvinok_options *options);
evalue *barvinok_enumerate_isl(Polyhedron *P,
          unsigned exist, unsigned nparam,
          struct barvinok_options *options);
evalue *barvinok_enumerate_scarf(Polyhedron *P,
          unsigned exist, unsigned nparam,
          struct barvinok_options *options);
```

The first function tries the simplification rules from Verdoolaege (2005, Section 4.6.2) before resorting to the method based on Parametric Integer Programming (PIP) from Verdoolaege (2005, Section 4.6.3). The second function immediately applies the technique from Verdoolaege (2005, Section 4.6.3). The argument `exist` refers to the number of existential variables, whereas the argument `nparam` refers to the number of parameters. The order of the dimensions in `P` is: counted variables first, then existential variables and finally the parameters. The function `barvinok_enumerate_scarf` performs the same computation as the function `barvinok_enumerate_scarf_series` below, but produces an explicit representation instead of a generating function.

```
gen_fun * barvinok_series(Polyhedron *P, Polyhedron* C,
                          unsigned MaxRays);
gen_fun * barvinok_series_with_options(Polyhedron *P,
    Polyhedron* C, barvinok_options *options);
gen_fun *barvinok_enumerate_e_series(Polyhedron *P,
                  unsigned exist, unsigned nparam,
                  barvinok_options *options);
gen_fun *barvinok_enumerate_scarf_series(Polyhedron *P,
                          unsigned exist, unsigned nparam,
                          barvinok_options *options);
```

The function `barvinok_series` or `barvinok_series_with_options` enumerates parametric polytopes in the form of a rational generating function. The polyhedron `P` is assumed to have only revlex-positive rays.
The function `barvinok_enumerate_e_series` computes a generating function for the number of point in the parametric set defined by `P` with `exist` existentially quantified variables using the projection theorem, as explained in subsection 5.25. The function `barvinok_enumerate_scarf_series` computes a generating function for the number of point in the parametric set defined by `P` with `exist` existentially quantified variables, which is assumed to be 2. This function implements the technique of Scarf and Woods (2006) using the neighborhood complex description of Scarf (1981). It is currently restricted to problems with 3 or 4 constraints involving the existentially quantified variables.

2.7 Auxiliary Functions

In this section we briefly mention some auxiliary functions available in the `barvinok` library.

```
void Polyhedron_Polarize(Polyhedron *P);
```

The function `Polyhedron_Polarize` polarizes its argument and is explained in Verdoolaege (2005, Section 4.4.2).

```
int unimodular_complete(Matrix *M, int row);
```

The function `unimodular_complete` extends the first `row` rows of `M` with an integral basis of the orthogonal complement as explained in Section 5.7. Returns non-zero if the resulting matrix is unimodular.

```
int DomainIncludes(Polyhedron *D1, Polyhedron *D2);
```

The function `DomainIncludes` extends the function `PolyhedronIncludes` provided by `PolyLib` to unions of polyhedra. It checks whether every polyhedron in the union `D2` is included in some polyhedron of `D1`.

```
Polyhedron *DomainConstraintSimplify(Polyhedron *P,
                                     unsigned MaxRays);
```

The value returned by `DomainConstraintSimplify` is a pointer to a newly allocated `Polyhedron` that contains the same integer points as its first argument but possibly has simpler constraints. Each constraint $g\langle \mathbf{a}, \mathbf{x} \rangle \geq c$ is replaced by $\langle \mathbf{a}, \mathbf{x} \rangle \geq \left\lceil \frac{c}{g} \right\rceil$, where g is the greatest common divisor (gcd) of the coefficients in the original constraint. The `Polyhedron` pointed to by `P` is destroyed.

```
Polyhedron* Polyhedron_Project(Polyhedron *P, int dim);
```

The function `Polyhedron_Project` projects `P` onto its last `dim` dimensions.

```
Matrix *left_inverse(Matrix *M, Matrix **Eq);
```

The `left_inverse` function computes the left inverse of `M` as explained in Section 5.6.

```
Matrix *Polyhedron_Reduced_Basis(Polyhedron *P,
                                 struct barvinok_options *options);
```

`Polyhedron_Reduced_Basis` computes a generalized reduced basis of `P`, which is assumed to be a polytope, using the algorithm of Cook et al. (1993). See subsection 5.20 for more information. The basis vectors are stored in the rows of the matrix returned.

```
Vector *Polyhedron_Sample(Polyhedron *P,
                          struct barvinok_options *options);
```

`Polyhedron_Sample` returns an integer point of `P` or `NULL` if `P` contains no integer points. The integer point is found using the algorithm of Cook et al. (1993) and uses `Polyhedron_Reduced_Basis` to compute the reduced bases. See subsection 5.20 for more information.

3 Applications included in the `barvinok` distribution

This section describes some application programs provided by the `barvinok` library, available from `https://barvinok.sourceforge.io/`. For compilation instructions we refer to the `README` file included in the distribution.

Common option to all programs:

`--version`	`-V`	print version
`--help`	`-?`	list available options

3.1 `barvinok_count`

The program `barvinok_count` enumerates a non-parametric polytope. It takes one polytope in `PolyLib` notation as input and prints the number of integer points in the polytope. The `PolyLib` notation corresponds to the internal representation of `Polyhedron`s as explained in Section 2.1. The first line of the input contains the number of rows and the number of columns in the `Constraint` matrix. The rest of the input is composed of the elements of the matrix. Recall that the number of columns is two more than the number of variables, where the extra first columns is one or zero depending on whether the constraint is an inequality (≥ 0) or an equality ($= 0$). The next columns contain the coefficients of the variables and the final column contains the constant in the constraint. E.g., the set $S = \{ s \mid s \geq 0 \land 2s \leq 13 \}$ from Verdoolaege (2005, Example 38 on page 134) corresponds to the following input and output.

```
> cat S
2 3

1 1 0
1 -2 13
> ./barvinok_count  < S
POLYHEDRON Dimension:1
           Constraints:2  Equations:0  Rays:0  Lines:0
Constraints 2 3
Inequality: [   1    0  ]
Inequality: [  -1    6  ]
Rays 0 3
   7
```

Note that if you use `PolyLib` version 5.22.0 or newer then the output may look slightly different as the computation of the `Rays` may have been postponed to a later stage. The program `latte2polylib.pl` can be used to convert a polytope from `LattE` (De Loera et al. 2003) notation to `PolyLib` notation.

As an alternative to the constraints based input, the input polytope may also be specified by its `Ray` matrix. The first line of the input contains the single word `vertices`. The second line contains the number of rows and the number of columns in the `Ray` matrix. The rest of the input is composed of the elements of the matrix. Recall that the number of columns is two more than the number of variables, where the extra first columns is one or zero depending on whether the ray is a line or not. The next columns contain the numerators of the coordinates and the final column contains the denominator of the vertex or 0 for a ray. E.g., the above set can also be described as

```
vertices

2 3

1 0 1
1 13 2
```

3.2 barvinok_enumerate

The program `barvinok_enumerate` enumerates a parametric polytope as a piecewise step-polynomial or rational generating function. It takes two polytopes in `PolyLib` notation as input, optionally followed by a list of parameter names. The two polytopes refer to arguments `P` and `C` of the corresponding function. (See Section 2.6.) The following example was taken by Loechner (1999) from Loechner (1997, Chapter II.2).

```
> cat loechner
# Dimension of the matrix:
7 7
# Constraints:
# i j k P Q cte
1 1 0 0 0 0 0 # 0 <= i
1 -1 0 0 1 0 0 # i <= P
1 0 1 0 0 0 0 # 0 <= j
1 1 -1 0 0 0 0 # j <= i
1 0 0 1 0 0 0 # 0 <= k
1 1 -1 -1 0 0 0 # k <= i-j
0 1 1 1 0 -1 0 # Q = i + j + k

# 2 parameters, no constraints.
0 4
> ./barvinok_enumerate < loechner
         P - Q  >= 0
         Q  >= 0
          1 >= 0

( 1/8 * Q^2 + ( -1/2 * {( 1/2 * Q + 0 )
} + 3/4 )
 * Q + ( -5/4 * {( 1/2 * Q + 0 )
} + 1 )
 )
         2P - Q  >= 0
         - P + Q  -1 >= 0
          1 >= 0
```

```
( -1/2 * P^2 + ( 1 * Q + 1/2 )
 * P + ( -3/8 * Q^2 + ( -1/2 * {( 1/2 * Q + 0 )
} + 1/4 )
 * Q + ( -5/4 * {( 1/2 * Q + 0 )
} + 1 )
 )
 )
```

The output corresponds to

$$\begin{cases} -\frac{1}{2}P^2 + PQ + \frac{1}{2}P - \frac{3}{8}Q^2 + \left(\frac{1}{4} - \frac{1}{2}\left\{\frac{1}{2}Q\right\}\right)Q + 1 - \frac{5}{4}\left\{\frac{1}{2}Q\right\} & \\ & \text{if } P \le Q \le 2P \\ \frac{1}{8}Q^2 + \left(\frac{3}{4} - \frac{1}{2}\left\{\frac{1}{2}Q\right\}\right) - \frac{5}{4}\left\{\frac{1}{2}Q\right\} & \text{if } 0 \le Q \le P - 1. \end{cases}$$

The following is an example of Petr Lisoněk.

```
> cat petr
4 6
1 -1 -1 -1 1 0
1 1 -1 0 0 0
1 0 1 -1 0 0
1 0 0 1 0 -1

0 3
n
> ./barvinok_enumerate --series < petr
(n^3)/((1-n) * (1-n) * (1-n^2) * (1-n^3))
```

Options:

`--floor`	`-f`	convert `fractional`s to `flooring`s
`--convert`	`-c`	convert `fractional`s to `periodic`s
`--series`	`-s`	compute rational generating function instead of piecewise step-polynomial
`--explicit`	`-e`	convert computed rational generating function to a piecewise step-polynomial

3.3 `barvinok_enumerate_e`

The program `barvinok_enumerate_e` enumerates a parametric projected set. It takes a single polytope in `PolyLib` notation as input, followed by two lines indicating the number or existential variables and the number of parameters and optionally followed by a list of parameter names. The syntax for the line indicating the number of existential variables is the letter `E` followed by a space and the actual number. For indicating the number of parameters, the letter `P` is used. The following example corresponds to Verdoolaege (2005, Example 36 on page 129).

```
> cat projected
```

```
5 6
#   k   i   j   p   cst
1   0   1   0   0   -1
1   0   -1  0   0   8
1   0   0   1   0   -1
1   0   0   -1  1   0
0   -1  6   9   0   -7

E 2
P 1
> ./barvinok_enumerate_e <projected
         P  -3 >= 0
          1 >= 0

( 3 * P + 10 )
         P  -1 >= 0
         - P + 2 >= 0

( 8 * P + 0 )
```

Options:

`--floor`	`-f`	convert `fractional`s to `flooring`s
`--convert`	`-c`	convert `fractional`s to `periodic`s
`--isl`	`-i`	call `barvinok_enumerate_isl` instead of `barvinok_enumerate_e`

3.4 `barvinok_union`

The program `barvinok_union` enumerates a union of parametric polytopes. It takes as input the number of parametric polytopes in the union, the polytopes in combined data and parameter space in `PolyLib` notation, the context in parameter space in `PolyLib` notation and optionally a list of parameter names.

Options:

`--series`	`-s`	compute rational generating function instead of piecewise step-polynomial

3.5 `barvinok_ehrhart`

The program `barvinok_ehrhart` computes the Ehrhart quasi-polynomial of a polytope P, i.e., a quasi-polynomial in n that evaluates to the number of integer points in the dilation of P by a factor n. The input is the same as that of `barvinok_count`, except that it may be followed by the variable name. The functionality is the same as running `barvinok_enumerate` on the cone over P placed at $n = 1$.

Options:

`--floor`	`-f`	convert `fractional`s to `flooring`s
`--convert`	`-c`	convert `fractional`s to `periodic`s
`--series`	`-s`	compute Ehrhart series instead of Ehrhart quasi-polynomial

3.6 `polyhedron_sample`

The program `polyhedron_sample` takes a polytope in `PolyLib` notation and prints an integer point in the polytope if there is one. The point is computed using `Polyhedron_Sample`.

3.7 `polytope_scan`

The program `polytope_scan` takes a polytope in `PolyLib` notation and prints a list of all integer points in the polytope. Unless the `--direct` options is given, the order is based on the reduced basis computed with `Polyhedron_Reduced_Basis`.

Options:

`--direct`	`-d`	list the points in the lexicographical order

3.8 `lexmin`

The program `lexmin` implements an algorithm for performing PIP based on rational generating functions and provides an alternative for the technique of Feautrier (1988), which is based on cutting planes (Gomory 1963). The input is the same as that of the `example` program from `piplib` (Feautrier 2006), except that the value for the "big parameter" needs to be -1, since there is no need for big parameters, and it does not read any options from the input file.

3.9 `barvinok_summate`

Given a piecewise step-polynomial in `isl` format, the program `barvinok_summate` computes the sum of the piecewise quasi-polynomial evaluated in all (integer) values of the variables. The result is an expression in the parameters. Note that `barvinok_enumerate` and `barvinok_enumerate_e` can produce piecewise step-polynomials when given the `-I` option, but they will have only parameters and no variables.

For example

```
> cat square_p3.pwqp
[n] -> { [x, y] -> x * y :
          n >= -9 + 3x and x >= 2 and y >= 4 and y <= 5 }
> ./barvinok_summate < square_p3.pwqp
[n] -> { (45 + 63/2 * floor((n)/3) + 9/2 * floor((n)/3)^2) : n >= -3 }
```

Options:

`--summation`	specifies which summation method to use; `box` refers to the method of Verdoolaege (2005, Section 4.5.4), `bernoulli` refers to the method of subsection 5.13, `euler` refers to the method of subsection 5.14, and `laurent` refers to the method of subsection 5.16.

3.10 barvinok_bound

Given a piecewise step-polynomial in `isl` format, the program `barvinok_bound` computes an upper bound (or lower bound) for the values attained by the piecewise quasi-polynomial over all (integer) values of the variables. The result is an expression in the parameters. Note that `barvinok_enumerate` and `barvinok_enumerate_e` can produce piecewise step-polynomials when given the `-I` option, but they will have only parameters and no variables.

```
> cat devos.pwqp
[U] -> { [V] -> ((1/3 * U + 2/3 * V) - [(U + 2V)/3]) :
         2V >= -3 - U and 2V <= -U and U >= 0 and U <= 10 }
> ./barvinok_bound < devos.pwqp
[U] -> { max(2/3) : 0 <= U <= 10 }
```

Options:

`--lower`	compute lower bound instead of upper bound

3.11 polytope_minimize

The program `polytope_minimize` has been superseded by `isl`'s `isl_polyhedron_minimize`.

3.12 polyhedron_integer_hull

The program `polyhedron_integer_hull` takes a polyhedron in `PolyLib` notation and prints its integer hull. The integer hull is computed as explained in subsection 5.21.

3.13 polytope_lattice_width

The program `polytope_lattice_width` computes the lattice width of a parametric polytope. The input is the same as that of `barvinok_enumerate`. The lattice width is computed as explained in subsection 5.23.

Options:

`--direction`	`-d`	print the lattice width directions

4 `polymake` clients

The `barvinok` distribution includes a couple of `polymake` (Gawrilow and Joswig 2000) clients in the `polymake` subdir.

- `lattice_points <file>`

  Computes the property `LATTICE_POINTS` of a polytope, the number of lattice points in the polytope.

- `h_star_vector <file>`

  Computes the property `H_STAR_VECTOR` of a lattice polytope, the h^*-vector of the polytope (Stanley 1993).

5 Implementation details

5.1 An interior point of a polyhedron

We often need a point that lies in the interior of a polyhedron. The function `inner_point` implements the following algorithm. Each polyhedron P can be written as the sum of a polytope P' and a cone C (the recession cone or characteristic cone of P). Adding a positive multiple of the sum of the extremal rays of C to the barycenter

$$\frac{1}{N}\sum_i \mathbf{v}_i(\mathbf{p})$$

of P', where N is the number of vertices, results in a point in the interior of P.

5.2 The integer points in the fundamental parallelepiped of a simple cone

This section is based on Barvinok (1992, Lemma 5.1) and De Loera and Köppe (2006).

In this section we will deal exclusively with simple cones, i.e. d-dimensional cones with d extremal rays and d facets. Some of the facets of these cones may be open. Since we will mostly be dealing with cones in their explicit representation, we will have occasion to speak of "open rays", by which we will mean that the facet not containing the ray is open. (There is only one such facet because the cone is simple.)

Definition 5.1 (Fundamental parallelepiped) *Let* $K = \mathbf{v} + \operatorname{pos}\{\, \mathbf{u}_i \,\}$ *be a closed (shifted) cone, then the* fundamental parallelepiped Π *of* K *is*

$$\Pi = \mathbf{v} + \left\{ \sum_i \alpha_i \mathbf{u}_i \mid 0 \le \alpha_i < 1 \right\}.$$

If some of the rays $\mathbf{u}_i$ *of* K *are open, then the constraints on the corresponding coefficient* α_i *are such that* $0 < \alpha_i \le 1$.

Lemma 5.2 (Integer points in the fundamental parallelepiped of a simple cone) *Let* $K = \mathbf{v} + \operatorname{pos}\{\, \mathbf{u}_i \,\}$ *be a closed simple cone and let* A *be the matrix with the generators* $\mathbf{u}_i$ *of* K *as rows. Furthermore let* $VAW^{-1} = S = \operatorname{diag} \mathbf{s}$ *be the Smith Normal Form (SNF) of* A*. Then the integer points in the fundamental parallelepiped of* K *are given by*

$$\begin{aligned} \mathbf{w}^T &= \mathbf{v}^T + \left\{ (\mathbf{k}^T W - \mathbf{v}^T) A^{-1} \right\} A \qquad (5.3) \\ &= \mathbf{v}^T + \sum_{i=1}^{d} \left\{ \langle \sum_{j=1}^{d} k_j \mathbf{w}_j^T - \mathbf{v}^T, \mathbf{u}_i^* \rangle \right\} \mathbf{u}_i^T, \end{aligned}$$

where $\mathbf{u}_i^*$ *are the columns of* A^{-1} *and* $k_j \in \mathbb{Z}$ *ranges over* $0 \le k_j < s_j$.

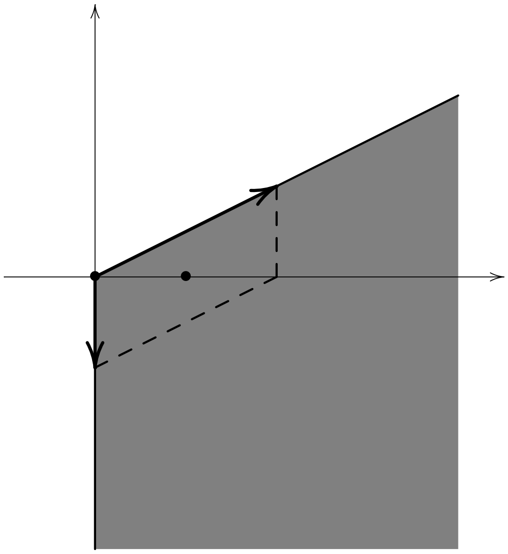


Figure 5.4: The integer points in the fundamental parallelepiped of K

Proof Since $0 \le \{x\} < 1$, it is clear that each such $\mathbf{w}$ lies inside the fundamental parallelepiped. Furthermore,

$$\begin{aligned}
\mathbf{w}^T &= \mathbf{v}^T + \left\{(\mathbf{k}^T W - \mathbf{v}^T)A^{-1}\right\} A \\
&= \mathbf{v}^T + \left((\mathbf{k}^T W - \mathbf{v}^T)A^{-1} - \left\lfloor (\mathbf{k}^T W - \mathbf{v}^T)A^{-1} \right\rfloor\right) A \\
&= \underbrace{\mathbf{k}^T W}_{\in \mathbb{Z}^{1\times d}} - \underbrace{\left\lfloor (\mathbf{k}^T W - \mathbf{v}^T)A^{-1} \right\rfloor}_{\in \mathbb{Z}^{1\times d}} \underbrace{A}_{\in \mathbb{Z}^{d\times d}} \in \mathbb{Z}^{1\times d}.
\end{aligned}$$

Finally, if two such $\mathbf{w}$ are equal, i.e., $\mathbf{w}_1 = \mathbf{w}_2$, then

$$\begin{aligned}
\mathbf{0}^T = \mathbf{w}_1^T - \mathbf{w}_2^T &= \mathbf{k}_1^T W - \mathbf{k}_2^T W + \mathbf{p}^T A \\
&= (\mathbf{k}_1^T - \mathbf{k}_2^T)\, W + \mathbf{p}^T V^{-1} S\, W,
\end{aligned}$$

with $\mathbf{p} \in \mathbb{Z}^d$, or $\mathbf{k}_1 \equiv \mathbf{k}_2 \mod \mathbf{s}$, i.e., $\mathbf{k}_1 = \mathbf{k}_2$. Since $\det S = |\det A|$, we obtain all points in the fundamental parallelepiped by taking all $\mathbf{k} \in \mathbb{Z}^d$ satisfying $0 \le k_j < s_j$.

□

If the cone K is not closed then the coefficients of the open rays should be in $(0, 1]$ rather than in $[0, 1)$. In (5.3), we therefore need to replace the fractional part $\{x\} = x - \lfloor x \rfloor$ by $\{\{x\}\} = x - \lceil x - 1 \rceil$ for the open rays.

Example 5.5 *Let K be the cone*

$$K = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \operatorname{pos} \left\{ \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\},$$

shown in Figure 5.4. Then

$$A = \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} 1/2 & 1/2 \\ 0 & -1 \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}.$$

We have $|\det A| = \det S = 2$ *and* $\mathbf{k}_1^T = \begin{bmatrix} 0 & 0 \end{bmatrix}$ *and* $\mathbf{k}_2^T = \begin{bmatrix} 0 & 1 \end{bmatrix}$. *Therefore,*

$$\mathbf{w}_1^T = \left\{ \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1/2 & 1/2 \\ 0 & -1 \end{bmatrix} \right\} \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

and

$$\begin{aligned} \mathbf{w}_2^T &= \left\{ \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1/2 & 1/2 \\ 0 & -1 \end{bmatrix} \right\} \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 1/2 & 1/2 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix}. \end{aligned}$$

5.3 Barvinok's decomposition of simple cones in primal space

As described by De Loera et al. (2004), the first implementation of Barvinok's counting algorithm applied Barvinok's decomposition (Barvinok 1994) in the dual space. Brion's polarization trick (Brion 1988) then ensures that you do not need to worry about lower-dimensional faces in the decomposition. Another way of avoiding the lower-dimensional faces, in the primal space, is to perturb the vertex of the cone such that none of the lower-dimensional face encountered contain any integer points (Köppe 2007). In this section, we describe another technique that is based on allowing some of the facets of the cone to be open.

The basic step in Barvinok's decomposition is to replace a d-dimensional simple cone $K = \operatorname{pos}\{\mathbf{u}_i\}_{i=1}^d \subset \mathbb{Q}^d$ by a signed sum of (at most) d cones K_j with a smaller determinant (in absolute value). The cones are obtained by successively replacing each generator of K by an appropriately chosen $\mathbf{w} = \sum_{i=1}^d \alpha_i \mathbf{u}_i$, i.e.,

$$K_j = \operatorname{pos}\left(\{\mathbf{u}_i\}_{i=1}^d \setminus \{\mathbf{u}_j\} \cup \{\mathbf{w}\} \right). \tag{5.6}$$

To see that we can use these K_j to perform a decomposition, rearrange the $\mathbf{u}_i$ such that for all $1 \le i \le k$ we have $\alpha_i < 0$ and for all $k+1 \le i \le d'$ we have $\alpha_i > 0$, with $d - d'$ the number of zero α_i. We may assume $k < d'$; otherwise replace $\mathbf{w} \in B$ by $-\mathbf{w} \in B$. We have

$$\mathbf{w} + \sum_{i=1}^{k} (-\alpha_i)\mathbf{u}_i = \sum_{i=k+1}^{d'} \alpha_i \mathbf{u}_i$$

or

$$\sum_{i=0}^{k} \beta_i \mathbf{u}_i = \sum_{i=k+1}^{d'} \alpha_i \mathbf{u}_i, \tag{5.7}$$

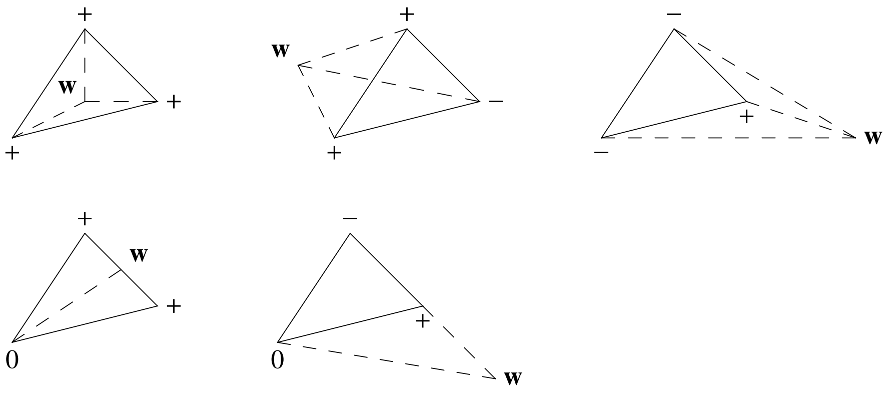


Figure 5.11: Possible locations of **w** with respect to the rays of a 3-dimensional cone. The figure shows a section of the cones.

with $\mathbf{u}_0 = \mathbf{w}$, $\beta_0 = 1$ and $\beta_i = -\alpha_i > 0$ for $1 \le i \le k$. Any two $\mathbf{u}_j$ and $\mathbf{u}_l$ on the same side of the equality are on opposite sides of the linear hull H of the other $\mathbf{u}_i$s since there exists a convex combination of $\mathbf{u}_j$ and $\mathbf{u}_l$ on this hyperplane. In particular, since α_j and α_l have the same sign, we have

$$\frac{\alpha_j}{\alpha_j + \alpha_l}\mathbf{u}_j + \frac{\alpha_l}{\alpha_j + \alpha_l}\mathbf{u}_l \in H \qquad \text{for } \alpha_i\alpha_l > 0. \tag{5.8}$$

The corresponding cones K_j and K_l (with $K_0 = K$) therefore intersect in a common face $F \subset H$. Let

$$K' := \operatorname{pos}\Big(\{\,\mathbf{u}_i\,\}_{i=1}^{d} \cup \{\,\mathbf{w}\,\}\Big),$$

then any $\mathbf{x} \in K'$ lies both in some cone K_i with $0 \le i \le k$ and in some cone K_i with $k+1 \le i \le d'$. (Just subtract an appropriate multiple of Equation (5.7).) The cones $\{\,K_i\,\}_{i=0}^{k}$ and $\{\,K_i\,\}_{i=k+1}^{d'}$ therefore both form a triangulation of K' and hence

$$[K'] = [K] + \sum_{i=1}^{k} [K_i] - \sum_{j \in J_1} \big[F_j\big] = \sum_{i=k+1}^{d'} [K_i] - \sum_{j \in J_2} \big[F_j\big] \tag{5.9}$$

or

$$[K] = \sum_{i=1}^{d'} \varepsilon_i \,[K_i] + \sum_{j} \delta_j \big[F_j\big], \tag{5.10}$$

with $\varepsilon_i = -1$ for $1 \le i \le k$, $\varepsilon_i = 1$ for $k+1 \le i \le d'$, $\delta_j \in \{-1, 1\}$ and F_j some lower-dimensional faces. Figure 5.11 shows the possible configurations in the case of a 3-dimensional cone.

As explained above there are several ways of avoiding the lower-dimensional faces in (5.10). Here we will apply the following proposition.

Proposition 5.12 (Köppe and Verdoolaege (2008)) *Let*

$$\sum_{i\in I_1} \epsilon_i[P_i] + \sum_{i\in I_2} \delta_k[P_i] = 0 \tag{5.13}$$

be a (finite) linear identity of indicator functions of closed polyhedra $P_i \subseteq \mathbb{Q}^d$, where the polyhedra P_i with $i \in I_1$ are full-dimensional and those with $i \in I_2$ lower-dimensional. Let each closed polyhedron be given as

$$P_i = \left\{ \mathbf{x} \mid \langle \mathbf{b}^*_{i,j}, \mathbf{x}\rangle \geq \beta_{i,j} \textit{ for } j \in J_i \right\}.$$

Let $\mathbf{y} \in \mathbb{Q}^d$ be a vector such that $\langle \mathbf{b}^*_{i,j}, \mathbf{y}\rangle \neq 0$ for all $i \in I_1 \cup I_2$, $j \in J_i$. For each $i \in I_1$, we define the half-open polyhedron

$$\begin{aligned}\tilde{P}_i = \Big\{ \mathbf{x} \in \mathbb{Q}^d \mid \langle \mathbf{b}^*_{i,j}, \mathbf{x}\rangle \geq \beta_{i,j} \textit{ for } j \in J_i \textit{ with } \langle \mathbf{b}^*_{i,j}, \mathbf{y}\rangle > 0,\\ \langle \mathbf{b}^*_{i,j}, \mathbf{x}\rangle > \beta_{i,j} \textit{ for } j \in J_i \textit{ with } \langle \mathbf{b}^*_{i,j}, \mathbf{y}\rangle < 0 \Big\}.\end{aligned} \tag{5.14}$$

Then

$$\sum_{i\in I_1} \epsilon_i[\tilde{P}_i] = 0. \tag{5.15}$$

When applying this proposition to (5.10), we obtain

$$\left[\tilde{K}\right] = \sum_{i=1}^{d'} \varepsilon_i \left[\tilde{K}_i\right], \tag{5.16}$$

where we start out from a given $\tilde{K}$, which may be K itself, i.e., a fully closed cone, or the result of a previous application of the proposition, either through a triangulation (Section 5.4) or a previous decomposition. In either case, a suitable $\mathbf{y}$ is available, either as an interior point of the cone or as the vector used in the previous application (which may require a slight perturbation if it happens to lie on one of the new facets of the cones K_i). We are, however, free to construct a new $\mathbf{y}$ on each application of the proposition. In fact, we will not even construct such a vector explicitly, but rather apply a set of rules that is equivalent to a valid choice of $\mathbf{y}$. Below, we will present an "intuitive" motivation for these rules. For a more algebraic, shorter, and arguably simpler motivation we refer to Köppe and Verdoolaege (2008).

The vector $\mathbf{y}$ has to satisfy $\langle \mathbf{b}^*_j, \mathbf{y}\rangle > 0$ for normals $\mathbf{b}^*_j$ of closed facets and $\langle \mathbf{b}^*_j, \mathbf{y}\rangle < 0$ for normals $\mathbf{b}^*_j$ of open facets of $\tilde{K}$. These constraints delineate a non-empty open cone R from which $\mathbf{y}$ should be selected. For some of the new facets of the cones $\tilde{K}_j$, the cone R will not be cut by the affine hull of the facet. The closedness of these facets is therefore predetermined by $\tilde{K}$. For the other facets, a choice will have to be made. To be able to make the choice based on local information and without computing an explicit vector $\mathbf{y}$, we use the following convention. We first assign an arbitrary total order to the rays. If (the affine hull of) a facet separates the two rays not on the facet $\mathbf{u}_i$ and $\mathbf{u}_j$, i.e., $\alpha_i \alpha_j > 0$ (5.8), then we choose $\mathbf{y}$ to lie on the side of the smallest ray, according to the chosen order. That is, $\langle \tilde{\mathbf{n}}_{ij}, \mathbf{y}\rangle > 0$, for $\tilde{\mathbf{n}}_{ij}$ the normal of the facet pointing towards this smallest ray. Otherwise, i.e., if $\alpha_i\alpha_j < 0$, the interior of K will lie on one side of the facet and then we choose $\mathbf{y}$ to lie on the other side. That is, $\langle \tilde{\mathbf{n}}_{ij}, \mathbf{y} \rangle > 0$, for $\tilde{\mathbf{n}}_{ij}$ the normal of the facet pointing away from the cone K. Figure 5.17 shows some example decompositions with an explicitly marked $\mathbf{y}$.

To see that there is a $\mathbf{y}$ satisfying the above constraints, we need to show that $R \cap S$ is non-empty, with $S = \{\mathbf{y} \mid \langle \tilde{\mathbf{n}}_{i_k j_k}, \mathbf{y} \rangle > 0 \text{ for all } k\}$. It will be easier to show this set is non-empty when the $\mathbf{u}_i$ form an orthogonal basis. Applying a non-singular linear transformation T does not change the decomposition of $\mathbf{w}$ in terms of the $\mathbf{u}_i$ (i.e., the α_i remain unchanged), nor does this change any of the scalar products in the constraints that define $R \cap S$ (the normals are transformed by $\left(T^{-1}\right)^T$). Finding a vector $\mathbf{y} \in T(R \cap S)$ ensures that $T^{-1}(\mathbf{y}) \in R \cap S$. Without loss of generality, we can therefore assume for the purpose of showing that $R \cap S$ is non-empty that the $\mathbf{u}_i$ indeed form an orthogonal basis.

In the orthogonal basis, we have $\mathbf{b}_i^* = \mathbf{u}_i$ and the corresponding inward normal $\mathbf{N}_i$ is either $\mathbf{u}_i$ or $-\mathbf{u}_i$. Furthermore, each normal of a facet of S of the first type is of the form $\tilde{\mathbf{n}}_{i_k j_k} = a_k\mathbf{u}_{i_k} - b_k\mathbf{u}_{j_k}$, with $a_k, b_k > 0$ and $i_k < j_k$, while for the second type each normal is of the form $\tilde{\mathbf{n}}_{i_k j_k} = -a_k\mathbf{u}_{i_k} - b_k\mathbf{u}_{j_k}$, with $a_k, b_k > 0$. If $\tilde{\mathbf{n}}_{i_k j_k} = a_k\mathbf{u}_{i_k} - b_k\mathbf{u}_{j_k}$ is the normal of a facet of S then either $(\mathbf{N}_{i_k}, \mathbf{N}_{j_k}) = (\mathbf{u}_{i_k}, \mathbf{u}_{j_k})$ or $(\mathbf{N}_{i_k}, \mathbf{N}_{j_k}) = (-\mathbf{u}_{i_k}, -\mathbf{u}_{j_k})$. Otherwise, the facet would not cut R. Similarly, if $\tilde{\mathbf{n}}_{i_k j_k} = -a_k\mathbf{u}_{i_k} - b_k\mathbf{u}_{j_k}$ is the normal of a facet of S then either $(\mathbf{N}_{i_k}, \mathbf{N}_{j_k}) = (\mathbf{u}_{i_k}, -\mathbf{u}_{j_k})$ or $(\mathbf{N}_{i_k}, \mathbf{N}_{j_k}) = (-\mathbf{u}_{i_k}, \mathbf{u}_{j_k})$. Assume now that $R \cap S$ is empty, then there exist $\lambda_k, \mu_i \geq 0$ not all zero such that $\sum_k \lambda_k \tilde{\mathbf{n}}_{i_k j_k} + \sum_l \mu_i \mathbf{N}_i = \mathbf{0}$. Assume $\lambda_k > 0$ for some facet of the first type. If $\mathbf{N}_{j_k} = -\mathbf{u}_{j_k}$, then $-b_k$ can only be canceled by another facet k' of the first type with $j_k = i_{k'}$, but then also $\mathbf{N}_{j_{k'}} = -\mathbf{u}_{j_{k'}}$. Since the j_k are strictly increasing, this sequence has to stop with a strictly positive coefficient for the largest $\mathbf{u}_{j_k}$ in this sequence. If, on the other hand, $\mathbf{N}_{i_k} = \mathbf{u}_{i_k}$, then a_k can only be canceled by the normal of a facet k' of the second kind with $i_k = j_{k'}$, but then $\mathbf{N}_{i_{k'}} = -\mathbf{u}_{i_{k'}}$ and we return to the first case. Finally, if $\lambda_k > 0$ only for normals of facets of the second type, then either $\mathbf{N}_{i_k} = -\mathbf{u}_{i_k}$ or $\mathbf{N}_{j_k} = -\mathbf{u}_{j_k}$ and so the coefficient of one of these basis vectors will be strictly negative. That is, the sum of the normals will never be zero and the set $R \cap S$ is non-empty.

For each ray $\mathbf{u}_j$ of cone K_i, i.e., the cone with $\mathbf{u}_i$ replaced by $\mathbf{w}$, we now need to determine whether the facet not containing this ray is closed or not. We denote the (inward) normal of this cone by $\mathbf{n}_{ij}$. Note that cone K_j (if it appears in (5.9), i.e., $\alpha_j \neq 0$) has the same facet opposite $\mathbf{u}_i$ and its normal $\mathbf{n}_{ji}$ will be equal to either $\mathbf{n}_{ij}$ or $-\mathbf{n}_{ij}$, depending on whether we are dealing with an "external" facet, i.e., a facet of K', or an "internal" facet. If, on the other hand, $\alpha_j = 0$, then $\mathbf{n}_{ij} = \mathbf{n}_{0j}$. If $\langle \mathbf{n}_{ij}, \mathbf{y} \rangle > 0$, then the facet is closed. Otherwise it is open. It follows that the two (or more) occurrences of external facets are either all open or all closed, while for internal facets, exactly one is closed.

First consider the facet not containing $\mathbf{u}_0 = \mathbf{w}$. If $\alpha_i > 0$, then $\mathbf{u}_i$ and $\mathbf{w}$ are on the same side of the facet and so $\mathbf{n}_{i0} = \mathbf{n}_{0i}$. Otherwise, $\mathbf{n}_{i0} = -\mathbf{n}_{i0}$. Second, if $\alpha_j = 0$, then replacing $\mathbf{u}_i$ by $\mathbf{w}$ does not change the affine hull of the facet and so $\mathbf{n}_{ij} = \mathbf{n}_{0j}$. Now consider the case that $\alpha_i\alpha_j < 0$, i.e., $\mathbf{u}_i$ and $\mathbf{u}_j$ are on the same side of the hyperplane through the other rays. If we project $\mathbf{u}_i$, $\mathbf{u}_j$ and $\mathbf{w}$ onto a plane orthogonal to the ridge through the other rays, then the possible locations of $\mathbf{w}$ with respect to $\mathbf{u}_i$ and $\mathbf{u}_j$ are

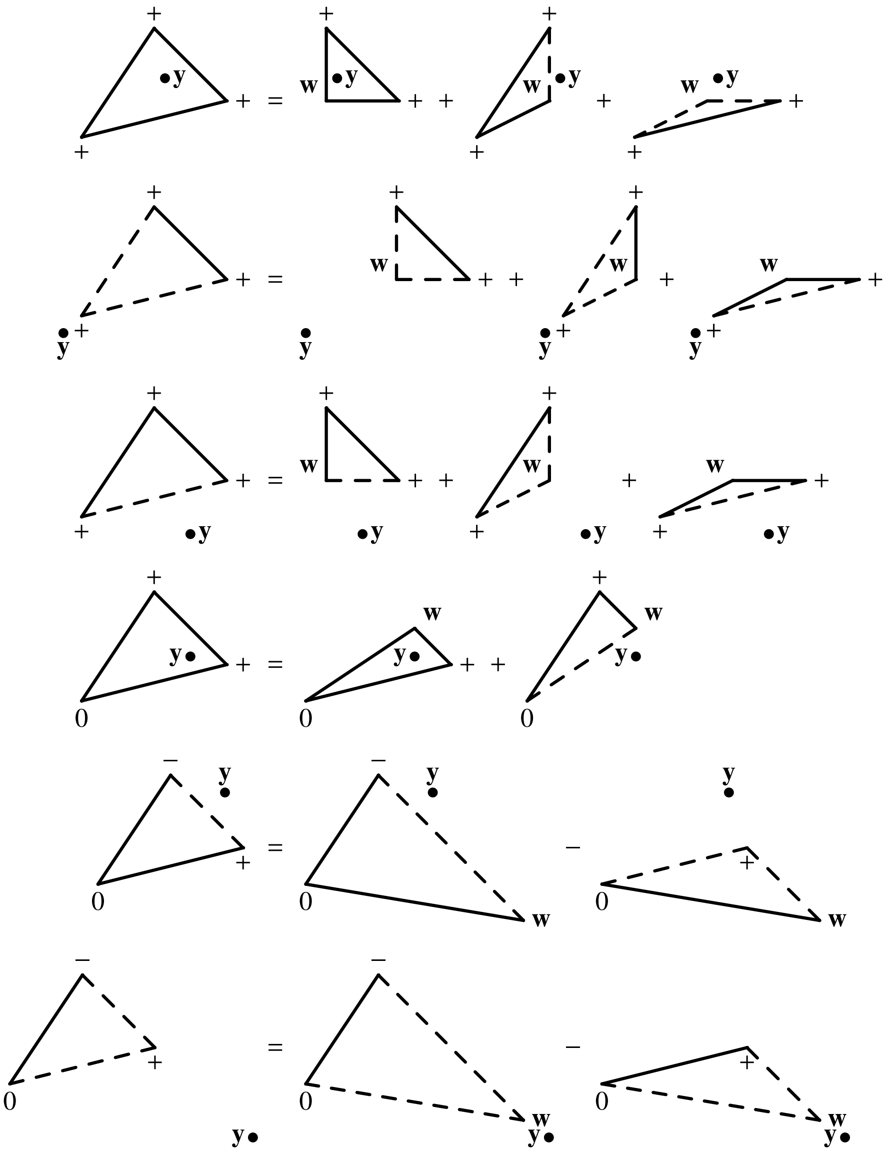


Figure 5.17: Examples of decompositions in primal space.

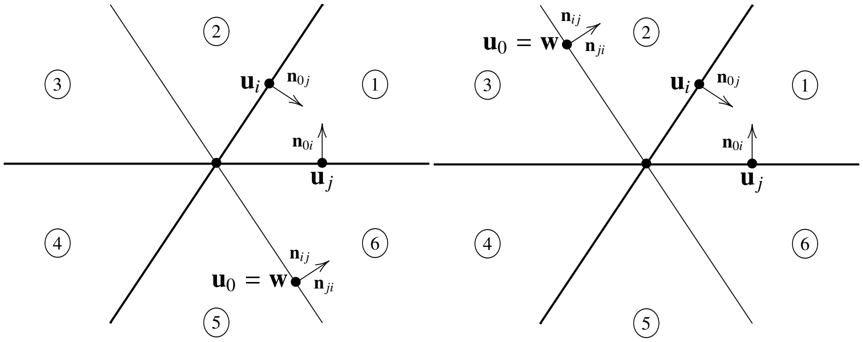


Figure 5.18: Possible locations of $\mathbf{w}$ with respect to $\mathbf{u}_i$ and $\mathbf{u}_j$, projected onto a plane orthogonal to the other rays, when $\alpha_i \alpha_j < 0$.

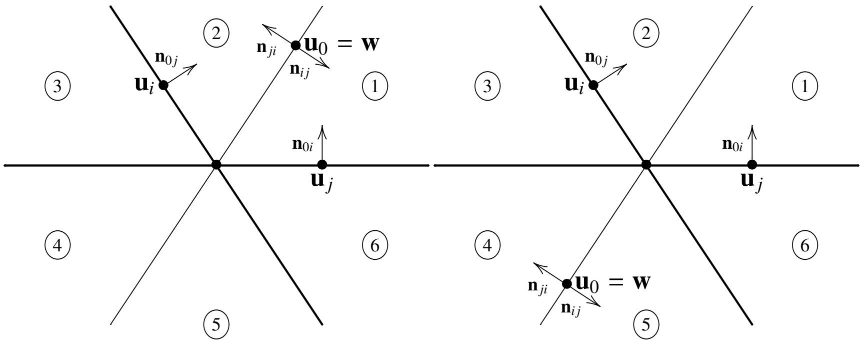


Figure 5.19: Possible locations of $\mathbf{w}$ with respect to $\mathbf{u}_i$ and $\mathbf{u}_j$, projected onto a plane orthogonal to the other rays, when $\alpha_i \alpha_j > 0$.

shown in Figure 5.18. If both $\mathbf{n}_{0i}$ and $\mathbf{n}_{0j}$ are closed then $\mathbf{y}$ lies in region 1 and therefore $\mathbf{n}_{ij}$ (as well as $\mathbf{n}_{ji}$) is closed too. Similarly, if both $\mathbf{n}_{0i}$ and $\mathbf{n}_{0j}$ are open then so is $\mathbf{n}_{ij}$. If only one of the facets is closed, then, as explained above, we choose $\mathbf{n}_{ij}$ to be open, i.e., we take $\mathbf{y}$ to lie in region 3 or 5. Figure 5.19 shows the possible configurations for the case that $\alpha_i \alpha_j > 0$. If exactly one of $\mathbf{n}_{0i}$ and $\mathbf{n}_{0j}$ is closed, then $\mathbf{y}$ lies in region 3 or region 5 and therefore $\mathbf{n}_{ij}$ is closed iff $\mathbf{n}_{0j}$ is closed. Otherwise, as explained above, we choose $\mathbf{n}_{ij}$ to be closed if $i < j$.

The algorithm is summarized in Algorithm 1, where we use the convention that in cone K_i, $\mathbf{u}_i$ refers to $\mathbf{u}_0 = \mathbf{w}$. Note that we do not need any of the rays or normals in this code. The only information we need is the closedness of the facets in the original cone and the signs of the α_i.

5.4 Triangulation in primal space

As in the case for Barvinok's decomposition (Section 5.3), we can transform a triangulation of a (closed) cone into closed simple cones into a triangulation of half-open simple cones that fully partitions the original cone, i.e., such that the half-open sim-

Algorithm 1 Determine whether the facet opposite $\mathbf{u}_j$ is closed in K_i.

