`barvinok`: User Guide

Version: barvinok-0.23

Sven Verdoolaege

April 28, 2007

Contents

List of Figures

1 Internal Representation of the `barvinok` library

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.

1.1 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

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

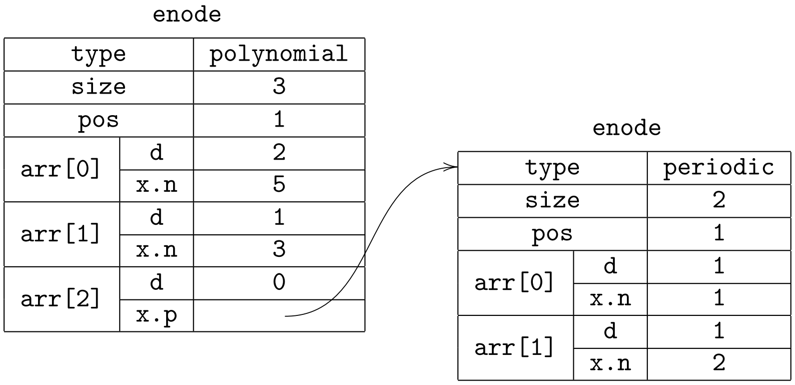
If it is a periodic, then it represents

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

Note that the elements of a `periodic` may themselves be other `periodics` or even `polynomials`. In our library, we only allow the elements of a `periodic` to be other `periodics` 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 1: The quasi-polynomial $[1,2]_p p^2 + 3p + \frac{5}{2}$.

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

1.2 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
    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
    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_NONE     0
    #define     BV_GBR_GLPK     1
    #define     BV_GBR_CDD      2
    int         gbr_lp_solver;

    /* bernstein options */
    #define     BV_BERNSTEIN_NONE   0
    #define     BV_BERNSTEIN_MAX    1
    #define     BV_BERNSTEIN_MIN   -1
    int         bernstein_optimize;

    #define     BV_BERNSTEIN_FACTORS    1
    #define     BV_BERNSTEIN_INTERVALS  2
    int         bernstein_recurse;
};

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. The default is selected by the `--enable-incremental` `configure` option. For more information we refer to Verdoolaege (2005, Section 4.4.3).