```
if $\alpha_j = 0$
    closed[$K_i$][$\mathbf{u}_j$] := closed[$\tilde{K}$][$\mathbf{u}_j$]
else if $i = j$
    if $\alpha_j > 0$
        closed[$K_i$][$\mathbf{u}_j$] := closed[$\tilde{K}$][$\mathbf{u}_j$]
    else
        closed[$K_i$][$\mathbf{u}_j$] := $\neg$closed[$\tilde{K}$][$\mathbf{u}_j$]
else if $\alpha_i \alpha_j > 0$
    if closed[$\tilde{K}$][$\mathbf{u}_i$] = closed[$\tilde{K}$][$\mathbf{u}_j$]
        closed[$K_i$][$\mathbf{u}_j$] := $i < j$
    else
        closed[$K_i$][$\mathbf{u}_j$] := closed[$\tilde{K}$][$\mathbf{u}_j$]
else
    closed[$K_i$][$\mathbf{u}_j$] := closed[$\tilde{K}$][$\mathbf{u}_i$] and closed[$\tilde{K}$][$\mathbf{u}_j$]
```

ple cones do not intersect at their facets. Again, we apply Proposition 5.12 with $\mathbf{y}$ an interior point of the cone (Section 5.1). Note that the interior point $\mathbf{y}$ may still intersect some of the internal facets, so we may need to perturb it slightly. In practice, we apply a lexicographical rule: for such (internal) facets, which always appear in pairs, we close the one with a lexico-positive normal and open the one with a lexico-negative normal.

5.5 Multivariate quasi-polynomials as lists of polynomials

There are many definitions for a (univariate) quasi-polynomial. Ehrhart (1977) uses a definition based on *periodic number*s.

Definition 5.20 *A rational* periodic number $U(p)$ *is a function* $\mathbb{Z} \to \mathbb{Q}$*, such that there exists a* period q *such that* $U(p) = U(p')$ *whenever* $p \equiv p' \mod q$.

Definition 5.21 *A (univariate)* quasi-polynomial f *of degree* d *is a function*

$$f(n) = c_d(n)\, n^d + \cdots + c_1(n)\, n + c_0,$$

where $c_i(n)$ *are rational periodic numbers. I.e., it is a polynomial expression of degree* d *with rational periodic numbers for coefficients. The* period *of a quasi-polynomial is the lcm of the periods of its coefficients.*

Other authors (e.g., Stanley 1986) use the following definition of a quasi-polynomial.

Definition 5.22 *A function* $f : \mathbb{Z} \to \mathbb{Q}$ *is a (univariate)* quasi-polynomial *of period* q *if there exists a list of* q *polynomials* $g_i \in \mathbb{Q}[T]$ *for* $0 \leq i < q$ *such that*

$$f(s) = g_i(s) \qquad \text{if } s \equiv i \mod q.$$

The functions g_i *are called the* constituents.

In our implementation, we use Definition 5.21, but whereas Ehrhart (1977) uses a list of q rational numbers enclosed in square brackets to represent periodic numbers, our periodic numbers are polynomial expressions in fractional parts (Section 2.3). These fractional parts naturally extend to multivariate quasi-polynomials. The bracketed ("explicit") periodic numbers can be extended to multiple variables by nesting them (e.g., Loechner 1999).

Definition 5.22 could be extended in a similar way by having a constituent for each residue modulo a vector period $\mathbf{q}$. However, as pointed out by Woods (2006), this may not result in the minimum number of constituents. A vector period can be considered as a lattice with orthogonal generators and the number of constituents is equal to the index or determinant of that lattice. By considering more general lattices, we can potentially reduce the number of constituents.

Definition 5.23 *A function $f : \mathbb{Z}^n \to \mathbb{Q}$ is a (multivariate)* quasi-polynomial *of period L if there exists a list of* $\det L$ *polynomials $g_{\mathbf{i}} \in \mathbb{Q}[T_1, \ldots, T_n]$ for $\mathbf{i}$ in the fundamental parallelepiped of L such that*

$$f(\mathbf{s}) = g_{\mathbf{i}}(\mathbf{s}) \qquad \textit{if}\ \mathbf{s} \equiv \mathbf{i} \mod L.$$

To compute the period lattice from a fractional representation, we compute the appropriate lattice for each fractional part and then take their intersection. Recall that the argument of each fractional part is an affine expression in the parameters $(\langle \mathbf{a}, \mathbf{p} \rangle + c)/m$, with $\mathbf{a} \in \mathbb{Z}^n$ and $c, m \in \mathbb{Z}$. Such a fractional part is translation invariant over any (integer) value of $\mathbf{p}$ such that $\langle \mathbf{a}, \mathbf{p} \rangle + mt = 0$, for some $\mathbf{t} \in \mathbb{Z}$. Solving this homogeneous equation over the integers (in our implementation, we use `PolyLib`'s `SolveDiophantine`) gives the general solution

$$\begin{bmatrix} \mathbf{p} \\ t \end{bmatrix} = \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \mathbf{x} \qquad \text{for } \mathbf{x} \in \mathbb{Z}^n.$$

The matrix $U_1 \in \mathbb{Z}^{n \times n}$ then has the generators of the required lattice as columns. The constituents are computed by plugging in each integer point in the fundamental parallelepiped of the lattice. These points themselves are computed as explained in Section 5.2. Note that for computing the constituents, it is sufficient to take any representative of the residue class. For example, we could take $\mathbf{w}^T = \mathbf{k}^T W$ in the notations of Lemma 5.2.

Example 5.24 (Woods (2006)) *Consider the parametric polytope*

$$P_{s,t} = \{\, x \mid 0 \le x \le (s+t)/2 \,\}.$$

The enumerator of $P_{s,t}$ is

$$\begin{cases} \frac{s}{2} + \frac{t}{2} + 1 & \textit{if} \begin{bmatrix} s \\ t \end{bmatrix} \in \begin{bmatrix} -1 & -2 \\ 1 & 0 \end{bmatrix} \mathbb{Z}^2 + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \frac{s}{2} + \frac{t}{2} + \frac{1}{2} & \textit{if} \begin{bmatrix} s \\ t \end{bmatrix} \in \begin{bmatrix} -1 & -2 \\ 1 & 0 \end{bmatrix} \mathbb{Z}^2 + \begin{bmatrix} -1 \\ 0 \end{bmatrix}. \end{cases}$$

The corresponding output of `barvinok_enumerate` *is*

```
         s + t  >= 0
          1 >= 0

Lattice:
[[-1 1]
[-2 0]
]
[0 0]
( 1/2 * s + ( 1/2 * t + 1 )
 )
[-1 0]
( 1/2 * s + ( 1/2 * t + 1/2 )
 )
```

5.6 Left inverse of an affine embedding

We often map a polytope onto a lower dimensional space to remove possible equalities in the polytope. These maps are typically represented by the inverse, mapping the coordinates $\mathbf{x}'$ of the lower-dimensional space to the coordinates $\mathbf{x}$ of (an affine subspace of) the original space, i.e.,

$$\begin{bmatrix}\mathbf{x}\\1\end{bmatrix} = \begin{bmatrix}T & \mathbf{v}\\ \mathbf{0}^T & 1\end{bmatrix}\begin{bmatrix}\mathbf{x}'\\1\end{bmatrix},$$

where, as usual in `PolyLib`, we work with homogeneous coordinates. To obtain the transformation that maps the coordinates of the original space to the coordinates of the lower dimensional space, we need to compute the left inverse of the above affine embedding, i.e., an A, $\mathbf{b}$ and d such that

$$d\begin{bmatrix}\mathbf{x}'\\1\end{bmatrix} = \begin{bmatrix}A & \mathbf{b}\\ \mathbf{0}^T & d\end{bmatrix}\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}$$

To compute this left inverse, we first compute the (right) Hermite Normal Form (HNF) of T,

$$\begin{bmatrix}U_1\\U_2\end{bmatrix} T = \begin{bmatrix}H\\0\end{bmatrix}.$$

The left inverse is then simply

$$\begin{bmatrix}dH^{-1}U_1 & -dH^{-1}\mathbf{v}\\ \mathbf{0}^T & d\end{bmatrix}.$$

We often also want a description of the affine subspace that is the range of the affine embedding and this is given by

$$\begin{bmatrix}U_2 & -U_2\mathbf{v}\\ \mathbf{0}^T & 1\end{bmatrix}\begin{bmatrix}\mathbf{x}\\1\end{bmatrix} = \mathbf{0}.$$

This computation is implemented in `left_inverse`.

5.7 Integral basis of the orthogonal complement of a linear subspace

Let $M_1 \in \mathbb{Z}^{m \times n}$ be a basis of a linear subspace. We first extend M_1 with zero rows to obtain a square matrix M' and then compute the (left) HNF of M',

$$\begin{bmatrix} M_1 \\ 0 \end{bmatrix} = \begin{bmatrix} H & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}.$$

The rows of Q_2 span the orthogonal complement of the given subspace. Since Q_2 can be extended to a unimodular matrix, these rows form an integral basis.

If the entries on the diagonal of H are all 1 then M_1 can be extended to a unimodular matrix, by concatenating M_1 and Q_2. The resulting matrix is unimodular, since

$$\begin{bmatrix} M_1 \\ Q_2 \end{bmatrix} = \begin{bmatrix} H & 0 \\ 0 & I_{n-m,n-m} \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}.$$

This method for extending a matrix of which only a few lines are known to a unimodular matrix is more general than the method described by Bik (1996), which only considers extending a matrix given by a single row.

5.8 Ensuring a polyhedron has only revlex-positive rays

The `barvinok_series_with_options` function and all further `gen_fun` manipulations assume that the effective parameter domain has only revlex-positive rays. When used to compute rational generating functions, the `barvinok_enumerate` application will therefore transform the effective parameter domain of a problem if it has revlex-negative rays. It will then not compute the generating function

$$f(\mathbf{x}) = \sum_{\mathbf{p} \in \mathbb{Z}^m} \#(P_{\mathbf{p}} \cap \mathbb{Z}^d)\, x^{\mathbf{p}},$$

but

$$g(\mathbf{z}) = \sum_{\mathbf{p}' \in \mathbb{Z}^n} \#(P_{T\mathbf{p}'+\mathbf{t}} \cap \mathbb{Z}^d)\, x^{\mathbf{p}'}$$

instead, where $\mathbf{p} = T\mathbf{p}' + \mathbf{t}$, with $T \in \mathbb{Z}^{m \times n}$ and $\mathbf{t} \in \mathbb{Z}^m$, is an affine transformation that maps the transformed parameter space back to the original parameter space.

First assume that the parameter domain does not contain any lines and that there are no equalities in the description of $P_{\mathbf{p}}$ that force the values of $\mathbf{p}$ for which $P_{\mathbf{p}}$ contains integer points to lie on a non-standard lattice. Let the effective parameter domain be given as $\{\, \mathbf{p} \mid A\mathbf{p} + \mathbf{c} \geq \mathbf{0} \,\}$, where $A \in \mathbb{Z}^{s \times d}$ of row rank d; otherwise the effective parameter domain would contain a line. Let H be the (left) HNF of A, i.e.,

$$A = HQ,$$

with H lower-triangular with positive diagonal elements and Q unimodular. Let $\tilde{Q}$ be the matrix obtained from Q by reversing its rows, and, similarly, $\tilde{H}$ from H by reversing the columns. After performing the transformation $\mathbf{p}' = \tilde{Q}\mathbf{p}$, i.e., $\mathbf{p} = \tilde{Q}^{-1}\mathbf{p}'$, the transformed parameter domain is given by

$$\{\, \mathbf{p}' \mid A\tilde{Q}^{-1}\mathbf{p}' + \mathbf{c} \geq \mathbf{0} \,\}$$

or

$$\{\, \mathbf{p}' \mid \tilde{H}\mathbf{p}' + \mathbf{c} \geq \mathbf{0} \,\}.$$

The first constraint of this domain is $h_{11} p'_m + c_1 \geq 0$. A ray with non-zero final coordinate therefore has a positive final coordinate. Similarly, the second constraint is $h_{22} p'_{m-1} + h_{21} p'_m + c_2 \geq 0$. A ray with zero nth coordinate, but non-zero $n-1$st coordinate, will therefore have a positive $n-1$st coordinate. Continuing this reasoning, we see that all rays in the transformed domain are revlex-positive.

If the parameter domain does contains lines, but is not restricted to a non-standard lattice, then the number of points in the parametric polytope is invariant over a translation along the lines. It is therefore sufficient to compute the number of points in the orthogonal complement of the linear subspace spanned by the lines. That is, we apply a prior transformation that maps a reduced parameter domain to this subspace,

$$\mathbf{p} = L^{\perp}\mathbf{p}' = \begin{bmatrix} L & L^{\perp} \end{bmatrix} \begin{bmatrix} 0 \\ I \end{bmatrix} \mathbf{p}',$$

where L has the lines as columns, and $L^{\perp}$ an integral basis for the orthogonal complement (Section 5.7). Note that the inverse transformation

$$\mathbf{p}' = L^{-\perp}\mathbf{p} = \begin{bmatrix} 0 & I \end{bmatrix} \begin{bmatrix} L & L^{\perp} \end{bmatrix}^{-1} \mathbf{p}$$

has integral coefficients since $L^{\perp}$ can be extended to a unimodular matrix.

If the parameter values $\mathbf{p}$ for which $P_{\mathbf{p}}$ contains integer points are restricted to a non-standard lattice, we first replace the parameters by a different set of parameters that lie on the standard lattice through "parameter compression"(Meister 2004),

$$\mathbf{p} = C\mathbf{p}'.$$

The (left) inverse of C can be computed as explained in Section 5.6, giving

$$\mathbf{p}' = C^{-L}\mathbf{p}.$$

We have to be careful to only apply this transformation when both the equalities computed in Section 5.6 are satisfied and some additional divisibility constraints. In particular if $\mathbf{a}^{T}/d$ is a row of C^{-L}, with $\mathbf{a} \in \mathbb{Z}^{n'}$ and $d \in \mathbb{Z}$, the transformation can only be applied to parameter values $\mathbf{p}$ such that d divides $\langle \mathbf{a}, \mathbf{p} \rangle$.

The complete transformation is given by

$$\mathbf{p} = CL^{\perp}\hat{Q}^{-1}\mathbf{p}'$$

or

$$\mathbf{p}' = \hat{Q}L^{-\perp}C^{-L}\mathbf{p}.$$

5.9 Parametric Volume Computation

The volume of a (parametric) polytope can serve as an approximation for the number of integer points in the polytope. We basically follow the description of Rabl (2006) here, except that we focus on volume computation for *linearly* parametrized polytopes, which we exploit to determine the sign of the determinants we compute, as explained below.

Note first that the vertices of a linearly parametrized polytope are affine expressions in the parameters that may be valid only in parts (chambers) of the parameter domain. Since the volume computation is based on the (active) vertices, we perform the computation in each chamber separately. Also note that since the vertices are affine expressions, it is easy to check whether they belong to a facet.

The volume of a d-simplex, i.e., a d-dimensional polytope with $d+1$ vertices, is relatively easy to compute. In particular, if $\mathbf{v}_i(\mathbf{p})$, for $0 \le i \le d$, are the (parametric) vertices of the simplex P then

$$\operatorname{vol} P = \frac{1}{d!} \left| \det \begin{bmatrix} v_{11}(\mathbf{p}) - v_{01}(\mathbf{p}) & v_{12}(\mathbf{p}) - v_{02}(\mathbf{p}) & \dots & v_{1d}(\mathbf{p}) - v_{0d}(\mathbf{p}) \\ v_{21}(\mathbf{p}) - v_{01}(\mathbf{p}) & v_{22}(\mathbf{p}) - v_{02}(\mathbf{p}) & \dots & v_{2d}(\mathbf{p}) - v_{0d}(\mathbf{p}) \\ \vdots & \vdots & \ddots & \vdots \\ v_{d1}(\mathbf{p}) - v_{01}(\mathbf{p}) & v_{d2}(\mathbf{p}) - v_{02}(\mathbf{p}) & \dots & v_{dd}(\mathbf{p}) - v_{0d}(\mathbf{p}) \end{bmatrix} \right| . \tag{5.25}$$

If P is not a simplex, i.e., $N > d+1$, with N the number of vertices of P, then the standard way of computing the volume of P is to first *triangulate* P, i.e., subdivide P into simplices, and then to compute and sum the volumes of the resulting simplices. One way of computing a triangulation is to compute the barycenter

$$\frac{1}{N} \sum_i \mathbf{v}_i(\mathbf{p})$$

of P and to perform a subdivision by computing the convex hulls of the barycenter with each of the facets of P. If a given facet of P is itself a simplex, then this convex hull is also a simplex. Otherwise the facet is further subdivided. This recursive process terminates as every 1-dimensional polytope is a simplex.

The triangulation described above is known as the boundary triangulation (Büeler et al. 2000) and is used by Rabl (2006) in his implementation. The Cohen-Hickey triangulation (Cohen and Hickey 1979; Büeler et al. 2000) is a much more efficient variation and uses one of the vertices instead of the barycenter. The facets incident on the vertex do not have to be considered in this case because the resulting subpolytopes would have zero volume. Another possibility is to use a "lifting" triangulation (Lee 1991; De Loera 1995). In this triangulation, each vertex is assigned a (random) "height" in an extra dimension. The projection of the "lower envelope" of the resulting polytope onto the original space results in a subdivision, which is a triangulation with very high probability.

A complication with the lifting triangulation is that the constraint system of the lifted polytope will in general not be linearly parameterized, even if the original polytope is. It is, however, sufficient to perform the triangulation for a particular value of the parameters inside the chamber since the parametric polytope has the same combinatorial structure throughout the chamber. The triangulation obtained for the instantiated vertices can then be carried over to the corresponding parametric vertices. We only need to be careful to select a value for the parameters that does not lie on any facet of the chambers. On these chambers, some of the vertices may coincide. For linearly parametrized polytopes, it is easy to find a parameter point in the interior of a chamber, as explained in Section 5.1. Note that this point need not be integer.

A direct application of the above algorithm, using any of the triangulations, would yield for each chamber a volume expressed as the sum of the absolute values of polynomials in the parameters. To remove the absolute value, we plug in a particular value of the parameters (not necessarily integer) belonging to the given chamber for which we know that the volume is non-zero. Again, it is sufficient to take any point in the interior of the chamber. The sign of the resulting value then determines the sign of the whole polynomial since polynomials are continuous functions and will not change sign without passing through zero.

5.10 Maclaurin series division

If $P(t)$ and $Q(t)$ are two Maclaurin series

$$\begin{aligned} P(t) &= a_0 + a_1 t + a_2 t^2 + \cdots \\ Q(t) &= b_0 + b_1 t + b_2 t^2 + \cdots , \end{aligned}$$

then, as outlined by Henrici (1974, 241–247), we can compute the coefficients c_l in

$$\frac{P(t)}{Q(t)} =: c_0 + c_1 t + c_2 t^2 + \cdots$$

by applying the recurrence relation

$$c_l = \frac{1}{b_0} \left(a_l - \sum_{i=1}^{l} b_i c_{l-i} \right).$$

To avoid dealing with denominators, we can also compute $d_l = b_0^{l+1} c_l$ instead as

$$d_l = b_0^l a_l - \sum_{i=1}^{l} b_0^{i-1} b_i c_{l-i}.$$

The coefficients c_l can then be directly read off as

$$c_l = \frac{d_l}{b_0^{l+1}}.$$

5.11 Specialization through exponential substitution

This section draws heavily from De Loera and Köppe (2006).

We define a “short” *rational generating function* to be a function of the form

$$f(\mathbf{x}) = \sum_{i \in I} \alpha_i \frac{\sum_{k=1}^{r} \mathbf{x}^{\mathbf{w}_{ik}}}{\prod_{j=1}^{k_i} (1 - \mathbf{x}^{\mathbf{b}_{ij}})}, \tag{5.26}$$

with $\mathbf{x} \in \mathbb{C}^d$, $\alpha_i \in \mathbb{Q}$, $\mathbf{w}_{ik} \in \mathbb{Z}^d$ and $\mathbf{b}_{ij} \in \mathbb{Z}^d \setminus \{\mathbf{0}\}$.

After computing the rational generating function (5.26) of a polytope (with $k_i = d$ for all i), the number of lattice points in the polytope can be obtained by evaluating $f(\mathbf{1})$. Since $\mathbf{1}$ is a pole of each term, we need to compute the constant term in the Laurent expansions of each term in (5.26) about $\mathbf{1}$. Since it is easier to work with univariate series, a substitution is usually applied, either a polynomial substitution

$$\mathbf{x} = (1 + t)^{\boldsymbol{\lambda}},$$

as implemented in `LattE` (De Loera et al. 2003), or an exponential substitution (see, e.g., Barvinok and Pommersheim 1999),

$$\mathbf{x} = e^{t\boldsymbol{\lambda}},$$

as implemented in `LattE macchiato` (Köppe 2006). In each case, $\boldsymbol{\lambda} \in \mathbb{Z}^d$ is a vector that is not orthogonal to any of the $\mathbf{b}_{ij}$. Both substitutions also transform the problem of computing the constant term in the Laurent expansions about $\mathbf{x} = \mathbf{1}$ to that of computing the constant term in the Laurent expansions about $t = 0$. Here, we discuss the exponential substitution.

Consider now one of the terms in (5.26),

$$g(t) = \frac{\sum_{k=1}^{r} e^{a_k t}}{\prod_{j=1}^{d} (1 - e^{c_j t})},$$

with $a_k = \langle \mathbf{w}_{ik}, \boldsymbol{\lambda} \rangle$ and $c_j = \langle \mathbf{b}_{ij}, \boldsymbol{\lambda} \rangle$. We rewrite this equation as

$$g(t) = (-1)^d \frac{\sum_{k=1}^{r} e^{a_k t}}{t^d \prod_{j=1}^{d} c_j} \prod_{j=1}^{d} \frac{-c_j t}{1 - e^{c_j t}}.$$

The second factor is analytic in a neighborhood of the origin $t = c_1 = \cdots = c_d = 0$ and therefore has a Taylor series expansion

$$\prod_{j=1}^{d} \frac{-c_j t}{1 - e^{c_j t}} = \sum_{m=0}^{\infty} \operatorname{td}_m(-c_1, \ldots, -c_d) t^m, \tag{5.27}$$

where td_m is a homogeneous polynomial of degree m called the m-th Todd polynomial (Barvinok and Pommersheim 1999). Also expanding the numerator in the first factor, we find

$$g(t) = \frac{(-1)^d}{t^d \prod_{j=1}^{d} c_j} \left(\sum_{n=0}^{\infty} \frac{\sum_{k=1}^{r} a_k^n}{n!} t^n \right) \left(\sum_{m=0}^{\infty} \operatorname{td}_m(-c_1, \ldots, -c_d) t^m \right),$$

with constant term

$$\frac{(-1)^d}{t^d \prod_{j=1}^d c_j} \left(\sum_{i=0}^d \frac{\sum_{k=1}^r a_k^i}{i!} \operatorname{td}_{d-i}(-c_1, \ldots, -c_d) \right) t^d = \frac{(-1)^d}{\prod_{j=1}^d c_j} \sum_{i=0}^d \frac{\sum_{k=1}^r a_k^i}{i!} \operatorname{td}_{d-i}(-c_1, \ldots, -c_d). \quad (5.28)$$

To compute the first $d+1$ terms in the Taylor series (5.27), we write down the truncated Taylor series

$$\frac{e^t - 1}{t} \equiv \sum_{i=0}^d \frac{1}{(i+1)!} t^i \equiv \frac{1}{(d+1)!} \sum_{i=0}^d \frac{(d+1)!}{(i+1)!} t^i \mod t^{d+1},$$

where we have

$$\frac{1}{(d+1)!} \sum_{i=0}^d \frac{(d+1)!}{(i+1)!} t^i \in \frac{1}{(d+1)!} \mathbb{Z}[t].$$

Computing the reciprocal as explained in Section 5.10, we find

$$\frac{-t}{1-e^t} = \frac{t}{e^t - 1} = \frac{1}{\frac{e^t - 1}{t}} \equiv (d+1)! \frac{1}{\sum_{i=0}^d \frac{(d+1)!}{(i+1)!} t^i} =: \sum_{i=0}^d b_i t^i. \quad (5.29)$$

Note that the constant term of the denominator is $1/(d+1)!$. The denominators of the quotient are therefore $((d+1)!)^{i+1}/(d+1)!$. Also note that the b_i are independent of the generating function and can be computed in advance. An alternative way of computing the b_i is to note that

$$\frac{t}{e^t - 1} = \sum_{i=0}^\infty B_i \frac{t^i}{i!},$$

with $B_i = i!\, b_i$ the Bernoulli numbers, which can be computed using the recurrence (5.34) (see Section 5.12).

Substituting t by $c_j t$ in (5.29), we have

$$\frac{-c_j t}{1 - e^{c_j t}} = \sum_{i=0}^d b_i c_j^i t^i.$$

Multiplication of these truncated Taylor series for each c_j results in the first $d+1$ terms of (5.27),

$$\sum_{m=0}^d \operatorname{td}_m(-c_1, \ldots, -c_d) t^m =: \sum_{m=0}^d \frac{\beta_m}{((d+1)!)^m} t^m,$$

from which it is easy to compute the constant term (5.28). Note that this convolution can also be computed without the use of rational coefficients,

$$\frac{(-1)^d}{\prod_{j=1}^d c_j} \sum_{i=0}^d \frac{\alpha_i}{i!} \frac{\beta_{d-i}}{((d+1)!)^{d-i}} = \frac{(-1)^d}{((d+1)!)^d \prod_{j=1}^d c_j} \sum_{i=0}^d \left(\frac{((d+1)!)^i}{i!} \alpha_i \right) \beta_{d-i},$$

with $\alpha_i = \sum_{k=1}^r a_k^i$.

Example 5.30 *Consider the rational generating function*

$$f(T;\mathbf{x}) = \frac{x_1^2}{(1-x_1^{-1})(1-x_1^{-1}x_2)} + \frac{x_2^2}{(1-x_2^{-1})(1-x_1x_2^{-1})} + \frac{1}{(1-x_1)(1-x_2)}$$

from Verdoolaege (2005, Example 39). Since this is a 2-dimensional problem, we first compute the first 3 Todd polynomials (evaluated at -1),

$$\frac{e^t-1}{t} \equiv 1 + \frac{1}{2}t + \frac{1}{6}t^2 = \frac{1}{6}\begin{bmatrix}6 & 3 & 1\end{bmatrix}$$

and

$$\frac{-t}{1-e^t} = \frac{t}{e^t-1} \equiv \begin{bmatrix}\frac{1}{1} & \frac{-3}{6} & \frac{3}{36}\end{bmatrix},$$

where we represent each truncated power series by a vector of its coefficients. The vector $\boldsymbol{\lambda} = (1,-1)$ is not orthogonal to any of the rays, so we can use the substitution $\mathbf{x} = e^{(1,-1)t}$ *and obtain*

$$\frac{e^{2t}}{(1-e^{-t})(1-e^{-2t})} + \frac{e^{-2t}}{(1-e^{t})(1-e^{2t})} + \frac{1}{(1-e^{t})(1-e^{-t})}.$$

We have

$$\begin{aligned}
\frac{t}{1-e^{-t}} &= \begin{bmatrix}\frac{1}{1} & \frac{3}{6} & \frac{3}{36}\end{bmatrix}\\
\frac{2t}{1-e^{-2t}} &= \begin{bmatrix}\frac{1}{1} & \frac{6}{6} & \frac{12}{36}\end{bmatrix}\\
\frac{-t}{1-e^{t}} &= \begin{bmatrix}\frac{1}{1} & \frac{-3}{6} & \frac{3}{36}\end{bmatrix}\\
\frac{-2t}{1-e^{2t}} &= \begin{bmatrix}\frac{1}{1} & \frac{-6}{6} & \frac{12}{36}\end{bmatrix}.
\end{aligned}$$

The first term in the rational generating function evaluates to

$$\begin{aligned}
&\frac{1}{-1\cdot -2}\begin{bmatrix}\frac{1}{1} & \frac{2}{1} & \frac{4}{2}\end{bmatrix} * \left(\begin{bmatrix}\frac{1}{1} & \frac{3}{6} & \frac{3}{36}\end{bmatrix}\begin{bmatrix}\frac{1}{1} & \frac{6}{6} & \frac{12}{36}\end{bmatrix}\right)\\
&= \frac{1}{2}\begin{bmatrix}\frac{1}{1} & \frac{2}{1} & \frac{4}{2}\end{bmatrix} * \begin{bmatrix}\frac{1}{1} & \frac{9}{6} & \frac{33}{36}\end{bmatrix}\\
&= \frac{1}{72}\begin{bmatrix}1 & 2\cdot 6 & 4\cdot 18\end{bmatrix} * \begin{bmatrix}1 & 9 & 33\end{bmatrix} = \frac{213}{72} = \frac{71}{24}.
\end{aligned}$$

Due to symmetry, the second term evaluates to the same value, while for the third term we find

$$\frac{1}{-1\cdot 1\cdot 36}\begin{bmatrix}1 & 0\cdot 6 & 0\cdot 18\end{bmatrix} * \begin{bmatrix}1 & 0 & -3\end{bmatrix} = \frac{-3}{-36} = \frac{1}{12}.$$

The sum is

$$\frac{71}{24} + \frac{71}{24} + \frac{1}{12} = 6.$$

Note that the run-time complexities of polynomial and exponential substitution are basically the same. The experiments of Köppe (2007) are somewhat misleading in this respect since the polynomial substitution (unlike the exponential substitution) had not been optimized to take full advantage of the stopped Barvinok decomposition. For comparison, Table 2 shows running times for the same experiments of that paper, but using barvinok version `barvinok-0.23-47-gaa9024e` on an Athlon MP 1500+ with 512MiB internal memory. This machine appears to be slightly slower than the machine used in the experiments of Köppe (2007) as computing `hickerson-14` using the dual decomposition with polynomial substitution and maximal index 1 took 2768 seconds on this machine using `LattE macchiato`. At this stage, it is not clear yet why the number of cones in the dual decomposition of `hickerson-13` differs from that of `LattE` (De Loera et al. 2003) and `LattE macchiato` (Köppe 2006). We conclude from Table 2 that (our implementation of) the exponential substitution is always slightly faster than (our implementation of) the polynomial substitution. The optimal maximal index for these examples is about 500, which agrees with the experiments of Köppe (2007).

5.12 Approximate Enumeration using Nested Sums

If $P \in \mathbb{Q}^d$ is a polyhedron and $p(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}]$ is a polynomial and we want to sum $p(\mathbf{x})$ over all integer values of (a subset of) the variables $\mathbf{x}$, then we can do this incrementally by taking a variable x_1 with lower bound $L(\hat{\mathbf{x}})$ and upper bound $U(\hat{\mathbf{x}})$, with $\hat{\mathbf{x}} = (x_2, \ldots, x_d)$, and computing

$$Q(\hat{\mathbf{x}}) = \sum_{x_1 = L(\hat{\mathbf{x}})}^{U(\hat{\mathbf{x}})} p(\mathbf{x}). \tag{5.31}$$

Since P is a polytope, the lower bound is a maximum of affine expressions in the remaining variables, while the upper bound is a minimum of such expressions. If the coefficients in these expressions are all integer, then we can compute $Q(\hat{\mathbf{x}})$ exactly as a piecewise polynomial using formulas for sums of powers, as proposed by, e.g., Tawbi (1994), Sakellariou (1997), Van Engelen et al. (2006). If some of the coefficients are not integer, we can apply the same formulas to obtain an approximation, which can is some cases be shown to be an overapproximation (Van Engelen et al. 2006). Note that if we take the initial polynomial to be the constant 1, then this gives us a method for computing an approximation of the number of integer points in a (parametric) polytope.

The first step is to compute the chamber decomposition of P when viewed as a 1-dimensional parametric polytope. That is, we need to partition the projection of P onto the remaining variables into polyhedral cells such that in each cell, both the upper and the lower bound are described by a single affine expression. Basically, for each pair of lower and upper bound, we compute the cell where the chosen lower bound is (strictly) smaller than all other lower bounds and similarly for the upper bound.

For any given pair of lower and upper bound $(l(\hat{\mathbf{x}}), u(\hat{\mathbf{x}}))$, the formula (5.31) is

	Dual decomposition			Primal decomposition		
		Time (s)			Time (s)	
Max. index	Cones	Poly	Exp	Cones	Poly	Exp
			`hickerson-12`			
1	11625	9.24	8.90	7929	4.80	4.55
10	4251	4.32	4.19	803	0.66	0.62
100	980	1.42	1.35	84	0.13	0.12
200	550	1.00	0.92	76	0.12	0.12
300	474	0.93	0.86	58	0.12	0.10
500	410	0.90	0.83	42	0.10	0.10
1000	130	0.42	0.38	22	**0.10**	**0.07**
2000	10	**0.10**	**0.10**	22	0.10	0.09
5000	7	0.12	0.11	7	0.12	0.10
			`hickerson-13`			
1	494836	489	463	483507	339	315
10	296151	325	309	55643	51	48
100	158929	203	192	9158	11	10
200	138296	184	173	6150	9	8
300	110438	168	157	4674	8	7
500	102403	163	151	3381	**8**	**7**
1000	83421	**163**	**149**	2490	8	7
2000	77055	170	153	1857	10	8
5000	57265	246	211	1488	13	11
10000	50963	319	269	1011	26	21
			`hickerson-14`			
1	1682743	2171	2064	552065	508	475
10	1027619	1453	1385	49632	62	59
100	455474	768	730	8470	14	13
200	406491	699	661	5554	11	10
300	328340	627	590	4332	11	9
500	303566	605	565	3464	**11**	**9**
1000	232626	**581**	**532**	2384	12	10
2000	195368	607	545	1792	14	12
5000	147496	785	682	1276	19	16
10000	128372	966	824	956	29	23

Table 2: Timing results of dual and primal decomposition with polynomial or exponential substitution on the Hickerson examples

computed for each monomial of $p(\mathbf{x})$ separately. For the constant term α_0, we have

$$\sum_{x_1=l(\hat{\mathbf{x}})}^{u(\hat{\mathbf{x}})} \alpha_0(\hat{\mathbf{x}}) = \alpha_0(\hat{\mathbf{x}})\,(u(\hat{\mathbf{x}}) - l(\hat{\mathbf{x}}) + 1)\,. \tag{5.32}$$

For the higher degree monomials, we use the formula

$$\sum_{k=0}^{m-1} k^n = \frac{1}{n+1}\sum_{k=0}^{n}\binom{n+1}{k} B_k m^{n+1-k} =: S_n(m), \tag{5.33}$$

with B_i the Bernoulli numbers, which can be computed using the recurrence

$$\sum_{j=0}^{m}\binom{m+1}{j} B_j = 0 \qquad B_0 = 1. \tag{5.34}$$

Note that (5.33) is also valid if $m = 0$, i.e., $S_n(0) = 0$, a fact that can be easily shown using Newton series (Van Engelen et al. 2006).

Since we can only directly apply the summation formula when the lower bound is zero (or one), we need to consider several cases.

1. $l(\hat{\mathbf{x}}) \geq 1$

$$\begin{aligned}\sum_{x_1=l(\hat{\mathbf{x}})}^{u(\hat{\mathbf{x}})} \alpha_n(\hat{\mathbf{x}})\, x_1^n &= \alpha_n(\hat{\mathbf{x}}) \left(\sum_{x_1=1}^{u(\hat{\mathbf{x}})} x_1^n - \sum_{x_1=1}^{l(\hat{\mathbf{x}})-1} x_1^n \right) \\ &= \alpha_n(\hat{\mathbf{x}})\,(S_n(u(\hat{\mathbf{x}}) + 1) - S_n(l(\hat{\mathbf{x}})))\end{aligned}$$

2. $u(\hat{\mathbf{x}}) \leq -1$

$$\begin{aligned}\sum_{x_1=l(\hat{\mathbf{x}})}^{u(\hat{\mathbf{x}})} \alpha_n(\hat{\mathbf{x}})\, x_1^n &= \alpha_n(\hat{\mathbf{x}})(-1)^n \sum_{x_1=-u(\hat{\mathbf{x}})}^{-l(\hat{\mathbf{x}})} \alpha_n(\hat{\mathbf{x}})\, x_1^n \\ &= \alpha_n(\hat{\mathbf{x}})(-1)^n\,(S_n(-l(\hat{\mathbf{x}}) + 1) - S_n(-u(\hat{\mathbf{x}})))\end{aligned}$$

3. $l(\hat{\mathbf{x}}) \leq 0$ and $u(\hat{\mathbf{x}}) \geq 0$

$$\begin{aligned}\sum_{x_1=l(\hat{\mathbf{x}})}^{u(\hat{\mathbf{x}})} \alpha_n(\hat{\mathbf{x}})\, x_1^n &= \alpha_n(\hat{\mathbf{x}}) \left(\sum_{x_1=0}^{u(\hat{\mathbf{x}})} x_1^n + (-1)^n \sum_{x_1=1}^{-l(\hat{\mathbf{x}})} x_1^n \right) \\ &= \alpha_n(\hat{\mathbf{x}})\,(S_n(u(\hat{\mathbf{x}}) + 1) + (-1)^n S_n(-l(\hat{\mathbf{x}}) + 1))\end{aligned}$$

If the coefficients in the lower and upper bound are all integer, then the above 3 cases partition (the integer points in) the projection of P onto the remaining variables. However, if some of the coefficients are rational, then the lower and upper bound can lie in the open interval $(0, 1)$ for some values of $\hat{\mathbf{x}}$. We may therefore also want to consider the following two cases.

4. $0 < l(\hat{\mathbf{x}}) < 1$

$$\sum_{x_1=l(\hat{\mathbf{x}})}^{u(\hat{\mathbf{x}})} \alpha_n(\hat{\mathbf{x}})\, x_1^n = \alpha_n(\hat{\mathbf{x}}) S_n(u(\hat{\mathbf{x}}) + 1)$$

5. $0 < -u(\hat{\mathbf{x}}) < 1$

$$\sum_{x_1=l(\hat{\mathbf{x}})}^{u(\hat{\mathbf{x}})} \alpha_n(\hat{\mathbf{x}})\, x_1^n = \alpha_n(\hat{\mathbf{x}})(-1)^n S_n(-l(\hat{\mathbf{x}}) + 1)$$

Note that we may add the constraint $u \geq 1$ to case 4 and the constraint $l \leq -1$ to case 5, since the correct value for these two cases would be zero if these extra constraints do not hold.

An alternative to adding the above two cases would be to simply ignore them, i.e., assume a value of 0. Another alternative would be to reduce case 3 to

$$l(\hat{\mathbf{x}}) \leq -1 \quad \text{and} \quad u(\hat{\mathbf{x}}) \geq 1,$$

while extending cases 4 and 5 to

$$-1 < l(\hat{\mathbf{x}}) < 1 \quad \text{and} \quad u \geq 1$$

and

$$-1 < u(\hat{\mathbf{x}}) < 1 \quad \text{and} \quad l \leq -1,$$

respectively, with the remaining cases ($-1 < l \leq u < 1$) having value 0. There does not appear to be a consistently better choice here, as each of these three approaches seems to yield better results on some examples. The last approach has the additional drawback that we would also have to deal with 5 cases, even if the bounds are integers.

If at least one of the lower or upper bound is an integer affine expression, then we can reduce the 3 (or 5) cases to a single case (case 3) by an affine substitution that ensure that the new (lower or upper) bound is zero. In particular, if $l(\hat{\mathbf{x}})$ is an integer affine expression, then we replace x by $x' + l(\hat{\mathbf{x}})$ and similarly for an upper bound.

5.13 Exact Enumeration using Nested Sums

The exact enumeration using nested sums proceeds in much the same way as the approximate enumeration from subsection 5.12, with the notable exception that we need to take the (greatest or least) integer part of any fractional bounds that may occur. This has several consequences, discussed below.

Since we will introduce floors during the recursive application of the procedure, we may as well allow the weight $p(\mathbf{x})$ in (5.31) to be a (piecewise) quasipolynomial.

For the constant term, (5.32) becomes

$$\sum_{x_1=l(\hat{\mathbf{x}})}^{u(\hat{\mathbf{x}})} \alpha_0(\hat{\mathbf{x}}) = \alpha_0(\hat{\mathbf{x}}) \left(\lfloor u(\hat{\mathbf{x}}) \rfloor - \lceil l(\hat{\mathbf{x}}) \rceil + 1\right).$$

Since we force the lower and upper bounds to be integers, cases 4 and 5 do not occur, while the conditions for cases 1 and 2 can be simplified to

$$l(\hat{\mathbf{x}}) > 0$$

and

$$u(\hat{\mathbf{x}}) < 0,$$

respectively.

If the variable x appears in any floor expression, either because such an expression was present in the original weight function or because it was introduced when another variable with an affine bound in x was summed, then the domain has to be "splintered" into D parts, where D is the least common multiple of the denominators of the coefficients of x in any of the integer parts. In particular, the domain is split into $x = Dy + i$ for each i in $[0, D - 1]$. Since D is proportional to the coefficients in the constraints, it is exponential in the input size. This splintering will therefore introduce exponential behavior, even if the dimension is fixed.

Splintering is clearly the most expensive step in the algorithm, so we want to avoid this step as much as possible. Pugh (1994) already noted that summation should proceed over variables with integer bounds first. This can be extended to choosing a variable with the smallest coefficient in absolute value. In this way, we can avoid splintering on the largest denominator.

Sakellariou (1996) claims that splintering can be avoided altogether. In particular, Sakellariou (1996, Lemma 3.2) shows that

$$\sum_{x=0}^{a} x^m \, (x \bmod b)^n \, ,$$

with a and b integers, is equal to

$$\begin{cases} \displaystyle\sum_{x=0}^{a} x^{m+n} & \text{if } a < b \\ \displaystyle\sum_{i=0}^{\lfloor a/b \rfloor - 1} \sum_{x=0}^{b-1} (x + ib)^m x^n + \sum_{x=0}^{a \bmod b} (x + b \lfloor a/b \rfloor)^m x^n & \text{if } a \ge b, \end{cases} \tag{5.35}$$

effectively avoiding splintering if a given monomial contains a single integer part expression with argument of the form x/b. An argument of the form $(x - c(\hat{\mathbf{x}}))/b$ can be handled through a variable substitution. If the argument is of the form cx/b, with $c \neq 1$, then Sakellariou (1996, (3.27)) proposes to rewrite the monomial as

$$\begin{aligned} \sum_{x=0}^{a} (cx \bmod b)^n &= \sum_{x=0}^{a} \sum_{y=cx}^{cx} (y \bmod b)^n \\ &= \sum_{x=0}^{a} \left(\sum_{y=0}^{cx} (y \bmod b)^n - \sum_{y=0}^{cx-1} (y \bmod b)^n \right) \end{aligned}$$

and applying (5.35). However, such an application results in an expression containing

$$\sum_{y=0}^{cx \bmod b} y^n,$$

which in turn leads to a polynomial of degree $n + 1$ in ($cx \bmod b$), i.e., of degree one higher than the original expression. Furthermore, if the bound on x is rational then a itself contains a floor, which, on application of (5.35), results in a nested floor expression, blocking the application of the same rule for the next variable. Finally, the case where a monomial contains multiple floor expressions, either occurring in the input quasi-polynomial or introduced by different variables having a rational bound with a non-zero coefficient in the same variable, is not handled. Also note that if we disallow nested floor expressions, then this rule will rarely be applicable since we try to eliminate variables with integer bounds first.

5.14 Summation using local Euler-Maclaurin formula

In this section we provide some implementation details on using local Euler-Maclaurin formula to compute the sum of a piecewise polynomial evaluated in all integer points of a two-dimensional parametric polytope. For the theory behind these formula and a discussion of the original implementation (for non-parametric simplices), we refer to Berline and Vergne (2006).

In particular, consider a parametric piecewise polynomial in n parameters and m variables $c : \mathbb{Z}^n \to \mathbb{Z}^m \to \mathbb{Q} : \mathbf{p} \mapsto c(\mathbf{p})$, with $c(\mathbf{p}) : \mathbb{Z}^m \to \mathbb{Q} : \mathbf{x} \mapsto c(\mathbf{p})(\mathbf{x})$ and

$$c_{\mathbf{p}}(\mathbf{x}) = \begin{cases} c_1(\mathbf{p})(\mathbf{x}) & \text{if } \mathbf{x} \in D_1(\mathbf{p}) \\ \vdots & \\ c_r(\mathbf{p})(\mathbf{x}) & \text{if } \mathbf{x} \in D_r(\mathbf{p}), \end{cases}$$

with the c_i polynomials, $c_i \in (\mathbb{Q}[\mathbf{p}])[\mathbf{x}]$, and the D_i disjoint linearly parametric polytopes. We want to compute

$$g(\mathbf{p}) = \sum_{\mathbf{x} \in \mathbb{Z}^m} c(\mathbf{p})(\mathbf{x}).$$

5.14.1 Reduction to the summation of a parametric polynomial over a parametric polytope with a fixed combinatorial structure

Since the D_i are disjoint, we can consider each (c_i, D_i)-pair individually and compute

$$g(\mathbf{p}) = \sum_{i=1}^{r} g_i(\mathbf{p}) = \sum_{i=1}^{r} \sum_{\mathbf{x} \in D_r(\mathbf{p}) \cap \mathbb{Z}^m} c_r(\mathbf{p})(\mathbf{x}).$$

The second step is to compute the chamber decomposition (Verdoolaege 2005, Section 4.2.3) of each parametric polytope D_i. The result is a subdivision of the parameter space into chambers C_{ij} such that D_i has a fixed combinatorial structure, in particular a fixed set of parametric vertices, on (the interior of) each C_{ij}. Applying Theorem 5.12, this subdivision can be transformed into a partition $\{\, \tilde{C}_{ij} \,\}$ by making some of the facets of the chambers open (Köppe and Verdoolaege 2008, Section 3.2). Since we are only interested in integer parameter values, any of the resulting open facets $\langle \mathbf{a}, \mathbf{p} \rangle + c > 0$, with $\mathbf{a} \in \mathbb{Z}^n$ and $c \in \mathbb{Z}$, can then be replaced by $\langle \mathbf{a}, \mathbf{p} \rangle + c - 1 \geq 0$. Again, we have

$$g_i(\mathbf{p}) = \sum_j g_{ij}(\mathbf{p}) = \sum_j \sum_{\mathbf{x} \in C_{ij}(\mathbf{p}) \cap \mathbb{Z}^m} c_r(\mathbf{p})(\mathbf{x}).$$

After this reduction, the technique of Berline and Vergne (2006) can be applied practically verbatim to the parametric polytope with a fixed combinatorial structure. In principle, we could also handle piecewise quasi-polynomials using the technique of Verdoolaege (2005, Section 4.5.4), except that we only need to create an extra variable for each distinct floor expression in a monomial, rather than for each occurrence of a floor expression in a monomial. However, since we currently only support two-dimensional polytopes, this reduction has not been implemented yet.

5.14.2 Summation over a one-dimensional parametric polytope

The basis for the summation technique is the local Euler-Maclaurin formula (Berline and Vergne 2006, Theorem 26)

$$\sum_{\mathbf{x} \in P(\mathbf{p}) \cap \Lambda} h(\mathbf{p})(\mathbf{x}) = \sum_{F(\mathbf{p}) \in \mathcal{F}(P(\mathbf{p}))} \int_{F(\mathbf{p})} D_{P(\mathbf{p}),F(\mathbf{p})} \cdot h(\mathbf{p}), \tag{5.36}$$

where $P(\mathbf{p})$ is a parametric polytope, Λ is a lattice, $\mathcal{F}(P(\mathbf{p}))$ are the faces of $P(\mathbf{p})$, $D_{P(\mathbf{p}),F(\mathbf{p})}$ is a specific differential operator associated to the face of a polytope. The Lebesgue measure used in the integral is such that the integral of the indicator function of a lattice element of the lattice $\Lambda \cap (\operatorname{aff}(F(\mathbf{p})) - F(\mathbf{p}))$ is 1, i.e., the intersection of Λ with the linear subspace parallel to the affine hull of the face $F(\mathbf{p})$. Note that the original theorem is formulated for a non-parametric polytope and a non-parametric polynomial. However, as we will see, in each of the steps in the computation, the parameters can be treated as symbolic constants without affecting the validity of the formula, see also Berline and Vergne (2006, Section 6).

The differential operator $D_{P(\mathbf{p}),F(\mathbf{p})}$ is obtained by plugging in the vector $\mathbf{D} = (D_1, \ldots, D_m)$ of first order differential operators, i.e., D_k is the first order differential operator in the kth variable, in the function $\mu_{P(\mathbf{p}),F(\mathbf{p})}$. This function is determined by the *transverse cone* of the polyhedron $P(\mathbf{p})$ along its face $F(\mathbf{p})$, which is the supporting cone of $P(\mathbf{p})$ along $F(\mathbf{p})$ projected into the linear subspace orthogonal to $F(\mathbf{p})$. The lattice associated to this space is the projection of Λ into this space.

In particular, for a zero-dimensional affine cone in the zero-dimensional space, we have $\mu = 1$ (Berline and Vergne 2006, Proposition 12), while for a one-dimensional affine cone $K = (-t + \mathbb{R}_+)r$ in the one-dimensional space, where r is a primitive integer vector and $t \in [0, 1)$, we have (Berline and Vergne 2006, (13))

$$\mu(K)(\xi) = \frac{e^{ty}}{1 - e^y} + \frac{1}{y} = -\sum_{n=0}^{\infty} \frac{b(n+1, t)}{(n+1)!} y^n, \tag{5.37}$$

with $y = \langle \xi, r\rangle$ and $b(n,t)$ the Bernoulli polynomials defined by the generating series

$$\mathrm{Todd}(t,y) = \frac{e^{ty}y}{e^y-1} = \sum_{n=0}^{\infty} \frac{b(n,t)}{n!}y^n. \tag{5.38}$$

The constant terms of these Bernoulli polynomials are the Bernoulli numbers.

Applying (5.36) to a one-dimensional parametric polytope $P(\mathbf{p}) = [v_1(\mathbf{p}), v_2(\mathbf{p})]$, we find

$$\begin{aligned}\sum_{x\in P(\mathbf{p})\cap\mathbb{Z}} h(\mathbf{p})(x) = \int_{P(\mathbf{p})} & D_{P(\mathbf{p}),P(\mathbf{p})}\cdot h(\mathbf{p}) \\ &+ \int_{v_1(\mathbf{p})} D_{P(\mathbf{p}),v_1(\mathbf{p})}\cdot h(\mathbf{p}) \\ &+ \int_{v_2(\mathbf{p})} D_{P(\mathbf{p}),v_2(\mathbf{p})}\cdot h(\mathbf{p}).\end{aligned}$$

The transverse cone of a polytope along the whole polytope is a zero-dimensional cone in a zero-dimensional space and so $D_{P(\mathbf{p}),P(\mathbf{p})} = \mu_{P(\mathbf{p}),P(\mathbf{p})}(D) = 1$. The transverse cone along $v_1(\mathbf{p})$ is $v_1(\mathbf{p}) + \mathbb{R}_+$ and so $D_{P(\mathbf{p}),v_1(\mathbf{p})} = \mu(v_1(\mathbf{p}) + \mathbb{R}_+)(D)$ as in (5.37), with $y = \langle D, 1\rangle = D$ and $t = \lceil v_1(\mathbf{p})\rceil - v_1(\mathbf{p}) = \{-v_1(\mathbf{p})\}$. Similarly we find $D_{P(\mathbf{p}),v_2(\mathbf{p})} = \mu(v_2(\mathbf{p}) - \mathbb{R}_+)(D)$ as in (5.37), with $y = \langle D, -1\rangle = -D$ and $t = v_2(\mathbf{p}) - \lfloor v_2(\mathbf{p})\rfloor = \{v_2(\mathbf{p})\}$. Summarizing, we find

$$\begin{aligned}\sum_{x\in P(\mathbf{p})\cap\mathbb{Z}} h(\mathbf{p})(x) = \int_{v_1(\mathbf{p})}^{v_2(\mathbf{p})} & h(\mathbf{p})(t)\,dt \\ &- \sum_{n=0}^{\infty} \frac{b(n+1,\{-v_1(\mathbf{p})\})}{(n+1)!}(D^n h(\mathbf{p}))(v_1(\mathbf{p})) \\ &- \sum_{n=0}^{\infty} (-1)^n \frac{b(n+1,\{v_2(\mathbf{p})\})}{(n+1)!}(D^n h(\mathbf{p}))(v_2(\mathbf{p})).\end{aligned}$$

Note that in order to apply this formula, we need to verify first that $v_1(\mathbf{p})$ is indeed smaller than (or equal to) $v_2(\mathbf{p})$. Since the combinatorial structure of $P(\mathbf{p})$ does not change throughout the interior of the chamber, we only need to check the order of the two vertices for one value of the parameters from the interior of the chamber, a point which we may compute as in subsection 5.1.

5.14.3 Summation over a two-dimensional parametric polytope

For two-dimensional polytope, formula (5.36) has three kinds of contributions: the integral of the polynomial over the polytope, contributions along edges and contributions along vertices. As suggested by Berline (2007), the integral can be computed by applying the Green-Stokes theorem:

$$\iint_{P(\mathbf{p})} \left(\frac{\partial M}{\partial x} - \frac{\partial L}{\partial y}\right) = \int_{\partial P(\mathbf{p})} (L\,dx + M\,dy).$$

In particular, if $M(\mathbf{p})(x, y)$ is such that $\frac{\partial M}{\partial x}(\mathbf{p})(x, y) = h(\mathbf{p})(x, y)$ then

$$\iint_{P(\mathbf{p})} h(\mathbf{p})(x, y) = \int_{\partial P(\mathbf{p})} M(\mathbf{p})(x, y)\, dy.$$

Care must be taken to integrate over the boundary in the positive direction. Assuming the vertices of the polygon are not given in a predetermined order, we can check the correct orientation of the vertices of each edge individually. Let $\mathbf{n} = (n_1, n_2)$ be the inner normal of a facet and let $\mathbf{v}_1(\mathbf{p})$ and $\mathbf{v}_2(\mathbf{p})$ be the two vertices of the facet, then the vertices are in the correct order if

$$\begin{vmatrix} v_{2,1}(\mathbf{p}) - v_{1,1}(\mathbf{p}) & n_1 \\ v_{2,2}(\mathbf{p}) - v_{1,2}(\mathbf{p}) & n_2 \end{vmatrix} \geq 0.$$

Since these two vertices belong to the same edge, their order will not change within a chamber and so we can again perform this check for a single value of the parameters. To integrate M over an edge F, let $\mathbf{f}$ be a primitive integer vector in the direction of the edge. Then $\mathbf{v}_2(\mathbf{p}) = \mathbf{v}_1(\mathbf{p}) + k(\mathbf{p})\,\mathbf{f}$ and any point on the edge can be written as $\mathbf{v}_1(\mathbf{p}) + \lambda \mathbf{f}$ with $0 \leq \lambda \leq k(\mathbf{p})$. That is,

$$\int_F M(\mathbf{p})(x, y)\, dy = \int_0^{k(\mathbf{p})} M(\mathbf{p})(v_{1,1}(\mathbf{p}) + \lambda f_1, v_{1,2}(\mathbf{p}) + \lambda f_2) f_2\, d\lambda. \tag{5.39}$$

For the edges, we can again apply (5.37), but we must first project the supporting cone at the edge into the linear subspace orthogonal to the edge. Let $\mathbf{n} = (n_1, n_2)$ be the (primitive integer) inner normal of this facet $F(\mathbf{p})$, then $\mathbf{f} = (-n_2, n_1)$ is parallel to the facet and we can write one of the vertices $\mathbf{v}(\mathbf{p})$ as a linear combination of these two vectors:

$$\mathbf{v}(\mathbf{p}) = \begin{bmatrix} \mathbf{f} & \mathbf{n} \end{bmatrix} \mathbf{a}(\mathbf{p}) = \begin{bmatrix} -n_2 & n_1 \\ n_1 & n_2 \end{bmatrix} \mathbf{a}(\mathbf{p}) \tag{5.40}$$

or

$$\mathbf{a}(\mathbf{p}) = \begin{bmatrix} -n_2 & n_1 \\ n_1 & n_2 \end{bmatrix}^{-1} \mathbf{v}(\mathbf{p}) = \begin{bmatrix} -n_2/d & n_1/d \\ n_1/d & n_2/d \end{bmatrix} \mathbf{v}(\mathbf{p}), \tag{5.41}$$

with $d = n_1^2 + n_2^2$. The lattice associated to the linear subspace orthogonal to the facet is the projection of Λ into this space. Since $\mathbf{n}$ is primitive, a basis for this lattice can be identified with $\mathbf{n}/d$. The coordinate of the whole facet in this space is therefore $da_2(\mathbf{p}) = \begin{bmatrix} n_1 & n_2 \end{bmatrix} \mathbf{v}(\mathbf{p})$, while the transverse cone is $da_2(\mathbf{p}) + \mathbb{R}_+$. Similarly, a linear functional $\boldsymbol{\xi}'$ projects onto a linear functional $\xi = \langle \boldsymbol{\xi}', \mathbf{n} \rangle / d$ in the linear subspace. Applying (5.37), with $y = \frac{n_1}{d} D_1 + \frac{n_2}{d} D_2$ and $t = \{-n_1 v_1(\mathbf{p}) - n_2 v_2(\mathbf{p})\}$, we therefore find

$$\begin{aligned} D_{P(\mathbf{p}),F(\mathbf{p})} &= -\sum_{n=0}^{\infty} \frac{b(n+1, \{-n_1 v_1(\mathbf{p}) - n_2 v_2(\mathbf{p})\})}{(n+1)!} \left(\frac{n_1}{d} D_1 + \frac{n_2}{d} D_2 \right)^n \\ &= -\sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \frac{b(i+j+1, \{-n_1 v_1(\mathbf{p}) - n_2 v_2(\mathbf{p})\})}{(i+j+1)!} \frac{n_1^i n_2^j}{d^{i+j}} D_1^i D_2^j. \end{aligned}$$

After applying this differential operator to the polynomial $h(\mathbf{p})(\mathbf{x})$, the resulting polynomial $h'(\mathbf{p})(\mathbf{x}) = D_{P(\mathbf{p}),F(\mathbf{p})} \cdot h(\mathbf{p})(\mathbf{x})$ needs to be integrated over the facet. The measure to be used is such that the integral of a lattice tile in the linear space parallel to the facet is 1, i.e.,

$$\int_{\mathbf{0}}^{\mathbf{f}} 1 = \int_0^1 1 dz = 1,$$

with z the coordinate along $\mathbf{f}$. Referring to (5.40) and (5.41), all points of the facet have the form $\mathbf{x}(\mathbf{p}) = z\,\mathbf{f} + a_2(\mathbf{p})\,\mathbf{n}$, while the z-coordinate of the vertices $\mathbf{v}_1(\mathbf{p})$ and $\mathbf{v}_2(\mathbf{p})$ are $(-n_2 v_{1,1} + n_1 v_{1,2})/d$ and $(-n_2 v_{2,1} + n_1 v_{2,2})/d$, respectively. That is, the contribution of the facet is equal to

$$\int_{(-n_2 v_{1,1}+n_1 v_{1,2})/d}^{(-n_2 v_{2,1}+n_1 v_{2,2})/d} h'(\mathbf{p})\,(z\,\mathbf{f} + a_2(\mathbf{p})\,\mathbf{n})\ dz,$$

where, again, we need to ensure that the lower limit is smaller than the upper limit using the usual method of plugging in a particular value of the parameters.

Finally, we consider the contributions of the vertices. The transverse cones are in this case simply the supporting cones. Since μ is a valuation, we may apply Barvinok's decomposition and assume that the cone is unimodular. For an affine cone

$$\begin{aligned} K &= \mathbf{v}(\mathbf{p}) + \mathbb{R}_+\mathbf{r}_1 + \mathbb{R}_+\mathbf{r}_2 \\ &= (a_1(\mathbf{p}) + \mathbb{R}_+)\mathbf{r}_1 + (a_2(\mathbf{p}) + \mathbb{R}_+)\mathbf{r}_2, \end{aligned}$$

with

$$\mathbf{a}(\mathbf{p}) = \begin{bmatrix}\mathbf{r}_1 & \mathbf{r}_2\end{bmatrix}^{-1} \mathbf{v}(\mathbf{p}),$$

we have (Berline and Vergne 2006, Proposition 31),

$$\mu(K)(\boldsymbol{\xi}) = \frac{e^{t_1 y_1 + t_2 y_2}}{(1-e^{y_1})(1-e^{y_2})} + \frac{1}{y_1}B(y_2 - C_1 y_1, t_2) + \frac{1}{y_2}B(y_1 - C_2 y_2, t_1) - \frac{1}{y_1 y_2}, \tag{5.42}$$

with

$$B(y,t) = \frac{e^{ty}}{1-e^y} + \frac{1}{y} = -\sum_{n=0}^{\infty} \frac{b(n+1,t)}{(n+1)!} y^n,$$

$y_i = \langle \boldsymbol{\xi}, \mathbf{r}_i \rangle$, $C_i = \langle \mathbf{v}_1, \mathbf{v}_2 \rangle / \langle \mathbf{v}_i, \mathbf{v}_i \rangle$ and $t_i = \{-a_i(\mathbf{p})\}$. Expanding (5.42), we find

$$\begin{aligned} \mu(K)(\boldsymbol{\xi}) &= \left(-\frac{b(0,t1)}{y_1} - \sum_{n=0}^{\infty} \frac{b(n+1,t_1)}{(n+1)!} y_1^n\right)\left(-\frac{b(0,t2)}{y_2} - \sum_{n=0}^{\infty} \frac{b(n+1,t_2)}{(n+1)!} y_2^n\right) \\ &\quad - \left(\sum_{n=0}^{\infty} \frac{b(n+1,t_2)}{(n+1)!} \frac{y_2^n}{y_1} + \sum_{n=0}^{\infty} \frac{b(n+1,t_2)}{(n+1)!} \frac{(y_2 - C_1 y_1)^n - y_2^n}{y_1}\right) \\ &\quad - \left(\sum_{n=0}^{\infty} \frac{b(n+1,t_1)}{(n+1)!} \frac{y_1^n}{y_2} + \sum_{n=0}^{\infty} \frac{b(n+1,t_1)}{(n+1)!} \frac{(y_1 - C_2 y_2)^n - y_1^n}{y_2}\right) \\ &\quad - \frac{1}{y_1 y_2} \\ &= \sum_{n_1=0}^{\infty} \sum_{n_2=0}^{\infty} c(C_1, C_2, t_1, t_2; n_1, n_2)\, y_1^n y_2^n, \end{aligned}$$

with

$$\begin{aligned} c(C_1, C_2, t_1, t_2; n_1, n_2) = & \frac{b(n_1 + 1, t_1)}{(n_1 + 1)!} \frac{b(n_2 + 1, t_2)}{(n_2 + 1)!} \\ & - \frac{b(n_1 + n_2 + 1, t_2)}{(n_1 + n_2 + 1)!} \binom{n_1 + n_2 + 1}{n_1 + 1} (-C_1)^{n_1+1} \\ & - \frac{b(n_1 + n_2 + 1, t_1)}{(n_1 + n_2 + 1)!} \binom{n_1 + n_2 + 1}{n_2 + 1} (-C_2)^{n_2+1} . \end{aligned}$$

For $\boldsymbol{\xi} = \mathbf{D}$, we have

$$\begin{aligned} y_1^n y_2^n &= (r_{1,1} D_1 + r_{1,2} D_2)^{n_1} \, (r_{2,1} D_1 + r_{2,2} D_2)^{n_2} \\ &= \left(\sum_{k=0}^{n_1} r_{1,1}^k r_{1,2}^{n_1-k} \binom{n_1}{k} D_1^k D_2^{n_1-k} \right) \left(\sum_{l=0}^{n_2} r_{2,1}^l r_{2,2}^{n_2-l} \binom{n_2}{l} D_1^l D_2^{n_2-l} \right) \end{aligned}$$

and so $D_{P(\mathbf{p}),\mathbf{v}(\mathbf{p})} = \mu(K)(\mathbf{D}) =$

$$\sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \sum_{\substack{i+j=n_1+n_2 \\ n_1 \geq 0 \\ n_2 \geq 0}} \sum_{\substack{k+l=i \\ 0 \leq k \leq n_1 \\ 0 \leq l \leq n_2}} c(C_1, C_2, t_1, t_2; n_1, n_2) r_{1,1}^k r_{1,2}^{n_1-k} r_{2,1}^l r_{2,2}^{n_2-l} \binom{n_1}{k} \binom{n_2}{l} D_1^i D_2^j .$$

The contribution of this vertex is then

$$h'(\mathbf{p})(\mathbf{v}(\mathbf{p})),$$

with $h'(\mathbf{p})(\mathbf{x}) = D_{P(\mathbf{p}),\mathbf{v}(\mathbf{p})} \cdot h(\mathbf{p})(\mathbf{x})$.

Example 5.43 *As a simple example, consider the (non-parametric) triangle in Figure 5.44 and assume we want to compute*

$$\sum_{\mathbf{x} \in T \cap \mathbb{Z}^2} x_1 x_2 .$$

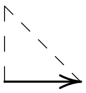
Since $T \cap \mathbb{Z}^2 = \{ (2,4), (3,4), (2,5) \}$*, the result should be*

$$2 \cdot 4 + 3 \cdot 4 + 2 \cdot 5 = 30.$$

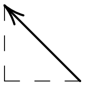
Let us first consider the integral

$$\iint_T x_1 x_2 = \int_{\partial T} \frac{x_1^2 x_2}{2} \, dx_2 .$$

Integration along each of the edges of the triangle yields the following.



For the edge in the margin, we have $\mathbf{f} = (1,0)$*, i.e.,* $f_2 = 0$*. The contribution of this edge to the integral is therefore zero.*



For this edge, we have $\mathbf{f} = (-1,1)$*. The contribution of this edge to the integral is therefore*

$$\int_0^1 \frac{(3-\lambda)^2 (4+\lambda)}{2} d\lambda = \frac{337}{24} .$$

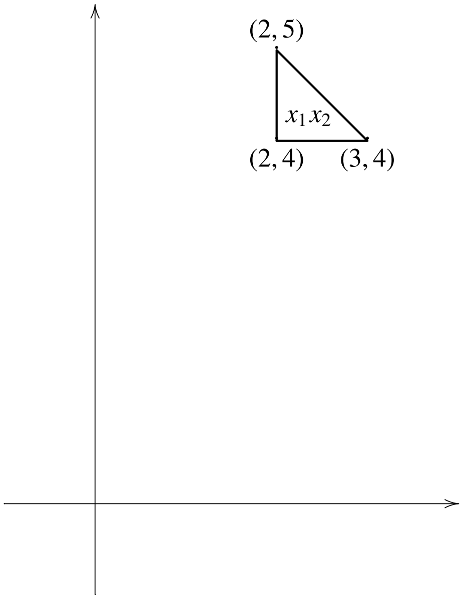
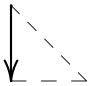


Figure 5.44: Sum of polynomial x_1x_2 over the integer points in a triangle T



For this edge, we have $\mathbf{f} = (0,-1)$*. The contribution of this edge to the integral is therefore*

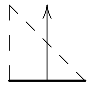
$$\int_0^1 \frac{2^2(5-\lambda)}{2}(-1)d\lambda = -9.$$

The total integral is therefore

$$\int_{\partial T} \frac{x_1^2x_2}{2}\,dx_2 = 0 + \frac{337}{24} - 9 = \frac{121}{24}.$$

Now let us consider the contributions of the edges. We will need the following Bernoulli numbers in our computations.

$$b(1,0) = -\frac{1}{2}$$
$$b(2,0) = \frac{1}{6}$$
$$b(3,0) = 0$$
$$b(4,0) = -\frac{1}{30}$$



The normal to the facet F_1 *in the margin is* $\mathbf{n} = (0,1)$*. The vector* $\mathbf{f} = (-1,0)$ *is parallel to the facet. We have*

$$\begin{bmatrix}2\\4\end{bmatrix} = -2\begin{bmatrix}-1\\0\end{bmatrix} + 4\begin{bmatrix}0\\1\end{bmatrix} \quad \textit{and} \quad \begin{bmatrix}3\\4\end{bmatrix} = -3\begin{bmatrix}-1\\0\end{bmatrix} + 4\begin{bmatrix}0\\1\end{bmatrix}.$$

Therefore $t = \{-4\} = 0$, $y = D_2$,

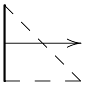
$$\begin{aligned} D_{T,F_1} &= -\sum_{j=0}^{\infty} \frac{b(j+1,0)}{(j+1)!} D_2^j \\ &= -\frac{b(1,0)}{1} - \frac{b(2,0)}{2} D_2 + \cdots \end{aligned}$$

and

$$h'(\mathbf{x}) = D_{T,F_1} \cdot x_1 x_2 = \left(\frac{1}{2} - \frac{1}{12} D_2\right) \cdot x_1 x_2 = \frac{1}{2} x_1 x_2 - \frac{1}{12} x_1.$$

With $x_1 = -z$ *and* $x_2 = 4$, *the contribution of this facet is*

$$\int_{-3}^{-2} -2z + \frac{1}{12} z \, dz = \frac{115}{24}.$$



The normal to the facet F_2 *in the margin is* $\mathbf{n} = (1,0)$. *The vector* $\mathbf{f} = (0,1)$ *is parallel to the facet. We have*

$$\begin{bmatrix} 2 \\ 4 \end{bmatrix} = 4 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 2 \\ 5 \end{bmatrix} = 5 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Therefore $t = \{-2\} = 0$, $y = D_1$,

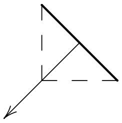
$$\begin{aligned} D_{T,F_2} &= -\sum_{i=0}^{\infty} \frac{b(i+1,0)}{(i+1)!} D_1^i \\ &= -\frac{b(1,0)}{1} - \frac{b(2,0)}{2} D_1 + \cdots \end{aligned}$$

and

$$h'(\mathbf{x}) = D_{T,F_2} \cdot x_1 x_2 = \left(\frac{1}{2} - \frac{1}{12} D_1\right) \cdot x_1 x_2 = \frac{1}{2} x_1 x_2 - \frac{1}{12} x_2.$$

With $x_1 = 2$ *and* $x_2 = z$, *the contribution of this facet is*

$$\int_{4}^{5} z - \frac{1}{12} z \, dz = \frac{33}{8}.$$



The normal to the facet F_3 *in the margin is* $\mathbf{n} = (-1,-1)$. *The vector* $\mathbf{f} = (1,-1)$ *is parallel to the facet. We have*

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} = -\frac{1}{2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} - \frac{7}{2} \begin{bmatrix} -1 \\ -1 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 2 \\ 5 \end{bmatrix} = -\frac{3}{2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} - \frac{7}{2} \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

Therefore $t = \{7\} = 0$, $y = -\frac{1}{2} D_1 - \frac{1}{2} D_2$,

$$\begin{aligned} D_{T,F_3} &= -\sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \frac{b(i+j+1,0)}{(i+j+1)!} \frac{(-1)^{i+j}}{2^{i+j}} D_1^i D_2^j \\ &= -\frac{b(1,0)}{1} + \frac{1}{2} \frac{b(2,0)}{2} D_1 + \frac{1}{2} \frac{b(2,0)}{2} D_2 + \cdots \end{aligned}$$

and

$$h'(\mathbf{x}) = D_{T,F_4} \cdot x_1x_2 = \left(\frac{1}{2} + \frac{1}{24}D_1 + \frac{1}{24}D_2\right) \cdot x_1x_2 = \frac{1}{2}x_1x_2 + \frac{1}{24}x_2 + \frac{1}{24}x_1.$$

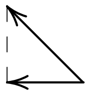
With $x_1 = z + \frac{7}{2}$ and $x_2 = -z + \frac{7}{2}$, the contribution of this facet is

$$\int_{-\frac{3}{2}}^{-\frac{1}{2}} \frac{1}{2}(z + \frac{7}{2})(-z + \frac{7}{2}) + \frac{1}{24}(-z + \frac{7}{2}) + \frac{1}{24}(z + \frac{7}{2})\, dz = \frac{47}{8}.$$

The total contribution of the edges is therefore

$$\frac{115}{24} + \frac{33}{8} + \frac{47}{8} = \frac{355}{24}.$$

Finally, we consider the contributions of the vertices.



For the vertex $\mathbf{v} = (3,4)$, we have $\mathbf{r}_1 = (-1,0)$ and $\mathbf{r}_2 = (-1,1)$. Since $\mathbf{v}$ is integer, we have $t_1 = t_2 = 0$. Also, $C_1 = 1$, $C_2 = 1/2$, $y_1 = -D_1$ and $y_2 = -D_1 + D_2$. Since the total degree of the polynomial x_1x_2 is two, we only need the coefficients of $\mu(K)(\boldsymbol{\xi})$ up to $n_1 + n_2 = 2$.

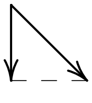
n_1	n_2	
0	0	$\left(\frac{b(1,0)}{1!}\frac{b(1,0)}{1!} - \frac{b(2,0)}{2!}\binom{1}{1}(-1)^1 - \frac{b(2,0)}{2!}\binom{1}{1}(-\frac{1}{2})^1\right)$
1	0	$\left(\frac{b(2,0)}{2!}\frac{b(1,0)}{1!} - \frac{b(3,0)}{3!}\binom{2}{2}(-1)^2 - \frac{b(3,0)}{3!}\binom{2}{1}(-\frac{1}{2})^1\right)(-D_1)$
0	1	$\left(\frac{b(1,0)}{1!}\frac{b(2,0)}{2!} - \frac{b(3,0)}{3!}\binom{2}{1}(-1)^1 - \frac{b(3,0)}{3!}\binom{2}{2}(-\frac{1}{2})^2\right)(-D_1 + D_2)$
2	0	$\left(\frac{b(3,0)}{3!}\frac{b(1,0)}{1!} - \frac{b(4,0)}{4!}\binom{3}{3}(-1)^3 - \frac{b(4,0)}{4!}\binom{3}{1}(-\frac{1}{2})^1\right)(-D_1)^2$
1	1	$\left(\frac{b(2,0)}{2!}\frac{b(2,0)}{2!} - \frac{b(4,0)}{4!}\binom{3}{2}(-1)^2 - \frac{b(4,0)}{4!}\binom{3}{2}(-\frac{1}{2})^2\right)(-D_1)(-D_1 + D_2)$
0	2	$\left(\frac{b(1,0)}{1!}\frac{b(3,0)}{3!} - \frac{b(4,0)}{4!}\binom{3}{1}(-1)^1 - \frac{b(4,0)}{4!}\binom{3}{3}(-\frac{1}{2})^3\right)(-D_1 + D_2)^2$

We find

$$\begin{aligned} h'(\mathbf{x}) &= \left(\frac{3}{8} - \frac{1}{24}(-D_1) - \frac{1}{24}(-D_1 + D_2) + \frac{7}{576}(-D_1D_2) - \frac{5}{1152}(-2D_1D2)\right)x_1x_2 \\ &= \frac{3}{8}x_1x_2 + \frac{1}{24}x_2 - \frac{1}{24}(-x_2 + x_1) + \frac{7}{576}(-1) - \frac{5}{1152}(-2). \end{aligned}$$

The contribution of this vertex is therefore

$$h'(3,4) = \frac{1355}{288}.$$



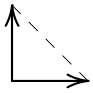
For the vertex $\mathbf{v} = (2,5)$, we have $\mathbf{r}_1 = (0,-1)$ and $\mathbf{r}_2 = (1,-1)$. Since $\mathbf{v}$ is integer, we have $t_1 = t_2 = 0$. Also, $C_1 = 1$, $C_2 = 1/2$, $y_1 = -D_2$ and $y_2 = D_1 - D_2$. We similarly find

$$h'(\mathbf{x}) = \frac{3}{8}x_1x_2 + \frac{1}{24}x_1 - \frac{1}{24}(x_2 - x_1) + \frac{7}{576}(-1) - \frac{5}{1152}(-2).$$

The contribution of this vertex is therefore

$$h'(2,5) = \frac{1067}{288}.$$

For the vertex $\mathbf{v} = (2, 4)$*, we have* $\mathbf{r}_1 = (1, 0)$ *and* $\mathbf{r}_2 = (0, 1)$*. Since* $\mathbf{v}$ *is integer, we have* $t_1 = t_2 = 0$*. The computations are easier in this case since* $C_1 = C_2 = 0$*,* $y_1 = D_1$ *and* $y_2 = D_2$*. We find*



$$h'(\mathbf{x}) = \frac{1}{4}x_1x_2 - \frac{1}{12}x_2 - \frac{1}{12}x_1 + \frac{1}{144}(1).$$

The contribution of this vertex is therefore

$$h'(2, 4) = \frac{253}{144}.$$

The total contribution of the vertices is then

$$\frac{1355}{288} + \frac{1067}{288} + \frac{253}{144} = \frac{61}{6}$$

and the total sum is

$$\frac{121}{24} + \frac{355}{24} + \frac{61}{6} = 30.$$

Example 5.45 *Consider the parametric polytope*

$$P(n) = \{\, \mathbf{x} \mid x_1 \geq 2 \wedge 3x_1 \leq n + 9 \wedge 4 \leq x_2 \leq 5 \,\}.$$

If $n \geq -3$*, then the vertices of this polytope are* $(2, 4)$*,* $(2, 5)$*,* $(3+n/3, 4)$ *and* $(3+n/3, 5)$*. The contributions of the faces of* $P(n)$ *to*

$$\sum_{\mathbf{x} \in P(n) \cap \mathbb{Z}^2} x_1x_2$$

for the chamber $n \geq -3$ *are shown in Table 3. The final result is*

$$\left\{ \tfrac{n^2}{2} - 3n\left\{\tfrac{n}{3}\right\} + \tfrac{21}{2}n + \tfrac{9}{2}\left\{\tfrac{n}{3}\right\}^2 - \tfrac{63}{2}\left\{\tfrac{n}{3}\right\} + 45 \quad \textit{if } n + 3 \geq 0. \right.$$

5.15 Summation through exponential substitution and Laurent expansions (old)

This section was inspired by Baldoni et al. (2008) and presents the old implemenation of the `laurent` summation method. The implementation is described in subsection 5.16.

Let $f(\mathbf{x})$ be the generating function of a polytope P, i.e.,

$$f(\mathbf{x}) = \sum_{\mathbf{t} \in P \cap \mathbb{Z}^d} \mathbf{x}^{\mathbf{t}}.$$

Substituting $\mathbf{x} = \mathbf{e}^{\mathbf{y}}$, we obtain

$$f(\mathbf{e}^{\mathbf{y}}) = \sum_{\mathbf{t} \in P \cap \mathbb{Z}^d} e^{\langle \mathbf{t}, \mathbf{y} \rangle} = \sum_{\mathbf{t} \in P \cap \mathbb{Z}^d} \sum_{\mathbf{n} \geq \mathbf{0}} \frac{\mathbf{t}^{\mathbf{n}} \mathbf{y}^{\mathbf{n}}}{\mathbf{n}!} = \sum_{\mathbf{n} \geq \mathbf{0}} \left(\sum_{\mathbf{t} \in P \cap \mathbb{Z}^d} \mathbf{t}^{\mathbf{n}} \right) \frac{\mathbf{y}^{\mathbf{n}}}{\mathbf{n}!},$$

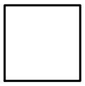
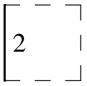
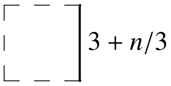
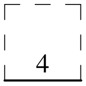
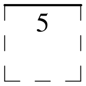
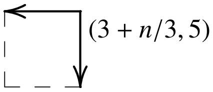
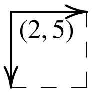
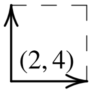
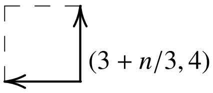
	$\frac{n^2}{4} + \frac{9}{2}n + \frac{45}{4}$
	$\frac{33}{8}$
	$-\frac{3}{2}n\left\{\frac{n}{3}\right\} + \frac{3}{4}n + \frac{9}{4}\left\{\frac{n}{3}\right\}^2 - \frac{63}{4}\left\{\frac{n}{3}\right\} + \frac{57}{8}$
	$\frac{23}{216}n^2 + \frac{23}{12}n + \frac{115}{24}$
	$\frac{31}{216}n^2 + \frac{31}{12}n + \frac{155}{24}$
	$-\frac{31}{36}n\left\{\frac{n}{3}\right\} + \frac{31}{72}n + \frac{31}{24}\left\{\frac{n}{3}\right\}^2 - \frac{217}{24}\left\{\frac{n}{3}\right\} + \frac{589}{144}$
	$\frac{341}{144}$
	$\frac{253}{144}$
	$-\frac{23}{36}n\left\{\frac{n}{3}\right\} + \frac{23}{72}n + \frac{23}{24}\left\{\frac{n}{3}\right\}^2 - \frac{161}{24}\left\{\frac{n}{3}\right\} + \frac{437}{144}$

Table 3: Contributions of the faces of $P(n)$ to the sum of x_1x_2 over the integer points of $P(n)$

with $\mathbf{n}! = n_1!n_2!\cdots n_d!$. We observe that the sum of the monomial $\mathbf{t}^\mathbf{n}$ over the integer points in P is equal to $\mathbf{n}!$ times the coefficient of the $\mathbf{y}^\mathbf{n}$ term in the Taylor expansion of $f(\mathbf{e}^\mathbf{y})$.

As in the case of unweighted counting (see subsection 5.11), we have to add the coefficients of these monomials in the Laurent expansions of the terms in (5.26). However, unlike the case of unweighted counting, we cannot transform this problem to a univariate problem and computing the coefficient of a monomial in the Laurent expansions does not reduce to computing the coefficient of a single higher-degree monomial in a Taylor expansion.

Consider now one of the terms $g(\mathbf{x}) = f_{ik}(\mathbf{x})$ in (5.26),

$$g(\mathbf{e}^\mathbf{y}) = \frac{e^{\sum_{j=1}^d s_j(\mathbf{p})\langle \mathbf{b}_j, \mathbf{y}\rangle}}{\prod_{j=1}^d \left(1 - e^{\langle \mathbf{b}_j, \mathbf{y}\rangle}\right)}, \tag{5.46}$$

with $\mathbf{w}_{ij}(\mathbf{p}) = \sum_{j=1}^d s_j(\mathbf{p})\mathbf{b}_j$ written in terms of the $\mathbf{b}_j$, which are assumed to form a basis and where we have made explicit the only place where the parameters $\mathbf{p}$ appear. We rewrite this equation as

$$g(\mathbf{e}^\mathbf{y}) = \left(\prod_{j=1}^d \frac{-1}{\langle \mathbf{b}_j, \mathbf{y}\rangle}\right)\left(\prod_{j=1}^d \frac{-\langle \mathbf{b}_j, \mathbf{y}\rangle\, e^{s_j(\mathbf{p})\langle \mathbf{b}_j, \mathbf{y}\rangle}}{1 - e^{\langle \mathbf{b}_j, \mathbf{y}\rangle}}\right). \tag{5.47}$$

The second factor is analytic and is a product of generating functions $\operatorname{Todd}(s_j(\mathbf{p}), \langle \mathbf{b}_j, \mathbf{y}\rangle)$ of Bernoulli polynomials (5.38). Plugging in these expressions, we find

$$\begin{aligned}
\operatorname{Todd}(s_j(\mathbf{p}), \langle \mathbf{b}_j, \mathbf{y}\rangle) &= \frac{-\langle \mathbf{b}_j, \mathbf{y}\rangle e^{s_j(\mathbf{p})\langle \mathbf{b}_j, \mathbf{y}\rangle}}{1 - e^{\langle \mathbf{b}_j, \mathbf{y}\rangle}} \\
&= \sum_{n=0}^{\infty} \frac{b(n, s_j(\mathbf{p}))}{n!} \langle \mathbf{b}_j, \mathbf{y}\rangle^n \\
&= \sum_{\mathbf{k}\geq\mathbf{0}} \frac{b(\sum k_i, s_j(\mathbf{p}))}{(\sum k_i)!} \binom{\sum k_i}{\mathbf{k}} \mathbf{b}_j^\mathbf{k} \mathbf{y}^\mathbf{k} \\
&= \sum_{\mathbf{k}\geq\mathbf{0}} \frac{b(\sum k_i, s_j(\mathbf{p}))}{\prod_i k_i!} \mathbf{b}_j^\mathbf{k} \mathbf{y}^\mathbf{k},
\end{aligned} \tag{5.48}$$

with

$$\binom{\sum k_i}{\mathbf{k}} = \binom{\sum k_i}{k_1, k_2, \ldots k_d} = \frac{(\sum k_i)!}{\mathbf{k}!} = \prod_{i=1}^d \binom{\sum_{j=1}^i k_j}{k_i}$$

the multinomial coefficients. For the first factor, we compute the Laurent expansion of each of its factors,

$$\begin{aligned}
\frac{-1}{\langle \mathbf{b}_j, \mathbf{y}\rangle} &= \frac{-1}{\sum_{k=f}^{d} b_{jk} y_k} \\
&= \frac{-1}{b_{jf} y_f \left(1 + \frac{\sum_{k=f+1}^{d} b_{jk} y_k}{b_{jf} y_f}\right)} \\
&= \frac{-1}{b_{jf} y_f} \sum_{n=0}^{\infty} (-1)^n \left(\frac{\sum_{k=f+1}^{d} b_{jk} y_k}{b_{jf} y_f}\right)^n \\
&= \sum_{\mathbf{n} \geq \mathbf{0}} \binom{\sum n_k}{\mathbf{n}} (-1)^{1+\sum n_k} \frac{\mathbf{b}_j'^{\mathbf{n}}}{b_{jf}^{1+\sum n_k}} \frac{\mathbf{y}'^{\mathbf{n}}}{y_f^{1+\sum n_k}},
\end{aligned} \tag{5.49}$$

where b_{jf} is the first non-zero coefficient of $\mathbf{b}_j$ and the vector $\mathbf{b}_j'$ contains the subsequent $d - f$ coefficients of $\mathbf{b}_j$.

Given a polynomial

$$q(\mathbf{y}, \mathbf{p}) = \sum_{\mathbf{m}} \beta_{\mathbf{m}}(\mathbf{p})\, \mathbf{y}^{\mathbf{m}}$$

that we wish to sum over the integer points of a polytope P, we perform the following operations for each unimodular cone in the decomposition of each vertex cone.

- For each $\mathbf{m}$ with $\beta_{\mathbf{m}}(\mathbf{p}) \neq 0$
  - Compute all sums $\mathbf{N} = \sum_{j=1}^{d} (\mathbf{0}, -\sum_k n_{jk} - 1, \mathbf{n}_j)$ of exponents from (5.49) such that $\mathbf{N} \leq \mathbf{m}$ and compute the corresponding coefficient $\gamma_{\mathbf{N}}$ in the product of Laurent series by enumerating all combinations of $\mathbf{n}_j$ leading to the same $\mathbf{N}$. Note that there are only a finite number of $\mathbf{N}$ satisfying this constraint since $\sum N_k = -d$. By reordering the variables such that the highest exponents occurs for the first variable, the number of $\mathbf{N}$ can be reduced.
  - For each of these $\mathbf{N}$
    - ∗ Compute the coefficient $\delta_{\mathbf{m}-\mathbf{N}}(\mathbf{p})$ of $\mathbf{y}^{\mathbf{m}-\mathbf{N}}$ in the product of Taylor expansions (5.48).
- The contribution of this cone is the sum of

  $$\mathbf{m}!\, \alpha\, \beta_{\mathbf{m}}(\mathbf{p})\, \gamma_{\mathbf{N}}\, \delta_{\mathbf{m}-\mathbf{N}}(\mathbf{p})$$

  over all considered $\mathbf{m}$ and $\mathbf{N}$.

Within each vertex cone computation, the coefficients $\gamma_{\mathbf{N}}$ and $\delta_{\mathbf{m}-\mathbf{N}}(\mathbf{p})$ only need to be computed once.

Example 5.50 *Consider once more the rational generating function*

$$f(T; \mathbf{x}) = \frac{x_1^2}{(1 - x_1^{-1})(1 - x_1^{-1} x_2)} + \frac{x_2^2}{(1 - x_2^{-1})(1 - x_1 x_2^{-1})} + \frac{1}{(1 - x_1)(1 - x_2)}$$

from Verdoolaege (2005, Example 39) and Example 5.30. Assume we want to compute

$$\sum_{\mathbf{y} \in T \cap \mathbb{Z}^2} y_1^2 + y_2^2.$$

We will need the following Bernoulli polynomials

$$\begin{aligned}
b(0, s) &= 1 \\
b(1, s) &= \frac{1}{2}\,(-1 + 2s) \\
b(2, s) &= \frac{1}{6}\left(1 - 6s + 6s^2\right) \\
b(3, s) &= \frac{1}{2}\left(s - 3s^2 + 2s^3\right) \\
b(4, s) &= \frac{1}{30}\left(-1 + 30s^2 - 60s^3 + 30s^4\right)
\end{aligned}$$

For the first term, we have $(2, 0) = -2\,(-1, 0) + 0\,(-1, 1)$ *and substitution yields*

$$\begin{aligned}
h(\mathbf{y}) =& \frac{1}{y_1}\frac{1}{y_1 - y_2}\frac{y_1 e^{(-2)(-y_1)}}{1 - e^{-y_1}}\frac{(y_1 - y2)e^{0(-y_1+y_2)}}{1 - e^{-y_1+y_2}} \\
=& \frac{1}{y_1}\left(\frac{1}{y_1}\left(1 + \frac{y_2}{y_1} + \frac{y_2^2}{y_1^2} + \cdots\right)\right) \\
& \left(1 + \frac{b(1,-2)}{1}(-y_1) + \frac{b(2,-2)}{2}(-y_1)^2 + \frac{b(3,-2)}{3!}(-y_1)^3 + \frac{b(4,-2)}{4!}(-y_1)^4 + \cdots\right) \\
& \left(1 + \frac{-1}{2}(-y_1 + y_2) + \frac{1}{12}(-y_1 + y_2)^2 + 0(-y_1 + y_2)^3 + \frac{1}{720}(-y_1 + y_2)^4 + \cdots\right)
\end{aligned}$$

We obtain the following results:

$\mathbf{m}$	$\mathbf{N}$	$\gamma_{\mathbf{N}}\mathbf{y}^{\mathbf{N}}$	$\mathbf{m} - \mathbf{N}$	$\delta_{\mathbf{m}-\mathbf{N}}\mathbf{y}^{\mathbf{m}-\mathbf{N}}$	$\mathbf{m}!\alpha\beta_{\mathbf{m}}\gamma_{\mathbf{N}}\delta_{\mathbf{m}-\mathbf{N}}$
$(2, 0)$	$(-2, 0)$	$1y_1^{-2}$	$(4, 0)$	$\frac{721}{240}y_1^4$	$\frac{721}{120}$
$(0, 2)$	$(-2, 0)$	$1y_1^{-2}$	$(2, 2)$	$\frac{179}{720}y_1^2y_2^2$	$\frac{179}{360}$
	$(-3, 1)$	$1y_1^{-3}y_2$	$(3, 1)$	$-\frac{211}{120}y_1^3y_1$	$-\frac{211}{60}$
	$(-4, 2)$	$1y_1^{-4}y_2^2$	$(4, 0)$	$\frac{721}{240}y_1^4$	$\frac{721}{120}$

For the second term, we similarly obtain

$\mathbf{m}$	$\mathbf{N}$	$\gamma_{\mathbf{N}}\mathbf{y}^{\mathbf{N}}$	$\mathbf{m}-\mathbf{N}$	$\delta_{\mathbf{m}-\mathbf{N}}\mathbf{y}^{\mathbf{m}-\mathbf{N}}$	$\mathbf{m}!\alpha\beta_{\mathbf{m}}\gamma_{\mathbf{N}}\delta_{\mathbf{m}-\mathbf{N}}$
$(2,0)$	$(-1,-1)$	$-1y_1^{-1}y_2^{-1}$	$(3,1)$	$\frac{1}{180}y_1^3y_1$	$-\frac{1}{90}$
	$(-2,0)$	$-1y_1^{-2}$	$(4,0)$	$-\frac{1}{720}y_1^4$	$\frac{1}{360}$
$(0,2)$	$(-1,-1)$	$-1y_1^{-1}y_2^{-1}$	$(1,3)$	$-\frac{211}{120}y_1y_2^3$	$\frac{211}{60}$
	$(-2,0)$	$-1y_1^{-3}y_2$	$(2,2)$	$\frac{179}{720}y_1^3y_2$	$-\frac{179}{360}$
	$(-3,1)$	$-1y_1^{-3}y_2$	$(3,1)$	$\frac{1}{180}y_1^3y_1$	$-\frac{1}{90}$
	$(-4,2)$	$-1y_1^{-4}y_2^2$	$(4,0)$	$-\frac{1}{720}y_1^4$	$\frac{1}{360}$

Finally, for the third term, we obtain

$\mathbf{m}$	$\mathbf{N}$	$\gamma_{\mathbf{N}}\mathbf{y}^{\mathbf{N}}$	$\mathbf{m}-\mathbf{N}$	$\delta_{\mathbf{m}-\mathbf{N}}\mathbf{y}^{\mathbf{m}-\mathbf{N}}$	$\mathbf{m}!\alpha\beta_{\mathbf{m}}\gamma_{\mathbf{N}}\delta_{\mathbf{m}-\mathbf{N}}$
$(2,0)$	$(-1,-1)$	$-1y_1^{-1}y_2^{-1}$	$(3,1)$	$0y_1^3y_1$	0
$(0,2)$	$(-1,-1)$	$-1y_1^{-1}y_2^{-1}$	$(1,3)$	$0y_1y_2^3$	0

Adding up all contributions in the final columns of these tables, we obtain a grand total of

$$12.$$

5.16 Summation through exponential substitution and Laurent expansions

This section was inspired by Baldoni et al. (2009).

As in subsection 5.15, we want to compute $\mathbf{n}!$ times the coefficient of the $\mathbf{y}^{\mathbf{n}}$ term in the Taylor expansion of $f(\mathbf{e}^{\mathbf{y}})$. However, we do not write (5.46), i.e.,

$$g(\mathbf{e}^{\mathbf{y}}) = \frac{e^{\sum_{j=1}^{d} s_j(\mathbf{p})\langle \mathbf{b}_j,\mathbf{y}\rangle}}{\prod_{j=1}^{d}\left(1-e^{\langle \mathbf{b}_j,\mathbf{y}\rangle}\right)}$$

as in (5.47), but as

$$g(\mathbf{e}^{\mathbf{y}}) = \prod_{j=1}^{d}\left(-\frac{1}{\langle \mathbf{b}_j,\mathbf{y}\rangle} + \left(\frac{1}{\langle \mathbf{b}_j,\mathbf{y}\rangle} + \frac{e^{s_j(\mathbf{p})\langle \mathbf{b}_j,\mathbf{y}\rangle}}{1-e^{\langle \mathbf{b}_j,\mathbf{y}\rangle}}\right)\right)$$

instead. The second term in each factor is of the form

$$
\begin{aligned}
\frac{1}{x} + \frac{e^{ux}}{1-e^{x}} &= \frac{1}{x} + \frac{1}{x}\frac{x\,e^{ux}}{1-e^{x}} \\
&= \frac{1}{x} - \sum_{k=0}^{\infty} \frac{b(k,u)}{k!}\, x^{k-1} \\
&= -\sum_{k=1}^{\infty} \frac{b(k,u)}{k!}\, x^{k-1} \\
&= -\sum_{k=0}^{\infty} \frac{b(k+1,u)}{(k+1)!}\, x^{k},
\end{aligned}
$$

with $b(k,u)$ the Bernoulli polynomials. The second line follows from (5.38) and the third line follows from $b(0,u) = 1$. We therefore have

$$
\begin{aligned}
g(\mathbf{e}^{\mathbf{y}}) &= \prod_{j=1}^{d} \left(-\frac{1}{\langle \mathbf{b}_j, \mathbf{y} \rangle} - \sum_{k=0}^{\infty} \frac{b(k+1, s_j(\mathbf{p}))}{(k+1)!} \langle \mathbf{b}_j, \mathbf{y} \rangle^{k} \right) \\
&= \prod_{j=1}^{d} \left(\left(\sum_{\mathbf{n} \geq \mathbf{0}} (-1)^{1+\sum n_k} \binom{\sum n_k}{\mathbf{n}} \frac{\mathbf{b}_j'^{\mathbf{n}}}{b_{jf_j}^{1+\sum n_k}} \frac{\mathbf{y}'^{\mathbf{n}}}{y_{f_j}^{1+\sum n_k}} \right) - \sum_{\mathbf{n} \geq \mathbf{0}} \frac{b(1+\sum n_k, s_j(\mathbf{p}))}{(1+\sum n_k)\prod n_k!} \mathbf{b}_j^{\mathbf{n}} \mathbf{y}^{\mathbf{n}} \right),
\end{aligned}
$$

where b_{jf_j} is the first non-zero coefficient of $\mathbf{b}_j$ and the vector $\mathbf{b}_j'$ contains the subsequent $d - f_j$ coefficients of $\mathbf{b}_j$. The expansion of the first term follows from (5.49), while the expansion of the second term follows from (5.48). Note, however, that unlike subsection 5.15, each factor is now a *sum* of two series instead of a product of two series. In particular, there are two kinds of (non-overlapping) terms. In the first kind of terms, there is exactly one variable y_{f_j} with a negative exponent, where f_j is such that b_{jf_j} is the first non-zero coefficient of $\mathbf{b}_j$, and the sum of all exponents is -1. Let $\mathbf{n}$ contain the non-negative exponents, then the coefficient of such a term is

$$
(-1)^{1+\sum n_k} \binom{\sum n_k}{\mathbf{n}} \frac{\mathbf{b}_j'^{\mathbf{n}}}{b_{jf}^{1+\sum n_k}}. \tag{5.51}
$$

In the second kind of terms, all exponents are non-negative and the coefficient is given by

$$
-\sum_{\mathbf{n} \geq \mathbf{0}} \frac{b(1+\sum n_k, s_j(\mathbf{p}))}{(1+\sum n_k)\prod n_k!} \mathbf{b}_j^{\mathbf{n}}. \tag{5.52}
$$

Given, as before, a polynomial

$$
q(\mathbf{y}, \mathbf{p}) = \sum_{\mathbf{m}} \beta_{\mathbf{m}}(\mathbf{p})\, \mathbf{y}^{\mathbf{m}}
$$

that we wish to sum over the integer points of a polytope P, we perform the following operations for each unimodular cone in the decomposition of each vertex cone.

- For each $\mathbf{m}$ with $\beta_{\mathbf{m}}(\mathbf{p}) \neq 0$

  - Consider all decomposition $\mathbf{m} = \sum_j \mathbf{n}_j$ such that each $\mathbf{n}_j$ corresponds to one of the two kinds of terms, i.e., either $n_{jf_j} < 0$ and $\sum_k n_{jk} = -1$ or $\mathbf{n}_j \geq \mathbf{0}$.
  - For each such decomposition, compute the coefficient $\alpha_{\mathbf{m}}(\mathbf{p})$ of $\mathbf{y}^{\mathbf{m}}$ as the product of the corresponding coefficients from (5.51) and (5.52).

- The contribution of the cone is the sum of

$$\mathbf{m}!\,\alpha(\mathbf{m})$$

over all these $\mathbf{m}$.

Example 5.53 *As in Example 5.50, let us compute*

$$\sum_{\mathbf{y}\in T\cap\mathbb{Z}^2} y_1^2 + y_2^2,$$

with T a triangle with generating function

$$f(T;\mathbf{x}) = \frac{x_1^2}{(1-x_1^{-1})(1-x_1^{-1}x_2)} + \frac{x_2^2}{(1-x_2^{-1})(1-x_1x_2^{-1})} + \frac{1}{(1-x_1)(1-x_2)}.$$

Consider the first term. As before, we write the exponent of the numerator of this term as $(2,0) = -2\,(-1,0) + 0\,(-1,1)$ *and so we obtain*

$$\begin{aligned}
h(\mathbf{y}) =& \left(\frac{1}{y_1} + \left(\frac{1}{-y_1} + \frac{e^{(-2)(-y_1)}}{1-e^{-y_1}}\right)\right)\left(\frac{1}{y_1-y_2} + \left(\frac{1}{-y_1+y_2} + \frac{e^{0(-y_1+y_2)}}{1-e^{-y_1+y_2}}\right)\right)\\
=& \left(\frac{1}{y_1} + \sum_{n=0}^{\infty}\frac{b(n+1,-2)}{(n+1)n!}(-1)^{n+1}y_1^n\right)\\
& \left(\sum_{n\geq 0}(-1)^{n+1}\binom{n}{n}\frac{1^n}{-1^{1+n}}\frac{y_2^n}{y_1^{1+n}} + \sum_{\mathbf{n}\geq 0}\frac{b(1+n_1+n_2,0)}{(1+n_1+n_2)n_1!n_2!}(-1)^{n_1}1^{n_2}y_1^{n_1}y_2^{n_2}\right)\\
=& \left(\frac{1}{y_1} + \frac{5}{2} + \frac{37}{12}y_1 + \frac{5}{2}y_1^2 + \frac{1079}{702}y_1^3 + \ldots\right)\\
& \left(\frac{1}{y_1} + \frac{y_2}{y_1^2} + \frac{y_2^2}{y_1^3} + \ldots + \frac{1}{2} + \frac{1}{12}y_1 - \frac{1}{12}y_2 + 0y_1^2 + \frac{1}{6}y_1y_2 - \frac{1}{12}y_2^2 - \frac{1}{720}y_1^3 + \frac{1}{240}y_1^2y_2 + \ldots\right).
\end{aligned}$$

The coefficient of y_1^2 is then

$$1(-\frac{1}{720}) + 0 + \frac{37}{12}\frac{1}{12} + \frac{5}{2}\frac{1}{2} + \frac{1079}{720}1 = \frac{721}{240},$$

which is the same value as the one we found in Example 5.50.

To compare the difference between the old implementation described in subsection 5.15 and the new implementation described here, we repeat the experiments of Verdoolaege and Bruynooghe (2008). The new experiments were performed on the same

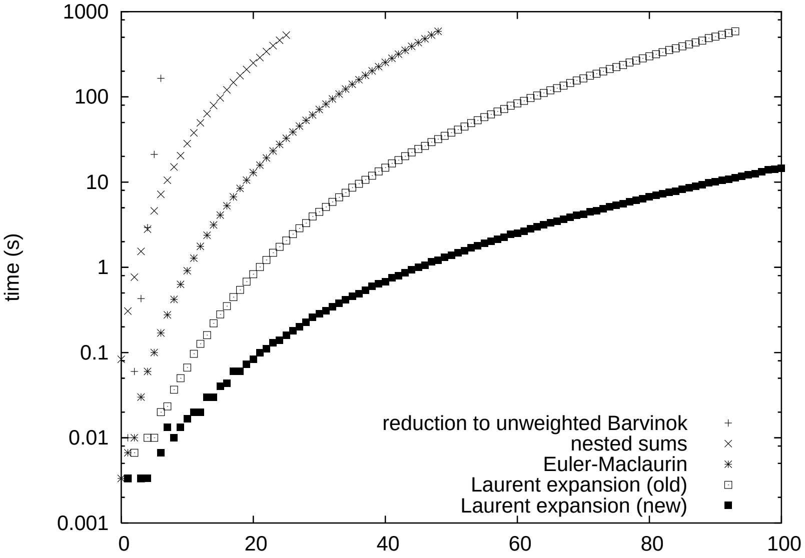


Figure 5.54: Execution times for summing a monomial over a difficult non-parametric triangle

hardware (Intel Core2) as the old experiments, but using barvinok version `barvinok-0.28-45-ga380232`. It should be noted that the running times reported by Verdoolaege and Bruynooghe (2008) had been mistakenly scaled down by a factor of 0.6. The experiments were cut off at 10 minutes (600 seconds). The new results are shown in Figures 5.54 and 5.55. The time measuring resolution is 0.01s, so very short execution times are not measured very accurately, resulting in some anomalies in the graphs. In the non-parametric triangle experiment in Figure 5.54, the new Laurent methods is by far the best. For the parametric triangle, Euler-Maclaurin still beats Laurent, but the new version is getting much closer than the old version.

5.17 Conversion to “standard form”

Some algorithms or tools expect a polyhedron to be specified in “standard form”, i.e.,

$$\begin{cases} A\mathbf{x} = \mathbf{b} \\ \quad \mathbf{x} \geq \mathbf{0}. \end{cases} \tag{5.56}$$

Given an arbitrary (parametric) polyhedron

$$\{\, \mathbf{x} \mid A\mathbf{x} + \mathbf{b}(\mathbf{p}) \geq 0 \,\}, \tag{5.57}$$

a conversion to standard form requires the introduction of slack variables and a way of dealing with variables of unrestricted sign. In this section we will be satisfied with a

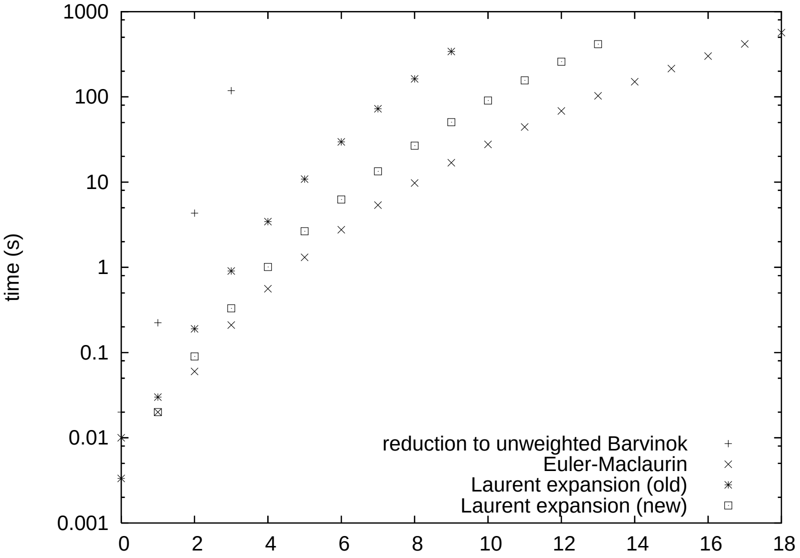


Figure 5.55: Execution times for summing a monomial over a difficult parametric triangle

reduction to the form

$$\begin{cases} A\mathbf{x} = \mathbf{b} \\ D\mathbf{x} \geq \mathbf{c}, \end{cases} \tag{5.58}$$

with D a diagonal matrix with positive entries. That is, we do not necessarily make all variables non-negative, but we do ensure that they have a lower bound. If needed, a subsequent reduction can then be performed.

The standard way of dealing with variables of unrestricted sign is to replace a variable x of unknown sign by the difference ($x = x' - x''$) of two non-negative variables ($x', x'' \geq 0$). However, some algorithms are somewhat sensitive with respect to the number of variables and so we would prefer to introduce as few extra variables as possible. We will therefore apply a unimodular transformation on the variables such that all transformed variables are known to be non-negative.

The first step is to compute the HNF of A, i.e., a matrix $H = AU$, with U unimodular, in column echelon form such that the first entry in each column is positive and the other entries on the corresponding row are non-negative and strictly smaller than this first entry. By reordering the rows we may assume that the top square part of H is lower-triangular. By a further unimodular transformation, the entries below the diagonal can be made non-positive and strictly smaller (in absolute value) than the diagonal entry of the same row.

For each of the new variables, we can take a positive combination of the corresponding row and the previous rows to obtain a positive multiple of the corresponding unit vector, implying that the variable has a lower bound. A slack variable can then be introduced for each of the rows in the top square part of H' that is not already a positive multiple of a unit vector and for each of the rows below the top square part of H'.

Example 5.59 *Consider the cone*

$$\left\{ \mathbf{x} \mid \begin{bmatrix} 67 & -13 \\ -52 & 53 \end{bmatrix} \mathbf{x} \geq \mathbf{0} \right\}.$$

This cone is already situated in the first quadrant, but this may not be obvious from the constraints. Furthermore, directly adding slack variables would lead to a total of 4 variables, whereas we can also represent this cone in standard form with only 3 variables. We have

$$H' = \begin{bmatrix} 1 & 0 \\ -1331 & 2875 \end{bmatrix} = \begin{bmatrix} 67 & -13 \\ -52 & 53 \end{bmatrix} \begin{bmatrix} -6 & 13 \\ -31 & 57 \end{bmatrix} = AU'.$$

Adding a slack variable for the second row of H', we obtain the equivalent problem

$$\begin{cases} \begin{bmatrix} -1331 & 2875 & -1 \end{bmatrix} \mathbf{x}' = \mathbf{0} \\ \mathbf{x}' \geq \mathbf{0} \end{cases}$$

with

$$\mathbf{x} = \begin{bmatrix} -6 & 13 & 0 \\ -31 & 57 & 0 \end{bmatrix} \mathbf{x}'.$$

A similar construction was used by Eisenbrand (2000, Lemma 3.10) and Hung and Rom (1990).

5.18 Using TOPCOM to compute Chamber Decompositions

In this section, we describe how to use the correspondence between the regular triangulations of a point set and the chambers of the Gale transform of the point set (Gelfand et al. 1994) to compute the chamber decomposition of a parametric polytope. This correspondence was also used by Pfeifle and Rambau (2003) Eisenschmidt and Köppe (2007).

Let us first assume that the parametric polytope can be written as

$$\begin{cases} \mathbf{x} \geq 0 \\ A\,\mathbf{x} \leq \mathbf{b}(\mathbf{p}), \end{cases} \tag{5.60}$$

where the right hand side $\mathbf{b}(\mathbf{p})$ is arbitrary and may depend on the parameters. The first step is to add slack variables $\mathbf{s}$ to obtain the vector partition problem

$$\begin{cases} A\,\mathbf{x} + I\,\mathbf{s} = \mathbf{b}(\mathbf{p}) \\ \mathbf{x}, \mathbf{s} \geq 0, \end{cases}$$

with I the identity matrix. Then we compute the (right) kernel K of the matrix $\begin{bmatrix} A & I \end{bmatrix}$, i.e.,

$$\begin{bmatrix} A & I \end{bmatrix} K = 0$$

and use `TOPCOM`'s `points2triangs` to compute the regular triangulations of the points specified by the rows of K. Each of the resulting triangulations corresponds to a chamber in the chamber complex of the above vector partition problem. Each simplex in a triangulation corresponds to a parametric vertex active on the corresponding chamber and each point in the simplex (i.e., a row of K) corresponds to a variable (x_j or s_j) that is set to zero to obtain this parametric vertex. In the original formulation of the problem (5.60) each such variable set to zero reflects the saturation of the corresponding constraint ($x_j = 0$ for $x_j = 0$ and $\langle \mathbf{a}_j, \mathbf{x} \rangle = b_j(\mathbf{p})$ for $s_j = 0$). A description of the chamber can then be obtained by plugging in the parametric vertices in the remaining constraints.

Example 5.61 *Consider the parametric polytope*

$$P(p,q,r) = \{\, (i,j) \mid 0 \leq i \leq p \wedge 0 \leq j \leq 2i+q \wedge 0 \leq k \leq i-p+r \wedge p \geq 0 \wedge q \geq 0 \wedge r \geq 0 \,\}.$$

The constraints involving the variables are

$$\left\{\begin{array}{l} \begin{bmatrix}1 & & \\ & 1 & \\ & & 1\end{bmatrix}\begin{bmatrix}i\\ j\\ k\end{bmatrix}\begin{matrix}\geq 0\\ \geq 0\\ \geq 0\end{matrix} \\ \begin{bmatrix}1 & 0 & 0\\ -1 & 0 & 1\\ -2 & 1 & 0\end{bmatrix}\begin{bmatrix}i\\ j\\ k\end{bmatrix}\begin{matrix}\leq p\\ \leq q\\ \leq -p+r\end{matrix}\end{array}\right.$$

We have

$$\begin{bmatrix}1 & 0 & 0 & 1 & 0 & 0\\ -1 & 0 & 1 & 0 & 1 & 0\\ -2 & 1 & 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}-1 & 0 & 0\\ -2 & 0 & -1\\ -1 & -1 & 0\\ 1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} = 0$$

Computing the regular triangulations of the rows of K using `TOPCOM`*, we obtain*

```
> cat e2.topcom
[
[  -1   0   0 ]
[  -2   0  -1 ]
[  -1  -1   0 ]
[   1   0   0 ]
[   0   1   0 ]
[   0   0   1 ]
]
> points2triangs --regular < e2.topcom
T[1]:={{0,1,2},{1,2,3},{0,1,4},{1,3,4},{0,2,5},{2,3,5},{0,4,5},{3,4,5}};
T[2]:={{1,2,3},{1,3,4},{2,3,5},{3,4,5},{1,2,5},{1,4,5}};
T[3]:={{1,2,3},{1,3,4},{2,3,5},{3,4,5},{1,2,4},{2,4,5}};
```

We see that we have three chambers in the decomposition, one with 8 vertices and two with 6 vertices. Take the second vertex ("`{1,2,3}`") of the first chamber. This vertex corresponds to the saturation of the constraints $j \geq 0$, $k \geq 0$ and $i \leq p$, i.e., $(i, j, k) = (p, 0, 0)$. Plugging in this vertex in the remaining constraints, we see that it is only valid in case $p \geq 0$, $r \geq 0$ and $2p + q \geq 0$. For the remaining vertices of the first chamber, we similarly find

`{0,1,2}`	$(0, 0, 0)$	$p \geq 0$, $-q + r \geq 0$ *and* $q \geq 0$
`{1,2,3}`	$(p, 0, 0)$	$p \geq 0$, $r \geq 0$ *and* $2p + q \geq 0$
`{0,1,4}`	$(0, 0, -p + r)$	$-q + r \geq 0$, $p \geq 0$ *and* $q \geq 0$
`{1,3,4}`	$(p, 0, r)$	$p \geq 0$, $r \geq 0$ *and* $2p + q \geq 0$
`{0,2,5}`	$(0, q, 0)$	$q \geq 0$, $p \geq 0$ *and* $-q + r \geq 0$
`{2,3,5}`	$(p, 2p + q, 0)$	$p \geq 0$, $2p + q \geq 0$ *and* $r \geq 0$
`{0,4,5}`	$(0, q, -p + r)$	$q \geq 0$, $-q + r \geq 0$ *and* $p \geq 0$
`{3,4,5}`	$(p, 2p + q, r)$	$p \geq 0$, $2p + q \geq 0$ *and* $r \geq 0$

Combining these constraints with the initial constraints of the problem on the parameters $p \geq 0$, $q \geq 0$ and $r \geq 0$, we find the chamber $\{\, (p, q, r) \mid p \geq 0 \wedge -p + r \geq 0 \wedge q \geq 0 \,\}$. For the second chamber, we have

`{1,2,3}`	$(p, 0, 0)$	$p \geq 0$, $r \geq 0$ *and* $2p + q \geq 0$
`{1,3,4}`	$(p, 0, r)$	$p \geq 0$, $r \geq 0$ *and* $2p + q \geq 0$
`{2,3,5}`	$(p, 2p + q, 0)$	$p \geq 0$, $2p + q \geq 0$ *and* $r \geq 0$
`{3,4,5}`	$(p, 2p + q, r)$	$p \geq 0$, $2p + q \geq 0$ *and* $r \geq 0$
`{1,2,5}`	$(-\frac{q}{2}, 0, 0)$	$-q \geq 0$, $2p + q \geq 0$ *and* $-2p - q + 2r \geq 0$
`{1,4,5}`	$(-\frac{q}{2}, 0, -p - \frac{q}{2} + r)$	$-q \geq 0$, $-2p - q + 2r \geq 0$ *and* $2p + q \geq 0$

The chamber is therefore $\{\, (p, q, r) \mid q = 0 \wedge p \geq 0 \wedge -p + r \geq 0 \,\}$. Note that by intersecting with the initial constraints this chamber is no longer full-dimensional and can therefore be discarded. Finally, for the third chamber, we have

`{1,2,3}`	$(p, 0, 0)$	$p \geq 0$, $r \geq 0$ *and* $2p + q \geq 0$
`{1,3,4}`	$(p, 0, r)$	$p \geq 0$, $r \geq 0$ *and* $2p + q \geq 0$
`{2,3,5}`	$(p, 2p + q, 0)$	$p \geq 0$, $2p + q \geq 0$ *and* $r \geq 0$
`{3,4,5}`	$(p, 2p + q, r)$	$p \geq 0$, $2p + q \geq 0$ *and* $r \geq 0$
`{1,2,4}`	$(p - r, 0, 0)$	$p - r \geq 0$, $r \geq 0$ *and* $2p + q - 2r \geq 0$
`{2,4,5}`	$(p - r, 2p + q - 2r, 0)$	$p - r \geq 0$, $2p + q - 2r \geq 0$ *and* $r \geq 0$

The chamber is therefore $\{\, (p, q, r) \mid p - r \geq 0 \wedge q \geq 0 \wedge r \geq 0 \,\}$.

Now let us consider general parametric polytopes. First note that we can follow the same procedure as above if we replace $\mathbf{x}$ by $\mathbf{x}' - \mathbf{c}(\mathbf{p})$ in (5.60), i.e., if our problem has the form

$$\begin{cases} \mathbf{x}' \geq \mathbf{c}(\mathbf{p}) \\ A\,\mathbf{x}' \leq \mathbf{b}(\mathbf{p}) + A\mathbf{c}(\mathbf{p}), \end{cases} \tag{5.62}$$

as saturating a constraint $x_i = 0$ is equivalent to saturating the constraint $x'_i = c_i(\mathbf{p})$ and, similarly, $\langle \mathbf{a}_j, \mathbf{x} \rangle = b_j(\mathbf{p})$ is equivalent to $\langle \mathbf{a}_j, \mathbf{x}' \rangle = b_j(\mathbf{p}) + \langle \mathbf{a}_j, \mathbf{c}(\mathbf{p}) \rangle$.

In the general case, the problem has the form

$$A\mathbf{x} \geq \mathbf{b}(\mathbf{p})$$

and then we apply the technique of subsection 5.17. Let A' be a non-singular square submatrix of A with the same number of columns and compute the (left) HNF $H =$

$A'U$ with U unimodular and H lower-triangular with non-positive elements below the diagonal. Replacing $\mathbf{x}$ by $U\mathbf{x}'$, we obtain

$$\begin{cases} H\mathbf{x}' \geq \mathbf{b}'(\mathbf{p}) \\ -A''U\,\mathbf{x}' \leq -\mathbf{b}''(\mathbf{p}), \end{cases}$$

with A'' the remaining rows of A and $\mathbf{b}(\mathbf{p})$ split in the same way. If H happens to be the identity matrix, then our problem is of the form (5.62) and we already know how to solve this problem. Note that, again, saturating any of the transformed constraints in $\mathbf{x}'$ is equivalent to saturating the corresponding constraint in $\mathbf{x}$. We therefore only need to compute $-A''U$ for the computation of the kernel K. To construct the parametric vertices in the original coordinate system, we can simply use the original constraints. The same reasoning holds if H is any diagonal matrix, since we can virtually replace $H\mathbf{x}$ by $\mathbf{x}'$ without affecting the non-negativity of the variables.

If H is not diagonal, then we can introduce new constraints $x'_j \geq d(\mathbf{p})$, where $d(\mathbf{p})$ is some symbolic constant. These constraints do not remove any solutions since each row in H expresses that the corresponding variable is greater than or equal to a non-negative combination of the previous variables plus some constant. We can then proceed as before. However, to reduce unnecessary computations we may remove from K the rows that correspond to these new rows. Any solution saturating the new constraint, would also saturate the corresponding constraint $\mathbf{h}_j^T$ and all the constraints corresponding to the non-zero entries in $\mathbf{h}_j^T$. If a chamber contains a vertex obtained by saturating such a new constraint, it would appear multiple times in the same chamber, each time combined with different constraints from the above set. Furthermore, there would also be another (as it turns out, identical) chamber where the vertex is only defined by the other constraints.

Example 5.63 *Consider the parametric polytope*

$$P(n) = \{\, (i,j) \mid 1 \leq i \wedge 2i \leq 3j \wedge j \leq n \,\}.$$

The constraints are

$$\begin{bmatrix} 1 & 0 \\ -2 & 3 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 1 \\ 0 \\ -n \end{bmatrix}.$$

The top 2×2 *submatrix is already in HNF. We have* $3j \geq 2i \geq 2$*, so we can add a constraint of the form* $j \geq c(n)$ *and obtain*

$$\begin{bmatrix} A & I \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 2 & -3 & 0 & 1 \end{bmatrix},$$

while K *with* $\begin{bmatrix} A & I \end{bmatrix} K = 0$ *is given by*

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 2 & -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -2 & 3 \end{bmatrix}.$$

The second row of K corresponds to the second variable, which in turn corresponds to the newly added constraint. Passing all rows of K to `TOPCOM` *we would get*

```
> points2triangs --regular <<EOF
> [[1 0],[0,1],[0,-1],[-2,3]]
> EOF
T[1]:={{0,1},{0,2},{1,3},{2,3}};
T[2]:={{0,2},{2,3},{0,3}};
T[3]:={};
```

The first vertex in the first chamber saturates the second row (row 1) and therefore saturates both the first (0) and fourth (3) and it appears a second time as `{1,3}`*. Combining these "two" vertices into one as* `{0,3}` *results in the second (identical) chamber. Removing the row corresponding to the new constraint from K we remove the duplicates*

```
> points2triangs --regular <<EOF
> [[1 0],[0,-1],[-2,3]]
> EOF
T[1]:={{0,1},{1,2},{0,2}};
T[2]:={};
```

Note that in this example, we also could have interchanged the second and the third constraint and then have replaced j *by* $-j'$*.*

In practice, this method of computing a chamber decomposition does not seem to perform very well, mostly because `TOPCOM` can not exploit all available information about the parametric polytopes and will therefore compute many redundant triangulations/chambers. In particular, any chamber that does not intersect with the parameter domain of the parametric polytope, or only intersects in a face of this parameter domain, is completely redundant. Furthermore, if the parametric polytope is not simple, then many different combinations of the constraints will lead to the same parametric vertex. Many triangulations will therefore correspond to one and the same chamber in the chamber complex of the parametric polytope. For example, for a dilated octahedron, `TOPCOM` will compute 150 triangulations/chambers, 104 of which are empty, while the remaining 46 refer to the same single chamber.

5.19 Computing the Hilbert basis of a cone

To compute the Hilbert basis of a cone, we use the `zsolve` library from `4ti2` (Hemmecke et al.), which implements the technique of Hemmecke (2002). We first remove all equalities from the cone through unimodular transformations and then apply the technique of subsection 5.17 to put the cone in "standard form". Note that for a (non-parametric) cone the constant term $\mathbf{b}$ in (5.57) is $\mathbf{0}$. The constraints $D\mathbf{x} \geq \mathbf{c} = \mathbf{0}$ of (5.58) are therefore equivalent to $\mathbf{x} \geq \mathbf{0}$.

5.20 Integer Feasibility

For testing whether a polytope $P \subset \mathbb{Q}^d$ contains any integer points, we use the technique of Cook et al. (1993), based on generalized basis reduction.

The technique basically looks for a "short vector" $\mathbf{c}$ in the lattice $\mathbb{Z}^d$, where shortness is measured in terms of the width of the polytope P along that direction,

$$\begin{aligned} \operatorname{width}_{\mathbf{c}} P &= \max\{\, \langle \mathbf{c}, \mathbf{x} \rangle \mid \mathbf{x} \in P \,\} - \min\{\, \langle \mathbf{c}, \mathbf{x} \rangle \mid \mathbf{x} \in P \,\} \\ &= \max\{\, \langle \mathbf{c}, \mathbf{x} - \mathbf{y} \rangle \mid \mathbf{x}, \mathbf{y} \in P \,\}. \end{aligned}$$

The *lattice width* is the minimum width over all non-zero integer directions:

$$\operatorname{width} P = \min_{\mathbf{c} \in \mathbb{Z}^d \setminus \{\mathbf{0}\}} \operatorname{width}_{\mathbf{c}} P.$$

If the dimension d is fixed then the lattice width of any polytope $P \subset \mathbb{Q}^d$ containing no integer points is bounded by a constant (Lagarias et al. 1990; Barvinok 2002; Banaszczyk et al. 1999). If we slice the polytope using hyperplanes orthogonal to a short direction, i.e., a direction where the width is small, we will therefore only need to inspect "few" of them before either finding one with an integer point, or running out of hyperplanes, meaning that the polytope did not contain any integer points. Each slice is checked for integer points by applying the above method recursively.

A nice feature of this technique is that it will not only tell you if there is any integer point in the given polytope, but it will actually compute one if there is any.

The short vector is obtained as the first vector of a "reduced basis" of the lattice $\mathbb{Z}^d$ with respect to the polytope. In particular, the first vector $\mathbf{b}_1$ of this reduced basis will satisfy

$$\operatorname{width}_{\mathbf{b}_1} P \le \frac{\operatorname{width} P}{\left(\frac{1}{2} - \varepsilon\right)^{d-1}},$$

with $0 < \varepsilon < 1/2$ a fixed constant. That is, the width in direction $\mathbf{b}_1$ is no more than a constant factor bigger than the lattice width. See (Cook et al. 1993) for details. In our implementation we use $\varepsilon = 1/4$. When used in the above integer feasibility testing algorithm, we will also terminate the reduced basis computation as soon as the width along the first basis vector is smaller than 2. This means that there will be at most 2 slices orthogonal to the chosen direction.

The computation of the above reduced basis requires the solution of many linear programs, for which we use any of the following external solvers:

- `GLPK` (Makhorin 2006)

  This solver is based on double precision floating point arithmetic and may therefore not be suitable if the coefficients of the constraints describing the polytope are large.

- `cdd` (Fukuda 1993)

  This solver is based on exact integer arithmetic. Note that you need version `cddlib 0.94k` or newer.

The LP solver to use can be selected with the `--gbr` option.

5.21 Computing the integer hull of a polyhedron

For computing the integer hull of a polyhedron, we first describe how to compute the convex hull of a set given as an oracle for optimizing a linear objective function over the set and then we explain how to optimize a linear objective function over the integer points of a polyhedron. Applying the first with the second as optimization oracle yields a method for computing the requested integer hull.

5.21.1 Computing the convex hull based on an optimization oracle

The algorithm described below is presented by Cook et al. (1992, Remark 2.5) as an extension of the algorithm by Edmonds et al. (1982, Section 3) for computing the *dimension* of a polytope for which only an optimization oracle is available. The algorithm is described in a bit more detail by Eisenbrand (2000) and reportedly stems from Hartmann (1989). Essentially the same algorithm has also been implemented by Huggins (2006), citing beneath/beyond (Preparata and Shamos 1985) as his inspiration.

The algorithm start out from an initial set of points from the set S. After computing the convex hull of this set of points, we take one of its bounding constraints and use the optimization oracle to compute an optimal point in S (but on the other side of the bounding hyperplane) along the outer normal of this bounding constraint. If a new point is found, it is added to the set of points and a new convex hull is computed, or the old one is adapted in a beneath/beyond fashion. Otherwise, the chosen bounding constraint is also a bounding constraint of S and need not be considered anymore. The process continues until all bounding constraints in the description of the current convex hull have been considered.

In principle, the initial set of points in the above algorithm may be empty, with a “convex hull” described by a set of conflicting constraints and each equality in the description of any intermediate lower-dimensional convex hull being considered as a pair of bounding constraints with opposite outer normals. However, in our implementation, we have chosen to first compute a maximal set of affinely independent points by first taking any point from S and then adding points from S not on one of the equalities satisfied by all points found so far. This allows us to not have to worry about equalities in the main algorithm. In the case of the computation of the integer hull, finding these affinely independent points can be accomplished using the technique of subsection 5.20.

Example 5.65 *Assume we want to compute the integer hull of the polytope in the left part of Figure 5.64. We first compute a set of three affinely independent points, shown in the same part of the figure. Of the three facets of the corresponding convex hull, optimization along the outer normal (depicted by an arrow in the figure) of only one facet will yield any additional points. The other two are therefore facets of the integer hull. Optimization along the above outer normal may yield any of the points marked by a* $\circ$. *Assuming it is the bottom one, we end up with the updated convex hull in the middle of the figure. This convex hull has only one new facet. Adding the point found by optimizing over this facet's outer normal, we obtain the convex hull on the right of the figure. There are two new facets, but neither of them yields any further points. We have therefore found the integer hull of the polytope.*

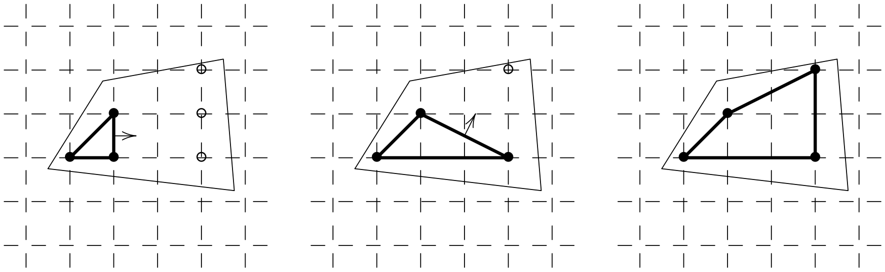


Figure 5.64: The integer hull of a polytope

5.21.2 Optimization over the integer points of a polyhedron

We assume that we want to find the *minimum* of some linear objective function. When used in the computation of the integer hull of some polytope, the objective function will therefore correspond to the inner normal of some facet.

During our search for an optimal integer point with respect to some objective function, we will keep track of the best point so far as well as a lower bound l and an upper bound u such that the value at the optimal point (if it is better than the current best) lies between those two bounds. Initially, there is no best point yet and values for l and u may be obtained from optimization over the linear relaxation. When used in the computation of the integer hull of some polytope, the upper bound u is one less than the value attained on the given facet of the current approximation.

As long as $l \le u$, we perform the following steps

- use the integer feasibility technique of subsection 5.20 to test whether there is any integer point with value in $[l, u']$, where u' is
  - u if the previous test for an integer point did not produce a point
  - $l + \left\lfloor \frac{u-l-1}{2} \right\rfloor$ if the previous test for an integer point *did* produce a point
- if a point is found, then remember it as the current best and replace u by the value at this point minus one,
- otherwise, replace l by $u' + 1$.

When used in the computation of the integer hull of some polytope, it is useful to not only keep track of the best point so far, but of all points found. These points will all lie outside of the current approximation of the integer hull and adding them all instead of just one, will typically get us to the complete integer hull quicker.

Example 5.67 *Assume that the values of some objective function attained by the integer points of some polytope are as shown in Figure 5.66 and assume we know that the optimal value lies between 1 and 16. In the first step we would look for a point attaining a value in the interval* $[1, 16]$. *Suppose this yields a point attaining the value* 8 *(second line of the figure). We record this point as the current best and update the*

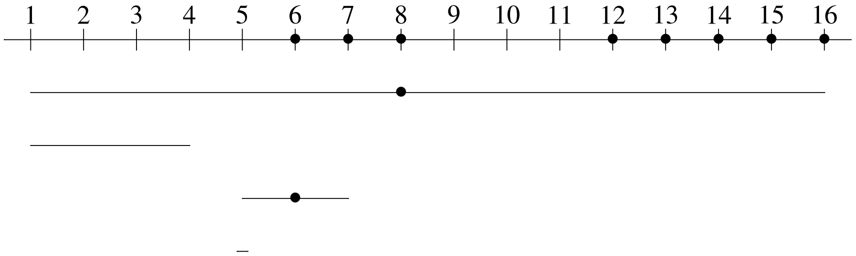


Figure 5.66: The integer points of a polytope projected on an objective function

search interval to $[1, 7]$*. In the second step, we look for a point attaining a value in the interval* $[1, 4]$*, but find nothing and set the search interval to* $[5, 7]$*. In the third step, we consider the interval* $[5, 7]$ *and find a point attaining the value 6. We update the current best value and set the search interval to* $[5, 5]$*. In the fourth step, we consider the interval* $[5, 5]$*, find no points and update the interval to "*$[6, 5]$*". Since the lower bound is now larger than the upper bound, the algorithm terminates, returning the best or all point(s) found.*

5.22 Computing the integer hull of a truncated cone

In subsection 5.23 we will need to compute the integer hull of a cone with the origin removed ($C \setminus \{\mathbf{0}\}$).

5.22.1 Using the Hilbert basis of the cone

As proposed by Köppe (2007), one way of computing this integer hull is to first compute the Hilbert basis of C (see subsection 5.19) and to then remove from that Hilbert basis the points that are not vertices of the integer hull of $C \setminus \{\mathbf{0}\}$. The Hilbert basis of C is the minimal set of points $\mathbf{b}_i \in C \cap \mathbb{Z}^d$ such that every integer point $\mathbf{x} \in C \cap \mathbb{Z}^d$ can be written as a non-negative *integer* combination of the $\mathbf{b}_i$. The vertices $\mathbf{v}_j$ of the integer hull of $C \setminus \{\mathbf{0}\}$ are such that every integer point $\mathbf{x} \in (C \cap \mathbb{Z}^d) \setminus \{\mathbf{0}\}$ can be written as s non-negative *rational* combination of $\mathbf{v}_j$. Clearly, any $\mathbf{v}_j$ is also a $\mathbf{b}_i$ since $\mathbf{v}_j$ can not be written as the sum of a (rational) convex combination of other integer points in $(C \cap \mathbb{Z}^d) \setminus \{\mathbf{0}\}$ and a non-negative combination of the extremal rays $\mathbf{r}_k$ of C. A fortiori, it can therefore not be written as an integer combination of other integer points in C. To obtain the $\mathbf{v}_j$ from the $\mathbf{b}_i$ we therefore simply need to remove first $(0, 0)$ and then those $\mathbf{b}_i$ that are not an extremal ray and that *can* be written as a combination

$$\mathbf{b}_i = \sum_{j \neq i} \boldsymbol{\alpha}_j \mathbf{b}_j + \sum_k \beta_k \mathbf{r}_k \qquad \text{with } \alpha_j, \beta_k \geq 0 \text{ and } \sum_{j \neq i} \alpha_j = 1.$$

Since the $\mathbf{r}_k$ are also among the $\mathbf{b}_j$, this can be simplified to checking whether there exists a rational solution for $\boldsymbol{\alpha}_j$ to

$$\mathbf{b}_i = \sum_{j \neq i} \boldsymbol{\alpha}_j \mathbf{b}_j \qquad \text{with } \alpha_j \geq 0 \text{ and } \sum_{j \neq i} \alpha_j \geq 1.$$

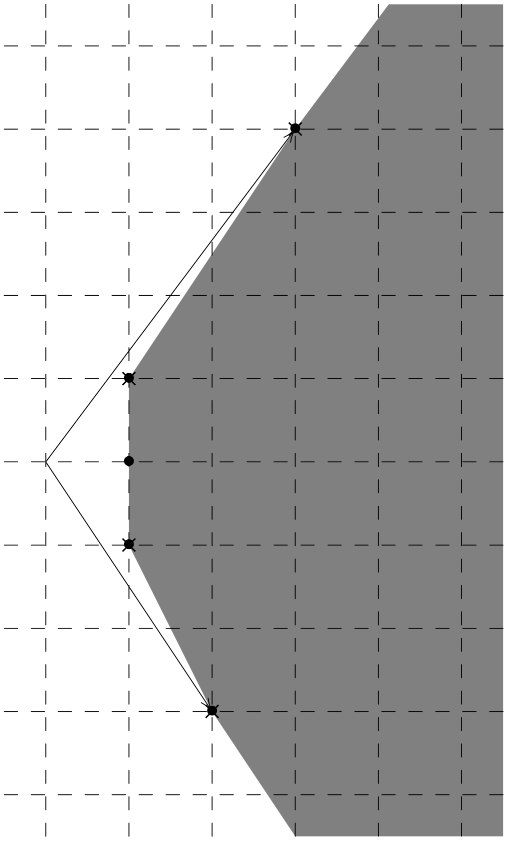


Figure 5.68: The Hilbert basis and the integer hull of a truncated cone

Example 5.69 *Consider the cone*

$$C = \text{pos}\,\{(2,-3),(3,4)\},$$

shown in Figure 5.68. The Hilbert basis of this cone is

$$\{(0,0),(2,-3),(3,4),(1,1),(1,-1),(1,0)\}.$$

We have $(1,0) = \frac{1}{2}(1,1) + \frac{1}{2}(1,-1)$*, while* $(1,1)$ *and* $(1,-1)$ *can not be written as overconvex combinations of the other* $\mathbf{b}_i \neq \mathbf{0}$*. The vertices of the integer hull of* $C \setminus \{\mathbf{0}\}$ *are therefore*

$$\{(2,-3),(3,4),(1,1),(1,-1)\}.$$

5.22.2 Using generalized basis reduction

Another way of computing the integer hull of a truncated cone is to apply the method of subsection 5.21. In this case, the initial set of points will consist of (the smallest integer representatives of) the extremal rays of the cone, together with the extremal

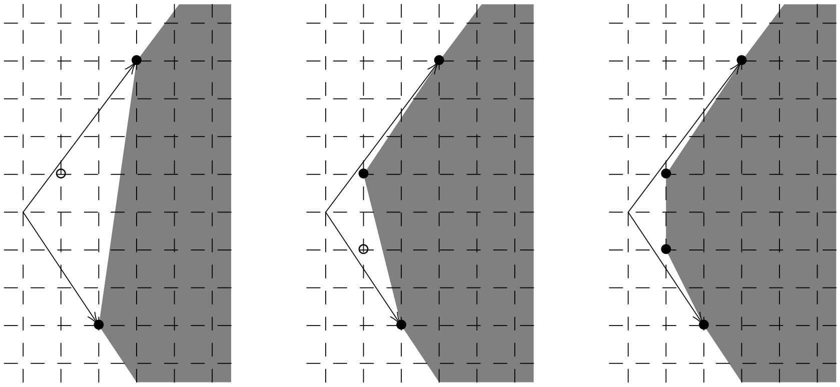


Figure 5.70: The integer hull of a truncated cone

rays themselves. That is, if $C = \operatorname{pos}\{\mathbf{r}_j\}$ with $\mathbf{r}_j \in \mathbb{Z}^d$, then our initial approximation of the integer hull of $C \setminus \{\mathbf{0}\}$ is

$$\operatorname{conv}\{\mathbf{r}_j\} + \operatorname{pos}\{\mathbf{r}_j\}.$$

Furthermore, we need never consider any of the bounding constraints that are also bounding constraints of the original cone. When optimizing along the normal of any of the other facets, we can take the lower bound to be 1. This will ensure that the origin is excluded, without excluding any other integer points.

Example 5.71 *Consider once more the cone*

$$C = \operatorname{pos}\{(2,-3),(3,4)\}$$

from Example 5.69. The initial approximation is

$$C = \operatorname{conv}\{(2,-3),(3,4)\} + \operatorname{pos}\{(2,-3),(3,4)\},$$

which is shown on the left of Figure 5.70. The only bounding constraint that does not correspond to a bounding constraint of C is $7x - y \geq 17$*. In the first step, we will therefore look for a point minimizing* $7x - y$ *with values in the interval* $[1, 16]$*. All values of this objective function in the given interval attained by points in C are shown in Figure 5.66. From Example 5.67, we know that the optimal value is* 6 *and this corresponds to the point* $(1, 1)$*. Adding this point to our hull, we obtain the approximation in the middle of Figure 5.70. This approximation has two new facets. The bounding constraint* $3x - 2y \geq 1$ *will not produce any new points since we would be looking for one in the interval "*$[1, 0]$*". The other new bounding constraint is* $4x + y \geq 5$*. Minimizing* $4x + y$ *with values in the interval* $[1, 4]$*, we find the minimal value* 3 *corresponding to the point* $(1, -1)$*. Adding this point, we obtain the complete integer hull shown on the right of Figure 5.70. Note that if in the first step we would have added not only the point corresponding to the optimal value, but instead all points found in Example 5.67, then we would have obtained the complete integer hull directly.*

5.23 Computing the lattice width of a parametric polytope

To compute the lattice width of a parametric polytope, we essentially use the technique of Eisenbrand and Shmonin (2007), which improves upon the technique of Kannan (1992). Given a parametric polytope

$$P(\mathbf{p}) = \{ \mathbf{x} \mid A\mathbf{x} + \mathbf{b}(\mathbf{p}) \geq \mathbf{0} \},$$

the width along a direction $\mathbf{c}$ is defined in the same way as for non-parametric polytopes (see subsection 5.20),

$$\operatorname{width}_{\mathbf{c}} P(\mathbf{p}) = \max\{ \langle \mathbf{c}, \mathbf{x} \rangle \mid \mathbf{x} \in P(\mathbf{p}) \} - \min\{ \langle \mathbf{c}, \mathbf{x} \rangle \mid \mathbf{x} \in P(\mathbf{p}) \}. \tag{5.72}$$

The *lattice width* is the minimum width over all non-zero integer directions:

$$\operatorname{width} P(\mathbf{p}) = \min_{\mathbf{c} \in \mathbb{Z}^d \setminus \{\mathbf{0}\}} \operatorname{width}_{\mathbf{c}} P(\mathbf{p}).$$

We assume that the parameter domain Q of $P(\mathbf{p})$, i.e., the set of parameter values for which $P(\mathbf{p}) \neq \emptyset$, is full-dimensional and that for each $\mathbf{p}$ from the interior of Q, $P(\mathbf{p})$ is also full-dimensional.

Clearly, for any given direction $\mathbf{c}$, the minimum and maximum in (5.72) are attained at (different) vertices of $P(\mathbf{p})$. The idea of the algorithm is then to consider all pairs of parametric vertices of $P(\mathbf{p})$, to compute all candidate integer directions for a given pair of vertices and then to compute the minimum width over all candidate integer directions found.

For any given parametric vertex $\mathbf{v}(\mathbf{p})$, the (rational) directions for which this vertex is minimal can be found as follows. Let $\mathbf{v}(\mathbf{p}) + C$ be the vertex cone of $\mathbf{v}(\mathbf{p})$. If $\mathbf{v}(\mathbf{p})$ is minimal for $\mathbf{c}$, then all other points in the vertex cone must yield a bigger or equal value, i.e., $\langle \mathbf{y}, \mathbf{c} \rangle \geq 0$ for all $\mathbf{y} \in C$. The set of directions is therefore the polar cone C^* of C. Note that, in principle, we should only do this for pairs of vertices that have a common activity domain, where the activity domains have been partially opened using the technique of Theorem 5.12 to avoid multiple vertices that coincide on a lower-dimensional chamber to all be considered on this intersection. However, this optimization has currently not been implemented.

Given a pair of vertices $\mathbf{v}_1(\mathbf{p})$ and $\mathbf{v}_2(\mathbf{p})$, we may assume that $\mathbf{v}_1(\mathbf{p})$ attains the minimum and $\mathbf{v}_2(\mathbf{p})$ attains the maximum. If $\mathbf{v}_1(\mathbf{p})+C_1$ and $\mathbf{v}_2(\mathbf{p})+C_2$ are the corresponding vertex cones, then the set of (rational) directions for this pair of vertices is

$$C_{1,2} = (C_1^* \cap -C_2^*) \setminus \{\mathbf{0}\}.$$

The set of candidate integer directions are therefore the vertices of the integer hull of $C_{1,2}$, which can be computed as explained in subsection 5.22. To see this, note that by construction $\langle \mathbf{c}, \mathbf{v}_1(\mathbf{p}) \rangle \leq \langle \mathbf{c}, \mathbf{v}_2(\mathbf{p}) \rangle$ and so

$$w_{\mathbf{c}}(\mathbf{p}) = \operatorname{width}_{\mathbf{c}} P(\mathbf{p}) = \langle \mathbf{c}, \mathbf{v}_2(\mathbf{p}) - \mathbf{v}_1(\mathbf{p}) \rangle \geq 0.$$

Any integer direction in $C_{1,2}$ will therefore yield a width that is at least as large as that of one of the vertices of the integer hull. Note that when using generalized basis

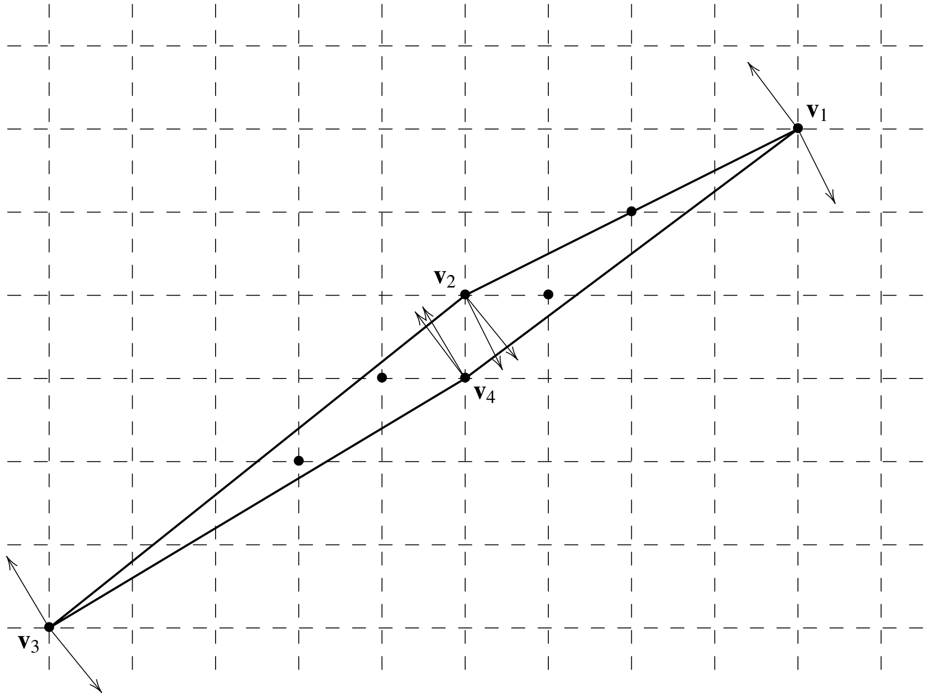


Figure 5.73: A polytope and its candidate width directions

reduction to compute the integer hull of these cones as in subsubsection 5.22.2, it can be helpful to use as vertices for the initial approximation not only the extremal rays of the cone, but also those vertices of previously computed integer hulls that are elements of the current cone.

After computing a list of all possible candidate width directions $\mathbf{c}_i$ and the corresponding widths $w_{\mathbf{c}_i}(\mathbf{p})$, we keep only a single direction of all those that yield the same width (as an affine function of the parameters). Then we construct the chambers where each of the widths is minimal, i.e.,

$$C_i = \{\, \mathbf{p} \in Q \mid \forall j : w_{\mathbf{c}_i}(\mathbf{p}) \le w_{\mathbf{c}_j}(\mathbf{p}) \,\}.$$

Note that many of the C_i may be empty or of lower dimension than Q and that the other C_i will intersect in common facets. To obtain a partition of partially-open full-dimensional chambers, we proceed as in subsection 5.4.

Example 5.74 *Consider the (non-parametric) polytope*

$$P = \left\{ \mathbf{x} \mid \begin{array}{r} -3x_1 + 5x_2 \ge 0 \\ 4x_1 - 5x_2 \ge 0 \\ x_1 - 2x_2 + 3 \ge 0 \\ -3x_1 + 4x_2 + 3 \ge 0 \end{array} \right\}$$

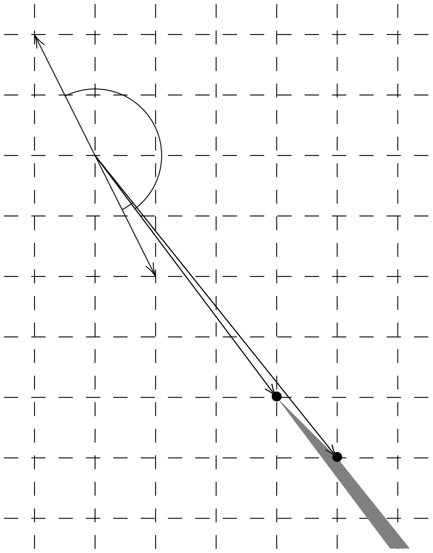


Figure 5.75: The cone of directions $C_{2,1}$

shown in Figure 5.73. The polytope has four vertices

$$\begin{aligned}\mathbf{v}_1 &= (9,6)\\ \mathbf{v}_2 &= (5,4)\\ \mathbf{v}_3 &= (0,0)\\ \mathbf{v}_4 &= (5,3).\end{aligned}$$

The corresponding cones of directions (for the given vertex to attain the minimum), also shown in Figure 5.73 are

$$\begin{aligned}C_1^* &= \operatorname{pos}\{(-3,4),(1,-2)\}\\ C_2^* &= \operatorname{pos}\{(4,-5),(1,-2)\}\\ C_3^* &= \operatorname{pos}\{(4,-5),(-3,5)\}\\ C_4^* &= \operatorname{pos}\{(-3,5),(-3,4)\}.\end{aligned}$$

Let us now consider the directions in which $\mathbf{v}_2$ *is minimal while* $\mathbf{v}_1$ *is maximal. We find*

$$C_{2,1} = \operatorname{pos}\{(4,-5),(3,-4)\} \setminus \{\mathbf{0}\},$$

as shown in Figure 5.75. The vertices of the integer hull of $C_{2,1}$ *are* $(4,-5)$ *and* $(3,-4)$. *The corresponding widths are*

$$\begin{aligned}\mathbf{c}_1 &= (4,-5) \quad w_{\mathbf{c}_1} = 6\\ \mathbf{c}_2 &= (3,-4) \quad w_{\mathbf{c}_2} = 4.\end{aligned}$$

We similarly find

$$C_{3,1} = \operatorname{pos}\{(4,-5),(-1,2)\} \setminus \{\mathbf{0}\},$$

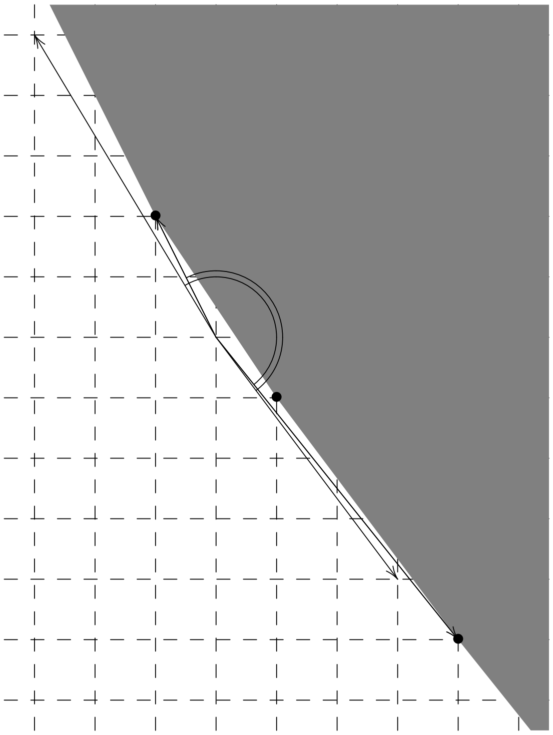


Figure 5.76: The cone of directions $C_{3,1}$

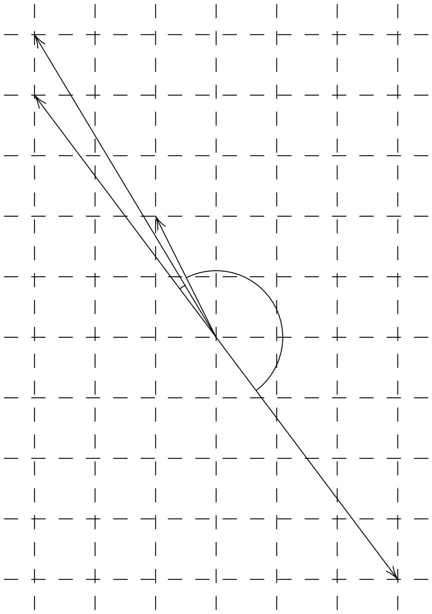


Figure 5.77: The cone of directions $C_{4,1}$

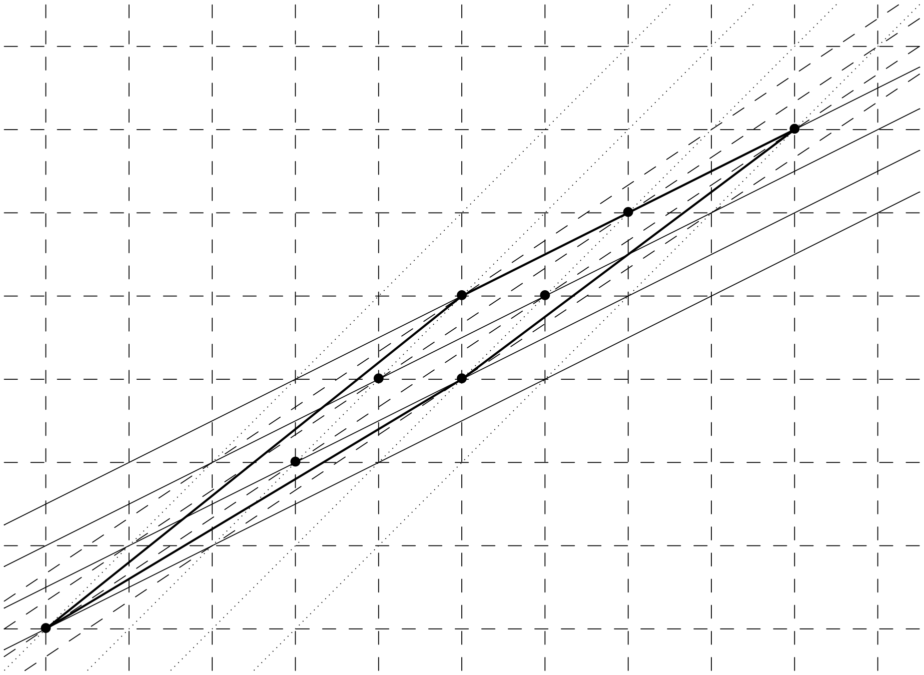


Figure 5.78: A polytope and its lattice width directions

with integer hull pos $\{(4,-5),(-1,2),(1,-1)\}$, *shown in Figure 5.76, yielding*

$$\begin{aligned} \mathbf{c}_3 &= (4,-5) & w_{\mathbf{c}_3} &= 6 \\ \mathbf{c}_4 &= (-1,2) & w_{\mathbf{c}_4} &= 3 \\ \mathbf{c}_5 &= (1,-1) & w_{\mathbf{c}_5} &= 3. \end{aligned}$$

On the other hand,

$$C_{4,1} = \emptyset,$$

as shown in Figure 5.77 and so this combination does not yield any width direction candidates. The other pairs of vertices further yield

$$\begin{aligned} \mathbf{c}_6 &= (-1,2) & w_{\mathbf{c}_6} &= 3 \\ \mathbf{c}_7 &= (-3,5) & w_{\mathbf{c}_7} &= 5 \\ \mathbf{c}_8 &= (-3,4) & w_{\mathbf{c}_8} &= 4 \\ \mathbf{c}_9 &= (-3,5) & w_{\mathbf{c}_9} &= 5 \\ \mathbf{c}_{10} &= (-2,3) & w_{\mathbf{c}_{10}} &= 3. \end{aligned}$$

Since the polytope under consideration is not parametric, there is only one (non-empty, 0*-dimensional) chamber and it corresponds to one of the directions, say* $\mathbf{c}_4 = (-1,2)$, *with width* 3 *(the other directions with the same width having been removed).*

Each of the three directions that yield the minimal width of 3 is shown in Figure 5.78.

Example 5.79 *Consider the polytope*

$$P(p) = \left\{ \mathbf{x} \mid \begin{array}{r} -2x_1 + p + 5 \geq 0 \\ 2x_1 + p + 5 \geq 0 \\ -2x_2 - p + 5 \geq 0 \\ 2x_2 - p + 5 \geq 0 \end{array} \right\}$$

from Woods (2004, Example 2.1.7). The parametric vertices are

$$\begin{aligned} \mathbf{v}_1(p) &= \left(\frac{p+5}{2}, \frac{-p+5}{2} \right) \\ \mathbf{v}_2(p) &= \left(\frac{p+5}{2}, \frac{p-5}{2} \right) \\ \mathbf{v}_3(p) &= \left(\frac{-p-5}{2}, \frac{-p+5}{2} \right) \\ \mathbf{v}_4(p) &= \left(\frac{-p-5}{2}, \frac{p-5}{2} \right). \end{aligned}$$

We find two essentially different candidate width directions

$$\begin{aligned} \mathbf{c}_1 &= (0, 1) \quad w_{\mathbf{c}_1}(p) = 5 - p \\ \mathbf{c}_2 &= (1, 0) \quad w_{\mathbf{c}_2}(p) = 5 + p. \end{aligned}$$

The first direction can be found by combining, say, $\mathbf{v}_1(p)$ *and* $\mathbf{v}_2(p)$*, while the second direction can be found by combining, say,* $\mathbf{v}_1(p)$ *and* $\mathbf{v}_3(p)$*. The parameter domain for the parametric polytope* $P(p)$ *is*

$$Q = \{ p \mid -5 \leq p \leq 5 \}.$$

The two (closed) chambers are therefore

$$\begin{aligned} C_1 &= \{ p \in Q \mid 5 - p \leq 5 + p \} \\ C_2 &= \{ p \in Q \mid 5 + p \leq 5 - p \}. \end{aligned}$$

To obtain a partition, subsection 5.1 gives the internal point $(0, 0)$*, which happens to meet the facets* $p \geq 0$ *and* $-p \geq 0$*. We therefore keep the facet with positive (inner) normal closed and open up the other facet. The result is*

$$\begin{aligned} \hat{C}_1 &= \{ p \mid 0 \leq p \leq 5 \} \\ \hat{C}_2 &= \{ p \mid -5 \leq p < 0 \}. \end{aligned}$$

Since we are usually only interested in integer parameter values, the latter chamber would become $\hat{C}_2 = \{ p \mid -5 \leq p \leq -1 \}$*.*

Our description differs slightly from that of of Eisenbrand and Shmonin (2007). First, they consider all pairs of basic solutions instead of all pairs of vertices, which means that they may consider basic solutions that are never feasible and that, in case of a non-simple polytope, they may consider the same parametric vertex more than once. The set of integer directions for a pair of vertices is the intersection of the sets of integer directions they obtain for each of the corresponding basic solutions. Second, they use a different method of creating a partition of partially-open chambers, which may lead to some lower-dimensional chambers surviving and hence to a larger total number of chambers.

5.24 Testing whether a set has an infinite number of points

In some situations we are given the generating function of some integer set and we would like to know if the set is infinite or not. Typically, we want to know if the set is empty or not, but we cannot simply count the number of elements in the standard way since we may not have any guarantee that the set has only a finite number of elements. We will consider the slightly more general case where we are given a rational generating function $f(\mathbf{x})$ of the form (5.26) such that

$$f(\mathbf{x}) = \sum_{\mathbf{s} \in Q \cap \mathbb{Z}^d} c(\mathbf{s})\, \mathbf{x}^{\mathbf{s}} \tag{5.80}$$

converges on some nonempty open subset of $\mathbb{C}^d$, Q is a pointed polyhedron and $c(\mathbf{s}) \geq 0$, and we want to compute

$$S = \sum_{\mathbf{s} \in Q \cap \mathbb{Z}^d} c(\mathbf{s}), \tag{5.81}$$

where the sum may diverge, i.e., "$S = \infty$". The following proposition shows that we can determine S in polynomial time. For a sketch of an alternative technique, see Woods (2005, Proof of Lemma 16).

Proposition 5.82 *Fix d and k. Given a rational generating function of the form* (5.26) *with $k_i \leq k$ and a pointed polyhedron $Q \subset \mathbb{Q}^d$, then there is a polynomial time algorithm that determines for the corresponding function* $c(\mathbf{s})$ (5.80) *whether the sum* (5.81) *diverges and computes the value of S* (5.81) *if it does not.*

Proof Since Q is pointed, the series (5.80) converges on a neighborhood of $e^{\boldsymbol{\ell}} = (e^{\ell_1}, \ldots, e^{\ell_d})$ for any $\boldsymbol{\ell}$ such that $\langle \mathbf{r}_k, \boldsymbol{\ell} \rangle < 0$ for any (extremal) ray $\mathbf{r}_k$ of Q and such that $\langle \mathbf{b}_{ij}, \boldsymbol{\ell} \rangle \neq 0$ for any $\mathbf{b}_{ij}$ in (5.26). Let $\boldsymbol{\alpha} = -\boldsymbol{\ell}$ and perform the substitution $\mathbf{x} = t^{\boldsymbol{\alpha}}$. The function $g(t) = f(t^{\boldsymbol{\alpha}})$ is then also a (short) rational generating function and

$$g(t) = \sum_{k \in \langle \boldsymbol{\alpha}, Q \rangle \cap \mathbb{Z}} \left(\sum_{\substack{\mathbf{s} \in Q \cap \mathbb{Z}^d \\ \langle \boldsymbol{\alpha}, \mathbf{s} \rangle = k}} c(\mathbf{s}) \right) t^k =: \sum_{k \in \langle \boldsymbol{\alpha}, Q \rangle \cap \mathbb{Z}} d(k)\, t^k,$$

with $\langle \boldsymbol{\alpha}, Q \rangle = \{ \langle \boldsymbol{\alpha}, \mathbf{x} \rangle \mid \mathbf{x} \in Q \}$, converges in a neighborhood of e^{-1}, while

$$S = \sum_{k \in \langle \boldsymbol{\alpha}, Q \rangle \cap \mathbb{Z}} d(k).$$

Since $c(\mathbf{s}) \geq 0$, we have $d(k) \geq 0$ and the above sum diverges iff any of the coefficients of the negative powers of t in the Laurent expansion of $g(t)$ is non-zero. If the sum converges, then the sum is simply the coefficient of the constant term in this expansion.

It only remains to show now that we can compute a suitable $\boldsymbol{\alpha}$ in polynomial time, i.e., an $\boldsymbol{\alpha}$ such that $\langle \mathbf{r}_k, \boldsymbol{\alpha} \rangle > 0$ for any (extremal) ray $\mathbf{r}_k$ of Q and $\langle \mathbf{b}_{ij}, \boldsymbol{\alpha} \rangle \neq 0$ for any $\mathbf{b}_{ij}$ in (5.26). By adding the $\mathbf{r}_k$ to the list of $\mathbf{b}_{ij}$ if needed, we can relax the first set of constraints to $\langle \mathbf{r}_k, \boldsymbol{\alpha} \rangle \geq 0$. Let Q be described by the constraints $A\mathbf{x} + \mathbf{c} \geq \mathbf{0}$ and let B be $d \times d$ non-singular submatrix of A, obtained by removing some of the rows of A. Such a B exists since Q does not contain any straight line. Clearly, $B\mathbf{r} \geq \mathbf{0}$ for any ray $\mathbf{r}$ of Q. Let $\mathbf{b}'_{ij} = B\mathbf{b}_{ij}$, then since $\mathbf{b}_{ij} \neq \mathbf{0}$ and B is non-singular, we have $\mathbf{b}'_{ij} \neq \mathbf{0}$. We may therefore find in polynomial time a point $\boldsymbol{\alpha}' \geq \mathbf{0}$ on the "moment curve" such that $\langle \mathbf{b}'_{ij}, \boldsymbol{\alpha}' \rangle \neq 0$ (Barvinok and Pommersheim 1999, Algorithm 5.2). Let $\boldsymbol{\alpha} = B^{\mathsf{T}}\boldsymbol{\alpha}'$. Then $\langle \mathbf{b}_{ij}, \boldsymbol{\alpha} \rangle = \langle \mathbf{b}_{ij}, B^{\mathsf{T}}\boldsymbol{\alpha}' \rangle = \langle B\mathbf{b}_{ij}, \boldsymbol{\alpha}' \rangle = \langle \mathbf{b}'_{ij}, \boldsymbol{\alpha}' \rangle \neq 0$ and $\langle \mathbf{r}_k, \boldsymbol{\alpha} \rangle = \langle \mathbf{r}_k, B^{\mathsf{T}}\boldsymbol{\alpha}' \rangle = \langle B\mathbf{r}_k, \boldsymbol{\alpha}' \rangle \geq 0$, as required. Note that in practice, we would, as usual, first try a fixed number of random vectors $\boldsymbol{\alpha}' \geq \mathbf{0}$ before resorting to looking for a point on the moment curve.

□

5.25 Enumerating integer projections of parametric polytopes

In this section we are interested in computing

$$c(\mathbf{s}) = \#\left\{ \mathbf{t} \in \mathbb{Z}^d \mid \exists \mathbf{u} \in \mathbb{Z}^m : (\mathbf{s}, \mathbf{t}, \mathbf{u}) \in P \right\}, \tag{5.83}$$

with $P \subset \mathbb{Q}^n \times \mathbb{Q}^d \times \mathbb{Q}^m$ a rational pointed polyhedron such that

$$P_{\mathbf{s}} = \left\{ (\mathbf{t}, \mathbf{u}) \in \mathbb{Q}^d \times \mathbb{Q}^m \mid (\mathbf{s}, \mathbf{t}, \mathbf{u}) \in P \right\}$$

is a polytope for any $\mathbf{s}$. This is equivalent to computing the number of points in the integer projection of a parametric polytope

$$c(\mathbf{s}) = \#(\pi(P_{\mathbf{s}} \cap \mathbb{Z}^{d+m})),$$

with $\pi : \mathbb{Q}^d \times \mathbb{Q}^m \to \mathbb{Q}^d$ defined by $\pi(\mathbf{t}, \mathbf{u}) = \mathbf{t}$. Exponential methods for computing $c(\mathbf{s})$ are described by Verdoolaege et al. (2005a) and Seghir and Loechner (2006). Here, we provide some implementation details for the polynomial method of Barvinok and Woods (2003, Theorem 1.7), for computing the generating function, $\sum_{\mathbf{s}} c(\mathbf{s})\, \mathbf{x}^{\mathbf{s}}$, which can then be converted into an explicit function $c(\mathbf{s})$ (Verdoolaege and Woods 2008, Corollary 1.11). Note that in contrast to Barvinok and Woods (2003, Theorem 1.7), we may allow P to be an unbounded (but still pointed) polyhedron here (as long as $P_{\mathbf{s}}$ is bounded), since we replace their application of Kannan (1992, Lemma 3.1) by Eisenbrand and Shmonin (2007, Theorem 5).

If there is only one existentially quantified variable ($m = 1$), then computing (5.83) is easy. You simply shift P by 1 in the u direction and subtract this shifted copy from the original,

$$D = P \setminus (\mathbf{e}_{n+d+1} + P).$$

(See, e.g., Barvinok and Woods (2003, Figure 1, page 973) or Verdoolaege (2005, Figure 4.33, page 186).) In the difference D there will be *exactly* one value of u for each value of the remaining variables for which there was *at least* one value of u in P,

$$\forall(\mathbf{s},\mathbf{t}): \quad (\exists u : (\mathbf{s},\mathbf{t},u) \in P) \iff (\exists! u : (\mathbf{s},\mathbf{t},u) \in D).$$

The function $c(\mathbf{s})$ can then be computed by counting the number of elements in $D(\mathbf{s})$. These operations can be performed either in the space of (unions of) parametric polytopes or on generating functions. In the first case, $D(\mathbf{s})$ can be written as a disjoint union of parametric polytopes that can be enumerated separately. In the second case, we first compute the generating function $f(\mathbf{x},\mathbf{y})$ of the set

$$S = \{(\mathbf{s},\mathbf{t}) \mid \exists u \in \mathbb{Z} : (\mathbf{s},\mathbf{t},u) \in P\}$$

and then obtain the generating function $C(\mathbf{x})$ of $c(\mathbf{s})$ as $C(\mathbf{x}) = f(\mathbf{x},\mathbf{1})$. In the remainder of this section, we will concentrate on the computation of the generating function of S. To compute this generating function in the current case where there is only one existentially quantified variable, we first compute the generating function $g(\mathbf{x},\mathbf{y},z)$ of $P(\mathbf{s},\mathbf{t},u)$, perform operations on the generating function equivalent to the set operations (see, e.g., Verdoolaege (2005, Section 4.5.3)), resulting in a generating function $g'(\mathbf{x},\mathbf{y},z)$, and then sum over all values (at most one for each value of $\mathbf{s}$ and $\mathbf{t}$) of u, i.e., $f(\mathbf{x},\mathbf{y}) = g'(\mathbf{c},\mathbf{y},1)$.

If there is more than one existentially quantified variable ($m > 1$), then we can in principle apply the above shifting and subtracting technique recursively to obtain a generating function $f(\mathbf{x},\mathbf{y})$ for the set

$$T = \{(\mathbf{s},\mathbf{t}) \mid \exists \mathbf{u} \in \mathbb{Z}^m : (\mathbf{s},\mathbf{t},\mathbf{u}) \in P\} \tag{5.85}$$

and then compute $C(\mathbf{x}) = f(\mathbf{x},\mathbf{1})$. There are however some complications. Most notably, after applying the technique in one direction and projecting out the corresponding variable, the resulting set, i.e.,

$$S = \{(\mathbf{s},\mathbf{t},u_1,\ldots,u_{m-1}) \mid \exists u_m \in \mathbb{Z} : (\mathbf{s},\mathbf{t},\mathbf{u}) \in P\},$$

in general does not correspond to the integer points in some polytope. For example, assume that the polytope in Figure 5.84 contains the values of $\mathbf{u}$ associated to a particular value of $(\mathbf{s},\mathbf{t})$. Since there are integer points in this polytope, we should count this value of $\mathbf{t}$, but only once. If we apply the above technique in the vertical direction (u_2), then we can compute (a generating function for) the set S shown on the bottom of the figure. Note, however, that there are "gaps" in this set, i.e., if we compute $S \setminus (\mathbf{e}_{n+d+1} + S)$ then we will not end up with a single point (for this value of $(\mathbf{s},\mathbf{t})$). Since the biggest gap is three wide, we need to compute

$$S \setminus (\mathbf{e}_{n+d+1} + S) \setminus (2\mathbf{e}_{n+d+1} + S) \setminus (3\mathbf{e}_{n+d+1} + S)$$

to obtain a single point. If we do the subtraction in the horizontal direction first, then we end up with a set (shown on the left) with gaps at most two wide, so afterwards we only need to subtract twice in the vertical direction.

In general, there is no bound on the widths of the gaps we may encounter in any given direction. However, there are directions in which the gaps are known to be

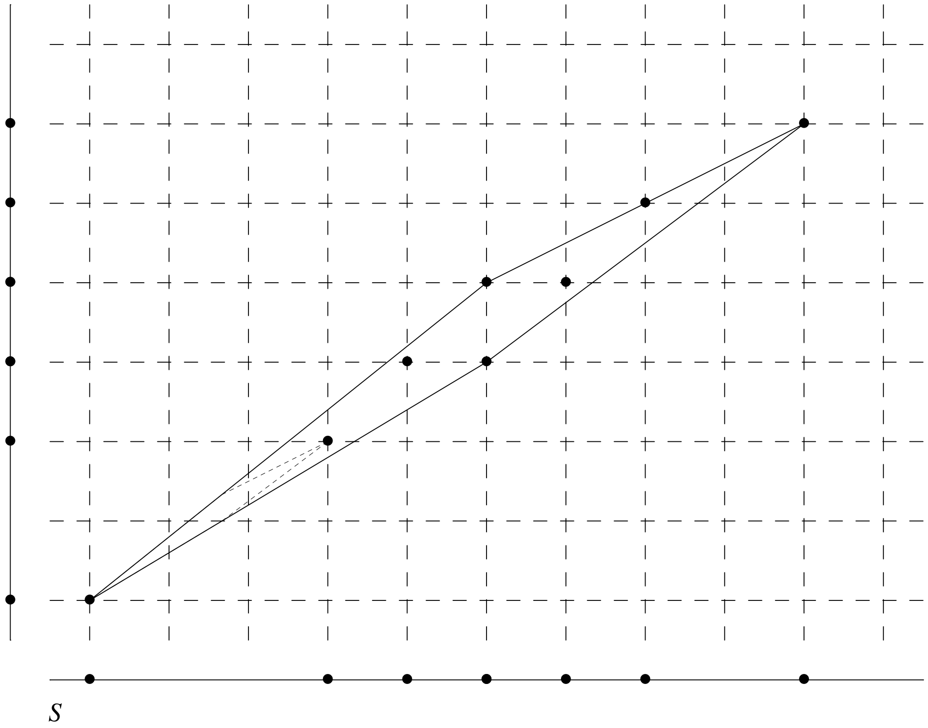


Figure 5.84: A polytope and its integer projections

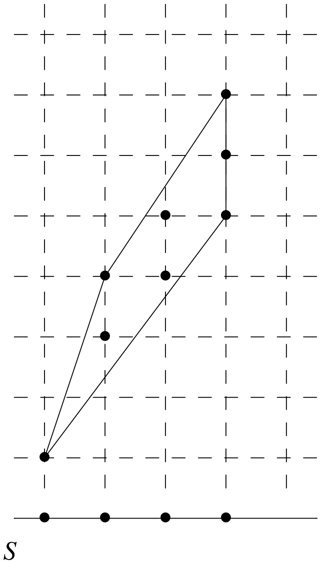


Figure 5.86: A transformed polytope and its integer projection

"small". In particular, if the dimension m is fixed, then the lattice width (see subsection 5.23) of lattice point free polytopes is bounded by a constant $\omega(m)$ (Lagarias et al. 1990; Barvinok 2002; Banaszczyk et al. 1999). This means that in the direction of the lattice width of a polytope, the gaps will be not be larger than $\omega(m)$ (Barvinok and Woods 2003, Theorem 4.3). Otherwise, we would be able to put a (uniformly) scaled down version of the polytope in the gap and it would contain no lattice points, which would contradict the fact that its lattice width is bounded by $\omega(m)$. Figure 5.84 contains such a scaled down copy of the original polytope. However, neither the horizontal nor the vertical direction is a lattice width direction of this polytope. The actual lattice width of this polytope was computed in Example 5.74 as 3 with corresponding direction $\mathbf{c} = (-1, 2)$. Figure 5.86 shows the result of applying the unimodular transformation

$$\begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix}$$

to the polytope. Note that the horizontal direction now has gaps of width at most 1. After shifting, subtracting and projecting in the vertical direction, we therefore end up with a set S with gaps of width 1 and we then only have to shift and subtract once in the remaining (horizontal) direction.

In fact, for two-dimensional polytopes the gaps in the lattice width direction will always be one, as shown by the following lemma.

Lemma 5.87 *For any rational polygon, the gaps in a lattice width direction are of width at most 1.*

Proof We may assume that x is the given lattice width direction of a given polygon P. If there is a gap of width 2, then there is an integer value x_1 of x such that $P \cap \{\, (x_1, y) \,\} \neq \emptyset$, $P \cap \{\, (x_1 + 2, y) \,\} \neq \emptyset$, while $P \cap \{\, (x_1 + 1, y) \,\} \cap \mathbb{Z}^2 = \emptyset$. Using Barvinok and Woods (2003, Lemma 4.2), we can put a scaled down copy P' of P between $x = x_1$ and $x = x_1 + 2$ (and inside of P). P' meets the line $x = x_1 + 1$ between two consecutive integer points, y_1 and $y_1 + 1$. Let P'' be the polygon bounded by $x = x_1$ and $x = x_1 + 2$ and two lines that separate P' from these two integer points. P'' will have the same width (2) in the x direction, while $P' \subset P''$. The x direction is therefore also a lattice width direction of P''. P'' cannot intersect both $x = x_1$ and $x = x_1 + 2$ in a segment of length greater than or equal to 1. Otherwise, it would also intersect $x = x_1 + 1$ in a segment of length greater than or equal to 1.

We may therefore assume that the length of the intersection of P'' with $x = x_1$ is smaller than 1. If this line segment contains an integer point, then call it y_2. Otherwise, let y_2 be the greatest integer smaller than the points in the line segment. We may assume that $y_1 = y_2$. Otherwise, we can apply the unimodular transformation

$$\begin{bmatrix} x \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ y_1 - y_2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix},$$

without changing the width in direction x. If P'' contains (x_1, y_1), it intersects $x = x_1$ in a segment $[y_1 - \alpha_1, y_1 + \alpha_2]$. We may then similarly assume that $\alpha_2 \geq \alpha_1$. P'' will only cut $x = x_1 + 2$ in points with y-coordinate smaller than $2 - \alpha_2$. The width in the y direction will therefore be smaller than $2 - \alpha_2 + \alpha_1 \leq 2$, contradicting that x is a

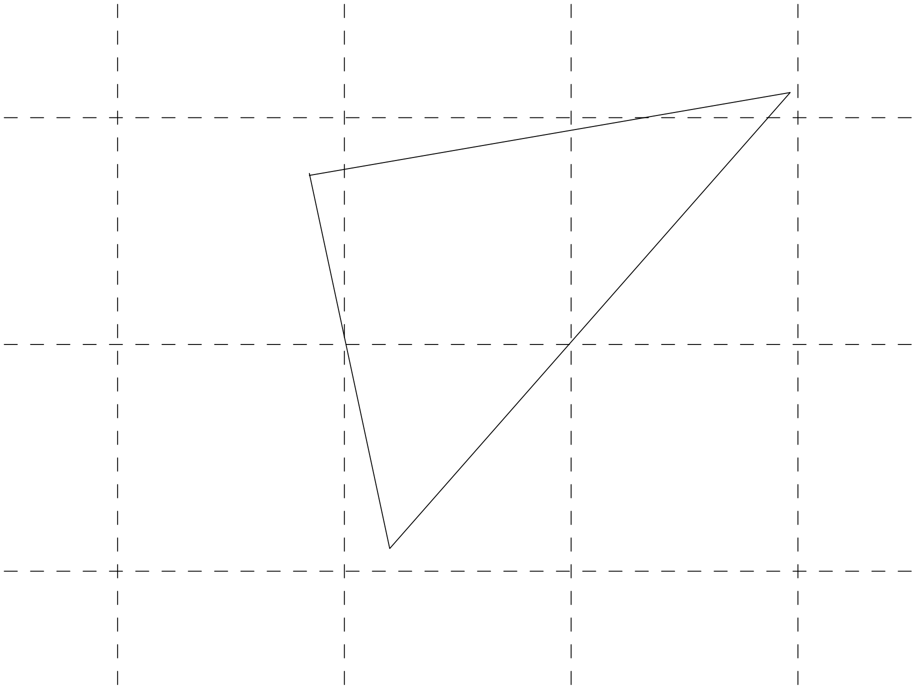


Figure 5.88: Lattice point free polygon with lattice width 2

lattice width direction. If P'' does not contain (x_1, y_1), then it only intersects $x = x_1$ in points with y-coordinate $y_1 + \alpha$ with $0 < \alpha < 1$. Given any such point, it is clear that P'' intersects $x = x_1 + 2$ only in points with y-coordinate strictly between $y_1 - \alpha$ and $y_1 + 1 - \alpha$, again showing that the width in the y direction is smaller than 2 and leading to the same contradiction. The contradiction shows that there can be no gaps of width 2 in the lattice width direction of P. □

Note that the $\omega(2)$ bound is too coarse to reach the above conclusion as $\omega(2) > 2$. An example of a polygon with lattice with greater than 2 is the polygon with vertices $(-17/110, 83/110)$, $(2/10, -9/10)$ and $(177/90, 100/90)$, shown in Figure 5.88, which has width $221/110$.

The idea of the projection algorithm is now to first find a direction in which the gaps are expected to be small and to unimodularly transform the existentially quantified variables such that this direction lies in the direction of one of the transformed variables. Then, the remaining existentially quantified variables are projected out by applying the technique recursively. The resulting generating function will have gaps at most $\omega(m)$ wide and so we have to subtract at most $\omega(m)$ shifted copies of this generating function before we can plug in 1 to project out the selected (and now only remaining) existentially quantified variable. We now look at each of these step in a bit more detail.

We are given a polyhedron P such that $P_{\mathbf{s}}$ is a polytope and we want to compute a generating function $f(\mathbf{x}, \mathbf{y})$ for the set T in (5.85). We first compute the lattice width directions of the m-dimensional parametric polytope $P_{\mathbf{s},\mathbf{t}}$ as in subsection 5.23. The result is a partition of the parameter domain, i.e., the projection of P onto the first $n + d$ coordinates, into partially open polyhedra Q_i, together with the lattice width direction $\mathbf{c}_i$ corresponding to each Q_i. Since the generating functions only encode integer points, we can replace each open facet $\langle \mathbf{a}, \mathbf{x} \rangle + b > 0$ by the closed facet $\langle \mathbf{a}, \mathbf{x} \rangle + b - 1 \geq 0$ to obtain a collection of closed polyhedra $\tilde{Q}_i$. Now let

$$P_i = P \cap \tilde{Q}_i \times \mathbb{Q}^m$$

and let $f_i(\mathbf{x}, \mathbf{y})$ be the generating function of the set

$$T_i = \{ (\mathbf{s}, \mathbf{t}) \mid \exists \mathbf{u} \in \mathbb{Z}^m : (\mathbf{s}, \mathbf{t}, \mathbf{u}) \in P_i \}.$$

Then clearly,

$$f(\mathbf{x}, \mathbf{y}) = \sum_i f_i(\mathbf{x}, \mathbf{y}).$$

From now on, we will consider a particular P_i with corresponding lattice width $\mathbf{c}_i$ and drop the i subscript.

We are now given a polyhedron P such that the lattice width direction of $P_{\mathbf{s},\mathbf{t}}$ is $\mathbf{c}$. We first extend $\mathbf{c}$ to an $m \times m$ unimodular matrix U using the technique of subsection 5.7,

$$U = \begin{bmatrix} \mathbf{c}^T \\ U' \end{bmatrix}$$

and then compute

$$P' = \begin{bmatrix} I_n & 0 & 0 \\ 0 & I_d & 0 \\ 0 & 0 & U \end{bmatrix} P.$$

We have

$$T = \{ (\mathbf{s}, \mathbf{t}) \mid \exists \mathbf{u}' \in \mathbb{Z}^m : (\mathbf{s}, \mathbf{t}, \mathbf{u}') \in P' \},$$

i.e., we may have changed the values of the existentially quantified variables, but we have not changed the set T. Now consider the set

$$T' = \{ (\mathbf{s}, \mathbf{t}, u_1') \mid \exists (u_2', \ldots, u_m') \in \mathbb{Z}^{m-1} : (\mathbf{s}, \mathbf{t}, \mathbf{u}') \in P' \}.$$

This set has only $m - 1$ existentially quantified variables, so we may apply this projection algorithm recursively and obtain the generating function $f'(\mathbf{x}, \mathbf{y}, z)$ for T'. The set T' may no longer correspond to the integer points in a polytope, but, by construction, the gaps in the final coordinate are small ($\leq \omega(m)$).

By now we have a generating function $f'(\mathbf{x}, \mathbf{y}, z)$ for the set T' (with small gaps in the final coordinate) and we have to compute the generating function $f(\mathbf{x}, \mathbf{y})$ for T. By computing

$$f''(\mathbf{x}, \mathbf{y}, z) = f'(\mathbf{x}, \mathbf{y}, z) \bigoplus_{k=1}^{\lfloor \omega(m) \rfloor} \left(z^k f'(\mathbf{x}, \mathbf{y}, z) \right), \tag{5.89}$$

where $\oplus$ represents the operation on generating functions that corresponds to set difference on the corresponding sets, we obtain a generating for the set T'' where only the smallest value of u_1' is retained. The total number of u_1's associated to any $(\mathbf{s}, \mathbf{t})$ is therefore either zero or one and so the "multiset" defined by taking as many copies of $(\mathbf{s}, \mathbf{t})$ as there are associated values of u_1' is actually the set T. That is

$$f(\mathbf{x}, \mathbf{y}) = f''(\mathbf{x}, \mathbf{y}, 1).$$

The only remaining problem is that the "$\oplus$" operation in (5.89) is fairly expensive. In particular, this operation is performed by first computing the Hadamard product of the two generating functions (which corresponds to the intersection of the sets) and then subtracting the resulting generating function from this first generating function. The last operation is fairly cheap, but the Hadamard product has a time complexity which while polynomial if the dimension (in this case the maximum of k_i in (5.26)) is fixed, is exponential in this dimension. Furthermore, this dimension increases by $\max k_i - d$ on each successive application of the Hadamard product, while $\max k_i > d$ as soon as some projection is performed, which certainly happens in the recursive application of the algorithm. Now, the total number of Hadamard products is bounded by a constant $\lfloor \omega(m) \rfloor$ and so the increase in dimension is also bounded by a constant, so the whole operation is still polynomial for fixed dimension. Nevertheless, we do not want to perform any additional Hadamard products if we do not really have to. That is, we would like to be able to detect when we can stop performing these operations *before* reaching the upper bound $\lfloor \omega(m) \rfloor$.

Let $f_0'(\mathbf{x}, \mathbf{y}, z) = f'(\mathbf{x}, \mathbf{y}, z)$ and let $f_k'(\mathbf{x}, \mathbf{y}, z)$ be the result of applying the "set difference" in (5.89) k times. Denote the corresponding sets by T_0' and T_k'. We want to find the smallest k such that $f''(\mathbf{x}, \mathbf{y}, z) = f_k'(\mathbf{x}, \mathbf{y}, z)$. Note that we are talking about equality of rational functions here. Any further application of the set difference operation will not change this rational function, but it *will* typically produce a different (more complex) representation. To check whether the current k is sufficient, we are going to count how many times any element of T_k' still appears in a shifted copy of T_0', with shift greater than or equal to $k + 1$. If this number is zero, any further set difference will have no effect. That is, we want to compute

$$\sum_{l=k+1}^{\infty} \left| T_l' \cap (\mathbf{e}_{n+d+1} + T') \right|,$$

or, in terms of generating functions,

$$h(\mathbf{x}, \mathbf{y}, z) = \sum_{l=k+1}^{\infty} f_k'(\mathbf{x}, \mathbf{y}, z) \star z^l f'(\mathbf{x}, \mathbf{y}, z).$$

We should point out here that while the Hadamard product ($\star$) corresponds to intersection when applied to generator functions of indicator functions (i.e., with coefficients one or zero), in general it will produce a generating function with coefficients that are the product of the corresponding coefficients in the two operands. We can therefore rewrite the above equation as

$$\begin{aligned}
h(\mathbf{x}, \mathbf{y}, z) &= \sum_{l=k+1}^{\infty} f'_k(\mathbf{x}, \mathbf{y}, z) \star z^l f'(\mathbf{x}, \mathbf{y}, z) \\
&= f'_k(\mathbf{x}, \mathbf{y}, z) \star \left(\sum_{l=k+1}^{\infty} z^l f'(\mathbf{x}, \mathbf{y}, z) \right) \\
&= f'_k(\mathbf{x}, \mathbf{y}, z) \star \frac{z^{k+1} f'(\mathbf{x}, \mathbf{y}, z)}{1 - z}.
\end{aligned}$$

Computing $h(\mathbf{x}, \mathbf{y}, 1)$ would give us a generating function with as coefficients how many times a point of T'_k still appears in a shifted copy of T'_0 for any particular value of $\mathbf{s}$ and $\mathbf{t}$. However, we want to know if this number is zero for *all* values of $\mathbf{s}$ and $\mathbf{t}$, so we compute $h(\mathbf{1}, \mathbf{1}, 1)$ instead. We have to be careful here since we allow the polyhedron P to be unbounded and so we should apply the technique of subsection 5.24 with Q the projection of P on the remaining coordinates.

Note that testing whether we can stop is more expensive than applying the next iteration (since we have an extra $(1 - z)$ factor in the denominator of one of the operands). However, we may save many iterations by stopping early and we will not needlessly replace a given representation of $f''(\mathbf{x}, \mathbf{y}, z)$ by a more complex representation (with more factors in the denominator). An alternative way of checking whether we can stop is to test whether $f'_k(\mathbf{x}, \mathbf{y}, z) = f'_{k+1}(\mathbf{x}, \mathbf{y}, z)$ (as rational functions). To do so, we would need to check that both

$$f'_k(\mathbf{x}, \mathbf{y}, z) - \left(f'_k(\mathbf{x}, \mathbf{y}, z) \star f'_{k+1}(\mathbf{x}, \mathbf{y}, z) \right)$$

and

$$f'_{k+1}(\mathbf{x}, \mathbf{y}, z) - \left(f'_k(\mathbf{x}, \mathbf{y}, z) \star f'_{k+1}(\mathbf{x}, \mathbf{y}, z) \right)$$

are zero and this Hadamard product is more expensive than the one above.

Example 5.91 *Consider once more the parametric polytope*

$$P(p) = \left\{ \mathbf{x} \mid \begin{array}{r} -2x_1 + p + 5 \geq 0 \\ 2x_1 + p + 5 \geq 0 \\ -2x_2 - p + 5 \geq 0 \\ 2x_2 - p + 5 \geq 0 \end{array} \right\}$$

from Woods (2004, Example 2.1.7) and Example 5.79 and assume we want to compute

$$c(p) = \left[\exists \mathbf{x} \in \mathbb{Z}^2 : (p, \mathbf{x}) \in P \right].$$

That is, we simply want to know for which values of p the polytope is non-empty. Now, there are more efficient ways of answering this particular question, but we will use it here as an example of the above algorithm. The polytope $P(p)$ is shown in Figure 5.90 for all integer value of the parameter for which the polytope is non-empty.

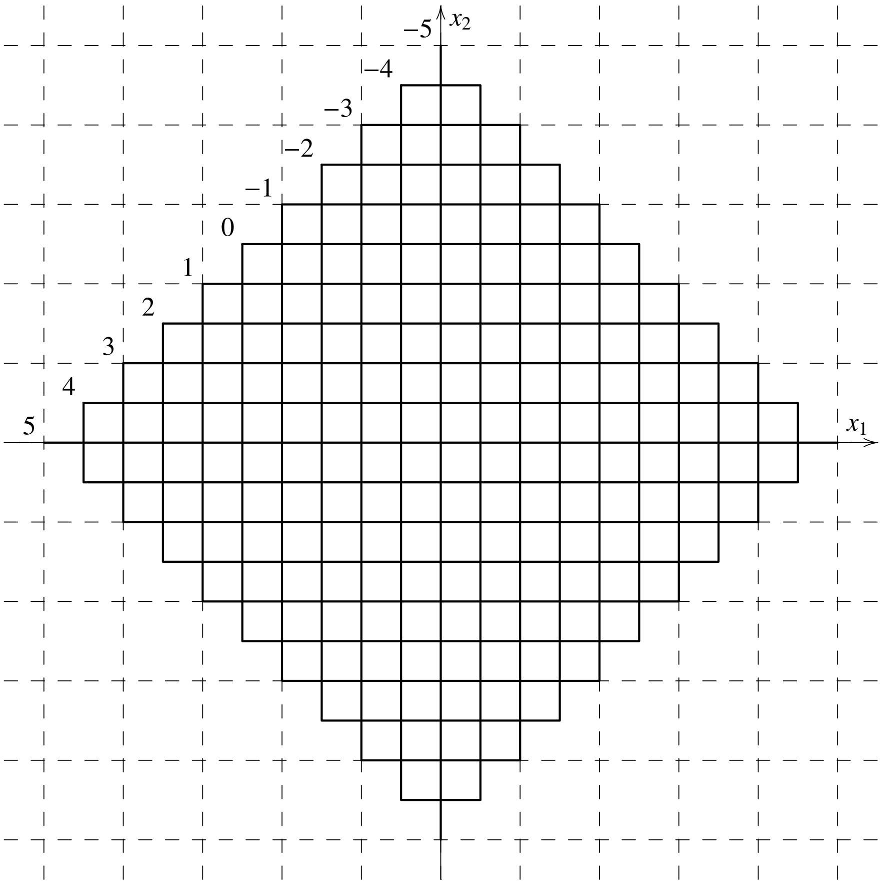


Figure 5.90: The parametric polytope from Example 5.91 for different values of the parameter

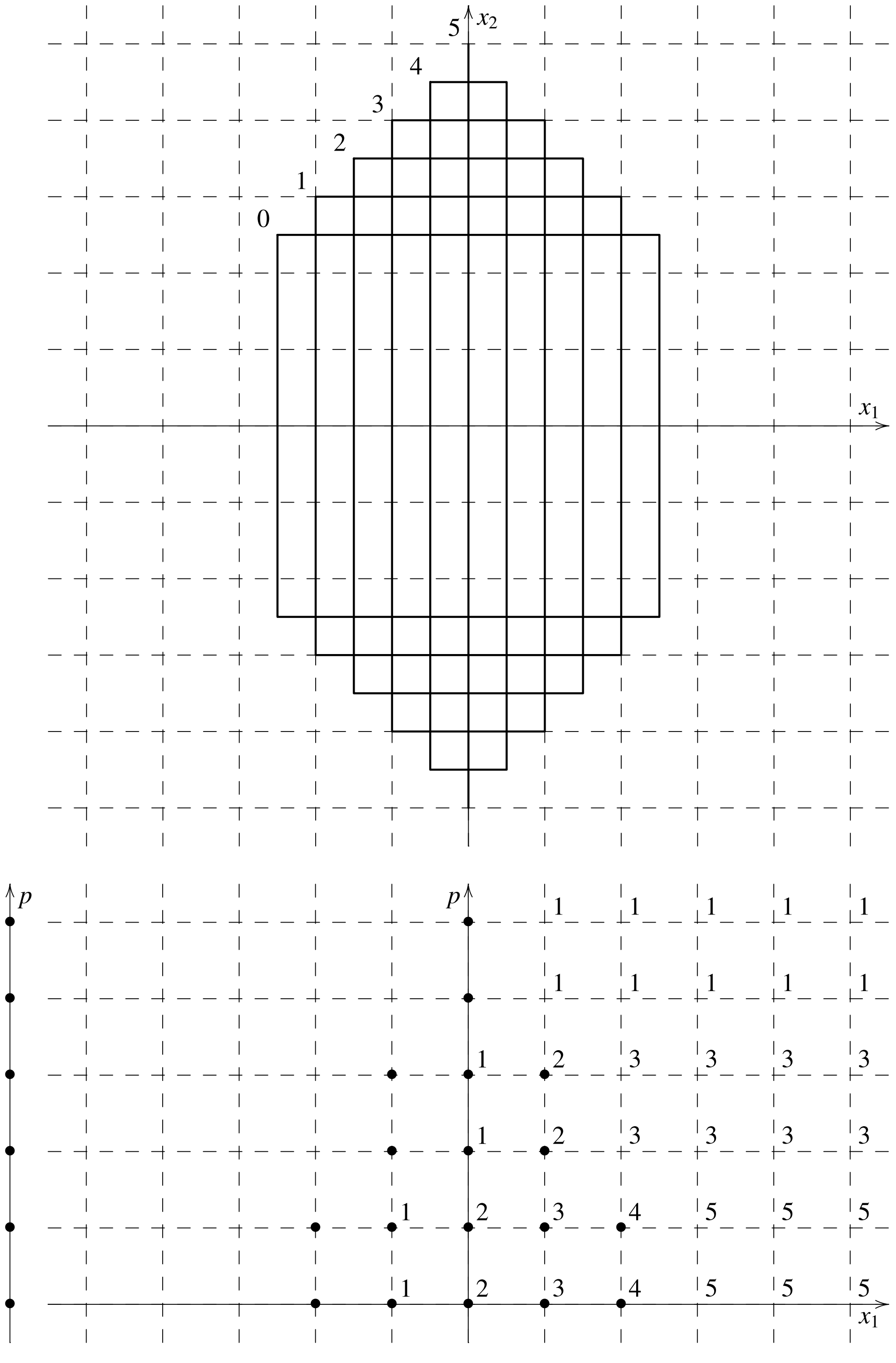


Figure 5.92: The transformed parametric polytope from Example 5.91 for $0 \leq p \leq 5$

The first step is to compute the lattice width of $P(p)$*. In Example 5.79, we already obtained the decomposition of the parameter domain into*

$$\begin{aligned}\hat{C}_1 &= \{\, p \mid 0 \le p \le 5 \,\}\\ \hat{C}_2 &= \{\, p \mid -5 \le p \le -1 \,\},\end{aligned}$$

with corresponding lattice widths and lattice width directions

$$\begin{aligned}\mathbf{c}_1 &= (0,1) \quad w_{\mathbf{c}_1}(p) = 5 - p\\ \mathbf{c}_2 &= (1,0) \quad w_{\mathbf{c}_2}(p) = 5 + p.\end{aligned}$$

Note that in this example, the gaps in both coordinate directions are 1*, so, in principle, there is no need to perform any unimodular transformation here. Nevertheless, we will apply the transformation that would be applied by the algorithm. On the first domain, we extend the lattice width direction to the unimodular matrix*

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

After application to the existentially quantified variables $\mathbf{x}$*, we obtain the parametric polytope*

$$P_1'(p) = \left\{ \mathbf{x} \mid \begin{array}{r} -2x_2 + p + 5 \ge 0 \\ 2x_2 + p + 5 \ge 0 \\ -2x_1 - p + 5 \ge 0 \\ 2x_1 - p + 5 \ge 0 \\ p \ge 0 \end{array} \right\}$$

shown in the top part of Figure 5.92. We now temporarily remove the existential quantification on x_1*, resulting in*

$$T' = \{(p, x_1) \in \mathbb{Z}^2 \mid \exists x_2 \in \mathbb{Z} : (s, \mathbf{x}) \in P'\}.$$

Since there is only one existentially quantified variable left, we know we only have to shift and subtract the set once to obtain a set with at most one value of x_2 *associated to each value of* (p, x_1)*. In particular, let* $f(x, z_1, z_2)$ *be the generating function of the integer points in* P'*. Then* $g(x, z_1) = f'(x, z_1, 1)$*, with* $f'(x, z_1, z_2) = f(x, z_1, z_2) - f(x, z_1, z_2) \star z_2 f(x, z_1, z_2)$*, is the generating function of* T'*.*

To check whether we need to subtract any shifted copies of $g(x, z_1)$ *from itself, we compute*

$$h(x, z_1) = g(x, z_1) \star \frac{z_1\, g(x, z_1)}{1 - z_1}.$$

The second argument of this Hadamard product is depicted in Figure 5.92 by its coefficients. The exponents in $h(x, z_1)$ *that have a non-zero coefficient are therefore those marked by both a dot (•) and a number. The total sum can be computed as* $h(1, 1) = 26$*, which is finite, but non-zero. We therefore need to subtract at least one shifted copy of* $g(x, z_1)$*. Let*

$$g'(x, z_1) = g(x, z_1) - g(x, z_1) \star z_1 g(x, z_1).$$

Computing

$$h'(x, z_1) = g'(x, z_1) \star \frac{z_1^2\, g(x, z_1)}{1 - z_1},$$

we would find that $h'(1, 1) = 0$ *and so we do not need to shift and subtract any further. However, since we are dealing with a two-dimensional problem, we already know from Theorem 5.87 that we can stop after one shift and subtract, so we do not even need to compute* $h'(x, z_1)$ *here. The function* $g'(x, z_1)$ *now only has non-zero coefficients for at most one exponent of* z_1 *for each exponent of* x*. We may therefore project down to obtain the function* $g'(x, 1)$*, which is the generating function of the set in the lower left part of Figure 5.92.*

For the chamber $\hat{C}_2$ *of the parameter domain, the computations are nearly identical and the final result is simply the sum of the two generating functions found for each of the two (disjoint) chambers.*

6 Publications

6.1 Publications about the Library

This is a list of some reports and publications explaining details of parts of the `barvinok` library.

- Analytical computation of Ehrhart polynomials and its applications for embedded systems (Verdoolaege, Beyls, Bruynooghe, Seghir, and Loechner; 2004b)
- Analytical computation of Ehrhart polynomials and its applications for embedded systems (Verdoolaege, Beyls, Bruynooghe, Seghir, and Loechner; 2004c)
- Analytical Computation of Ehrhart Polynomials and its Application in Compile–Time Generated Cache Hints (Seghir, Verdoolaege, Beyls, and Loechner; 2004)
- Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations (Verdoolaege, Seghir, Beyls, Loechner, and Bruynooghe; 2004d)
- Experiences with enumeration of integer projections of parametric polytopes (Verdoolaege, Beyls, Bruynooghe, and Catthoor; 2004a)
- Experiences with enumeration of integer projections of parametric polytopes (Verdoolaege, Beyls, Bruynooghe, and Catthoor; 2005a)
- Computation and Manipulation of Enumerators of Integer Projections of Parametric Polytopes (Verdoolaege, Woods, Bruynooghe, and Cools; 2005b)
- Incremental Loop Transformations and Enumeration of Parametric Sets (Verdoolaege; 2005)
- Symbolic Polynomial Maximization over Convex Sets and its Application to Memory Requirement Estimation (Clauss, Fernández, Garbervetsky, and Verdoolaege; 2006)
- Counting with rational generating functions (Verdoolaege and Woods; 2008)
- Counting integer points in parametric polytopes using Barvinok's rational functions (Verdoolaege, Seghir, Beyls, Loechner, and Bruynooghe; 2007b)
- Polynomial Approximations in the Polytope Model: Bringing the Power of Quasi-Polynomials to the Masses (Meister and Verdoolaege; 2008)
- Bounds on Quasi-Polynomials for Static Program Analysis (Devos, Verdoolaege, Van Campenhout, and Stroobandt; 2007)
- Computing parametric rational generating functions with a primal Barvinok algorithm (Köppe and Verdoolaege; 2008)
- An Implementation of the Barvinok–Woods Integer Projection Algorithm (Köppe, Verdoolaege, and Woods; 2008)

- Algorithms for Weighted Counting over Parametric Polytopes: A Survey and a Practical Comparison (Verdoolaege and Bruynooghe; 2008)

6.2 Publications Refering to the Library

This is a list of some reports and publications refering to the `barvinok` library.

- Formulas of Brion, Lawrence, and Varchenko on rational generating functions for cones (Beck, Haase, and Sottile; 2009)
- Generating Cache Hints for Improved Program Efficiency (Beyls and D'Hollander; 2005)
- An alternative algorithm for counting lattice points in a convex polytope (Lasserre and Zeron; 2005)
- Volume Calculation and Estimation of Parameterized Integer Polytopes (Rabl; 2006)
- Improved Derivation of Process Networks (Verdoolaege, Nikolov, and Stefanov; 2006)
- Computing the Ehrhart quasi-polynomial of a rational simplex (Barvinok; 2006)
- Memory Optimization by Counting Points in Integer Transformations of Parametric Polytopes (Seghir and Loechner; 2006)
- GRAPHITE: Polyhedral Analyses and Optimizations for GCC (Pop, Silber, Cohen, Bastoul, Girbal, and Vasilache; 2006)
- Volume Computation for Polytopes and Partition Functions for Classical Root Systems. (Baldoni-Silva, Beck, Cochet, and Vergne; 2006)
- A primal Barvinok algorithm based on irrational decompositions (Köppe; 2007)
- pn: A Tool for Improved Derivation of Process Networks (Verdoolaege, Nikolov, and Stefanov; 2007a)
- Theoretical and Computational Methods for Lattice Point Enumeration in Inside-Out Polytopes (van Herick; 2007)
- On Ehrhart Polynomials and Probability Calculations in Voting Theory (Lepelley, Louichi, and Smaoui; 2008)
- Local Euler-Maclaurin formula for polytopes (Berline and Vergne; 2006)
- Exact algorithms and software in optimization and polyhedral computation (Fukuda; 2008)
- Enumeration of 4×4 magic squares (Beck and van Herick; 2010)

- Symbolic and Analytic Techniques for Resource Analysis of Java Bytecode (Aspinall, Atkey, MacKenzie, and Sannella; 2010)
- Parametric Integer Programming Algorithm for Bilevel Mixed Integer Programs (Koeppe, Queyranne, and Ryan; 2010)
- Symmetric games with piecewise linear utilities (Ryan, Jiang, and Leyton-Brown; 2010)
- Counting chemical compositions using Ehrhart quasi-polynomials (Hubler and Craciun; 2012)
- Precise Quantitative Information Flow Analysis – A Symbolic Approach (Klebanov; 2014)

References

Aspinall, D., R. Atkey, K. MacKenzie, and D. Sannella (2010). Symbolic and analytic techniques for resource analysis of java bytecode. In *Trustworthly Global Computing: 5th International Symposium*, pp. 1–22. Springer. [117]

Baldoni, V., N. Berline, and M. Vergne (2008, March). Sum over lattice points of a polygon with iterated Laurent series. user's guide. [75]

Baldoni, V., N. Berline, and M. Vergne (2009, May). Summing a polynomial function over integral points of a polygon. user's guide. [80]

Baldoni-Silva, M. W., M. Beck, C. Cochet, and M. Vergne (2006). Volume computation for polytopes and partition functions for classical root systems. *Discrete & Computational Geometry 35*(4), 551–595. [116]

Banaszczyk, W., A. E. Litvak, A. Pajor, and S. J. Szarek (1999, August). The flatness theorem for nonsymmetric convex bodies via the local theory of banach spaces. *Mathematics of Operations Research 24*(3), 728–750. [90, 106]

Barvinok, A. (2002). *A Course in Convexity*, Volume 54 of *Graduate Studies in Mathematics*. Providence, RI: American Mathematical Society. [90, 106]

Barvinok, A. I. (1992). Computing the volume, counting integral points, and exponential sums. In *Proceedings of the eighth annual symposium on Computational geometry*, pp. 161–170. ACM Press. [43]

Barvinok, A. I. (1994). Computing the Ehrhart polynomial of a convex lattice polytope. *Dicrete Comput. Geom. 12*, 35–48. [45]

Barvinok, A. I. (2006). Computing the Ehrhart quasi-polynomial of a rational simplex. *Math. Comp. 75*, 1449–1466. [116]

Barvinok, A. I. and J. Pommersheim (1999). An algorithmic theory of lattice points in polyhedra. *New Perspectives in Algebraic Combinatorics 38*, 91–147. [58, 103]

Barvinok, A. I. and K. M. Woods (2003, April). Short rational generating functions for lattice point problems. *J. Amer. Math. Soc. 16*, 957–979. [103, 106]

Beck, M., C. Haase, and F. Sottile (2009, January). Formulas of Brion, Lawrence, and Varchenko on rational generating functions for cones. *The Mathematical Intelligencer 31*(1). [116]

Beck, M. and A. W. van Herick (2010). Enumeration of 4 × 4 magic squares. *Math. Comp.*. [116]

Berline, N. (2007, August). personal communication. [68]

Berline, N. and M. Vergne (2006, July). Local Euler-Maclaurin formula for polytopes. http://arXiv.org/abs/math/0507256. [66, 67, 70, 116]

Beyls, K. and E. D'Hollander (2005, 4). Generating cache hints for improved program efficiency. *Journal of Systems Architecture 51*(4), 223–250. [116]

Bik, A. J. C. (1996). *Compiler Support for Sparse Matrix Computations*. Ph. D. thesis, University of Leiden, The Netherlands. [54]

Boulet, P. and X. Redon (2001). Sppoc: manipulation automatique de polyèdres pour la compilation. *Technique et Science Informatiques 20*(8), 1019–1048. [13]

Brion, M. (1988). Points entiers dans les polyèdres convexes. *Annales Scientifiques de l'École Normale Supérieure. Quatrième Série 21*(4), 653–663. [45]

Büeler, B., A. Enge, and K. Fukuda (2000). Exact volume computation for polytopes: A practical study. DMV Seminar Band 29. [56]

Clauss, P. and V. Loechner (1998, July). Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing 19*(2), 179–194. [22]

Clauss, P., F. J. Fernández, D. Garbervetsky, and S. Verdoolaege (2006, October). Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. ICPS Research Report 06-04, Université Louis Pasteur. [115]

Cohen, J. and T. Hickey (1979). Two algorithms for determining volumes of convex polyhedra. *J. ACM 26*(3), 401–414. [56]

Cook, W., M. Hartmann, R. Kannan, and C. McDiarmid (1992). On integer points in polyhedra. *Combinatorica 12*(1), 27–37. [91]

Cook, W., T. Rutherford, H. E. Scarf, and D. F. Shallcross (1993). An implementation of the generalized basis reduction algorithm for integer programming. *ORSA Journal on Computing 5*(2). [35, 90]

De Loera, J. A. (1995, May). *Triangulations of Polytopes and Computational Algebra*. Ph. D. thesis, Cornell University. [56]

De Loera, J. A., D. Haws, R. Hemmecke, P. Huggins, J. Tauzer, and R. Yoshida (2003, November). A user's guide for latte v1.1. software package `LattE` is available at `http://www.math.ucdavis.edu/~latte/`. [36, 58, 61]

De Loera, J. A., R. Hemmecke, J. Tauzer, and R. Yoshida (2004). Effective lattice point counting in rational convex polytopes. *The Journal of Symbolic Computation 38*(4), 1273–1302. [45]

De Loera, J. A. and M. Köppe (2006). Experiments with an algebraic scheme for estimating the number of lattice points in polyhedra. Manuscript in preparation. [43, 57]

Devos, H., S. Verdoolaege, J. Van Campenhout, and D. Stroobandt (2007). Bounds on quasi-polynomials for static program analysis. manuscript in preparation. [115]

Edmonds, J., L. Lovász, and W. R. Pulleyblank (1982). Brick decompositions and the matching rank of graphs. *Combinatorica 2*(3), 247–274. [91]

Ehrhart, E. (1977). *Polynômes arithmétiques et Méthode des Polyèdres en Combinatoire*, Volume 35 of *International Series of Numerical Mathematics*. Basel/Stuttgart: Birkhauser Verlag. [51, 52]

Eisenbrand, F. (2000, July). *Gomory-Chvátal cutting planes and the elementary closure of polyhedra*. Ph. D. thesis, Universität des Saarlandes. [85, 91]

Eisenbrand, F. and G. Shmonin (2007). Parametric integer programming in fixed dimension. [96, 101, 103]

Eisenschmidt, E. and M. Köppe (2007). Integrally indecomposable polytopes and the survivable network design problem. In *Electronic proceedings of the 6th International Workshop on the Design of Reliable Communication Networks, DRCN 2007*. To appear. [85]

Feautrier, P. (1988). Parametric integer programming. *Operationnelle/Operations Research 22*(3), 243–268. [40]

Feautrier, P. (2006). Solving systems of affine (in)equalities: PIP's user's guide. [40]

Fukuda, K. (1993). cdd.c: C-implementation of the double description method for computing all vertices and extremal rays of a convex polyhedron given by a system of linear inequalities. Technical report, Department of Mathematics, Swiss Federal Institute of Technology, Lausanne, Switzerland. program available from http://www.ifor.math.ethz.ch/ fukuda/fukuda.html. [90]

Fukuda, K. (2008). Exact algorithms and software in optimization and polyhedral computation. In *ISSAC '08: Proceedings of the twenty-first international symposium on Symbolic and algebraic computation*, New York, NY, USA, pp. 333–334. ACM. [116]

Gawrilow, E. and M. Joswig (2000). polymake: a framework for analyzing convex polytopes. In G. Kalai and G. M. Ziegler (Eds.), *Polytopes — Combinatorics and Computation*, pp. 43–74. Birkhäuser. [42]

Gelfand, I. M., M. Kapranov, and A. V. Zelevinsky (1994). *Discriminants, Resultants and Multidimensional Determinants*. Birkhauser, Boston. [85]

Gomory, R. E. (1963). An algorithm for integer solutions to linear programming. In R. L. Graves and P. Wolfe (Eds.), *Recent Advances in Mathematical Programming*, New York, pp. 269–302. McGraw-Hill. [40]

Hartmann, M. E. (1989). *Cutting planes and the complexity of the integer hull*. Ph. D. thesis, Ithaca, NY, USA. [91]

Hemmecke, R. (2002). On the computation of hilbert bases of cones. World Scientific. [89]

Hemmecke, R., R. Hemmecke, M. Köppe, P. Malkin, and M. Walter. 4ti2 – a software package for algebraic, geometric and combinatorial problems on linear spaces. Available at `www.4ti2.de`. [89]

Henrici, P. (1974). *Applied and Computational Complex Analysis*. Pure and applied mathematics. New York: Wiley-Interscience [John Wiley & Sons]. Volume 1: Power series—integration—conformal mapping—location of zeros, Pure and Applied Mathematics. [57]

Hubler, S. L. and G. Craciun (2012). Counting chemical compositions using ehrhart quasi-polynomials. *Journal of Mathematical Chemistry*, 1–25. [117]

Huggins, P. (2006). *iB4e*: A software framework for parametrizing specialized LP problems. In A. Iglesias and N. Takayama (Eds.), *ICMS 2006, Proceedings of the Second International Congress on Mathematical Software*, Volume 4151 of *Lecture Notes in Computer Science*, pp. 245–247. Springer. [91]

Hung, M. S. and W. O. Rom (1990, October). An application of the hermite normal form in integer programming. *Linear Algebra and its Applications 140*(15), 163–179. [85]

Kannan, R. (1992). Lattice translates of a polytope and the Frobenius problem. *Combinatorica 12*(2), 161–177. [96, 103]

Kelly, W., V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott (1996, November). The Omega calculator and library. Technical report, University of Maryland. [13]

Klebanov, V. (2014). Precise quantitative information flow analysis – a symbolic approach. *Theoretical Computer Science*. [117]

Koeppe, M., M. Queyranne, and C. T. Ryan (2010). Parametric integer programming algorithm for bilevel mixed integer programs. *Journal of Optimization Theory and Applications 146*(1), 137–150. [117]

Köppe, M. (2006). LattE macchiato, version 1.2-mk-0.7.1, an improved version of De Loera et al.'s LattE program for counting integer points in polyhedra with variants of Barvinok's algorithm. Available from URL `http://www.math.uni-magdeburg.de/~mkoeppe/latte/`. [58, 61]

Köppe, M. (2007). A primal Barvinok algorithm based on irrational decompositions. *SIAM Journal on Discrete Mathematics 21*(1), 220–236. [45, 61, 116]

Köppe, M. (2007, June). personal communication. [93]

Köppe, M. and S. Verdoolaege (2008). Computing parametric rational generating functions with a primal Barvinok algorithm. *The Electronic Journal of Combinatorics 15*, #R16. [47, 67, 115]

Köppe, M., S. Verdoolaege, and K. M. Woods (2008, July). An implementation of the barvinok–woods integer projection algorithm. In M. Beck and T. Stoll

(Eds.), *The 2008 International Conference on Information Theory and Statistical Learning*. [115]

Lagarias, J. C., H. W. Lenstra, Jr., and C.-P. Schnorr (1990). Korkin-zolotarev bases and successive minima of a lattice and its reciprocal lattice. *Combinatorica 10*(4), 333–348. [90, 106]

Lasserre, J. B. and E. S. Zeron (2005). An alternative algorithm for counting lattice points in a convex polytope. *Math. Oper. Res. 30*. [116]

Lee, C. W. (1991). Regular triangulations of convex polytopes. *Applied Geometry and Discrete Mathematics — The Victor Klee Festschrift 4*, 443–456. [56]

Lepelley, D., A. Louichi, and H. Smaoui (2008, April). On Ehrhart polynomials and probability calculations in voting theory. *Social Choice and Welfare 30*(3), 363–383. [116]

Loechner, V. (1997). *Contribution à l'étude des polyèdres paramétrés et applications en parallélisation automatique*. Ph. D. thesis, University Louis Pasteur, Strasbourg. [37]

Loechner, V. (1999, March). Polylib: A library for manipulating parameterized polyhedra. Technical report, ICPS, Université Louis Pasteur de Strasbourg, France. [22, 23, 24, 26, 37, 52]

Loechner, V. and D. K. Wilde (1997, December). Parameterized polyhedra and their vertices. *International Journal of Parallel Programming 25*(6), 525–549. [22]

Makhorin, A. (2006, July). Gnu linear programming kit, reference manual, version 4.11. [90]

Meister, B. (2004, December). *Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization*. Ph. D. thesis, ICPS, Université Louis Pasteur de Strasbourg, France. [32, 55]

Meister, B. and S. Verdoolaege (2008, April). Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In J. Sankaran and T. Vander Aa (Eds.), *Digest of the 6th Workshop on Optimization for DSP and Embedded Systems, ODES-6*. [115]

Pfeifle, J. and J. Rambau (2003). Computing triangulations using oriented matroids. In M. Joswig and N. Takayama (Eds.), *Algebra, Geometry, and Software Systems*, pp. 49–75. Springer. [85]

Pop, S., G.-A. Silber, A. Cohen, C. Bastoul, S. Girbal, and N. Vasilache (2006). GRAPHITE: Polyhedral analyses and optimizations for GCC. Technical Report A/378/CRI, Centre de Recherche en Informatique, École des Mines de Paris, Fontainebleau, France. Contribution to the GNU Compilers Collection Developers Summit 2006 (GCC Summit 06), Ottawa, Canada, June 28–30, 2006. [116]

Preparata, F. P. and M. I. Shamos (1985, August). *Computational Geometry: An Introduction (Monographs in Computer Science)*. Springer. [91]

Pugh, W. (1994). Counting solutions to Presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, pp. 121–134. [65]

Rabl, T. (2006, January). Volume calculation and estimation of parameterized integer polytopes. Master's thesis, Universität Passau. [56, 116]

Ryan, C. T., A. X. Jiang, and K. Leyton-Brown (2010). Symmetric games with piecewise linear utilities. In *BQGT '10: Proceedings of the Behavioral and Quantitative Game Theory*, New York, NY, USA, pp. 1–1. ACM. [117]

Sakellariou, R. (1996, October). *On the Quest for Perfect Load Balance in Loop-Based Parallel Computations*. Ph. D. thesis, University of Manchester. [65]

Sakellariou, R. (1997, August). Symbolic evaluation of sums for parallelising compilers. In A. Sydow (Ed.), *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, Volume 2 of *Wissenschaft & Technik Verlag*, pp. 685–690. [61]

Scarf, H. E. (1981, March). Production sets with indivisibilities-part II: The case of two activities. *Econometrica 49*(2), 395–423. [34]

Scarf, H. E. and K. M. Woods (2006). Neighborhood complexes and generating functions for affine semigroups. *Discrete & Computational Geometry 35*(3), 385–403. [34]

Seghir, R. (2002, June). Dénombrement des point entiers de l'union et de l'image des polyédres paramétrés. Master's thesis, ICPS, Université Louis Pasteur de Strasbourg, France. [29]

Seghir, R., S. Verdoolaege, K. Beyls, and V. Loechner (2004, February). Analytical computation of Ehrhart polynomials and its application in compile-time generated cache hints. Technical Report 118, ICPS, Université Louis Pasteur de Strasbourg, France. [115]

Seghir, R. and V. Loechner (2006, October). Memory optimization by counting points in integer transformations of parametric polytopes. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, CASES 2006, Seoul, Korea.* [103, 116]

Shoup, V. (2004). NTL. Available from `http://www.shoup.net/ntl/`. [30]

Stanley, R. P. (1986). *Enumerative Combinatorics*, Volume 1. Cambridge University Press. [51]

Stanley, R. P. (1993). A monotonicity property of h-vectors and h*-vectors. *European Journal of Combinatorics 14*(3), 251–258. [42]

Tawbi, N. (1994). Estimation of nested loops execution time by integer arithmetic in convex polyhedra. In *Proceedings of the 8th International Parallel Processing Symposium*, pp. 217–221. IEEE Computer Society Press. [61]

Van Engelen, R. A., K. Gallivan, and B. Walsh (2006, September). Parametric timing estimation with the Newton-Gregory formulae. *Journal of Concurrency and Computation: Practice and Experience 18*(10), 1434–1464. [61, 63]

van Herick, A. W. (2007, July). Theoretical and computational methods for lattice point enumeration in inside-out polytopes. Master's thesis, San Francisco State University. [116]

Verdoolaege, S. (2005, April). *Incremental Loop Transformations and Enumeration of Parametric Sets*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium. [26, 28, 29, 30, 32, 34, 36, 38, 40, 60, 66, 67, 79, 103, 104, 115]

Verdoolaege, S. (2016). Presburger formulas and polyhedral compilation. [6]

Verdoolaege, S., K. Beyls, M. Bruynooghe, and F. Catthoor (2004a, October). Experiences with enumeration of integer projections of parametric polytopes. Report CW 395, K.U.Leuven, Department of Computer Science. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW395.abs.html. [115]

Verdoolaege, S., K. Beyls, M. Bruynooghe, R. Seghir, and V. Loechner (2004b, March). Analytical computation of Ehrhart polynomials and its applications for embedded systems. In *2nd Workshop on Optimization for DSP and Embedded Systems, ODES-2*. [115]

Verdoolaege, S., K. Beyls, M. Bruynooghe, R. Seghir, and V. Loechner (2004c, jan). Analytical computation of Ehrhart polynomials and its applications for embedded systems. Report CW 376, Department of Computer Science, K.U.Leuven, Leuven, Belgium. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW376.abs.html. [115]

Verdoolaege, S., R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe (2004d, September). Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations. In *Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, Washington D.C.*, pp. 248–258. [115]

Verdoolaege, S., K. Beyls, M. Bruynooghe, and F. Catthoor (2005a). Experiences with enumeration of integer projections of parametric polytopes. In R. Bodik (Ed.), *Proceedings of 14th International Conference on Compiler Construction, Edinburgh, Scotland*, Volume 3443 of *Lecture Notes in Computer Science*, Berlin, pp. 91–105. Springer-Verlag. [103, 115]

Verdoolaege, S., K. M. Woods, M. Bruynooghe, and R. Cools (2005b). Computation and manipulation of enumerators of integer projections of parametric polytopes. Report CW 392, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium.

[115]

Verdoolaege, S., H. Nikolov, and T. Stefanov (2006, March). Improved derivation of process networks. In *4th Workshop on Optimization for DSP and Embedded Systems, ODES-4*. [116]

Verdoolaege, S., H. Nikolov, and T. Stefanov (2007a). pn: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems, special issue on Embedded Digital Signal Processing Systems 2007*. [116]

Verdoolaege, S., R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe (2007b, June). Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica 48*(1), 37–66. [14, 115]

Verdoolaege, S. and M. Bruynooghe (2008, July). Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison. In M. Beck and T. Stoll (Eds.), *The 2008 International Conference on Information Theory and Statistical Learning*. [82, 83, 116]

Verdoolaege, S. and K. M. Woods (2008). Counting with rational generating functions. *J. Symb. Comput. 43*(2), 75–91. [103, 115]

Verdoolaege, S., G. Janssens, and M. Bruynooghe (2009, June). Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification 21*, pp. 599–613. Springer. [11]

Wilde, D. K. (1993). A library for doing polyhedral operations. Technical Report 785, IRISA, Rennes, France.
http://www.irisa.fr/EXTERNE/bibli/pi/pi785.html. [22]

Woods, K. M. (2004). *Rational Generating Functions and Lattice Point Sets*. Ph. D. thesis, University of Michigan. [101, 110]

Woods, K. M. (2005). Computing the period of an Ehrhart quasi-polynomial. *The Electronic Journal of Combinatorics 12*, R34. [102]

Woods, K. M. (2006, June). personal communication. [52]

List of Acronyms

DNF	Disjunctive Normal Form
GCD	greatest common divisor
HNF	Hermite Normal Form
LCM	least common multiple
LLL	Lenstra, Lenstra and Lovasz' basis reduction algorithm
PIP	Parametric Integer Programming

SNF	Smith Normal Form

Index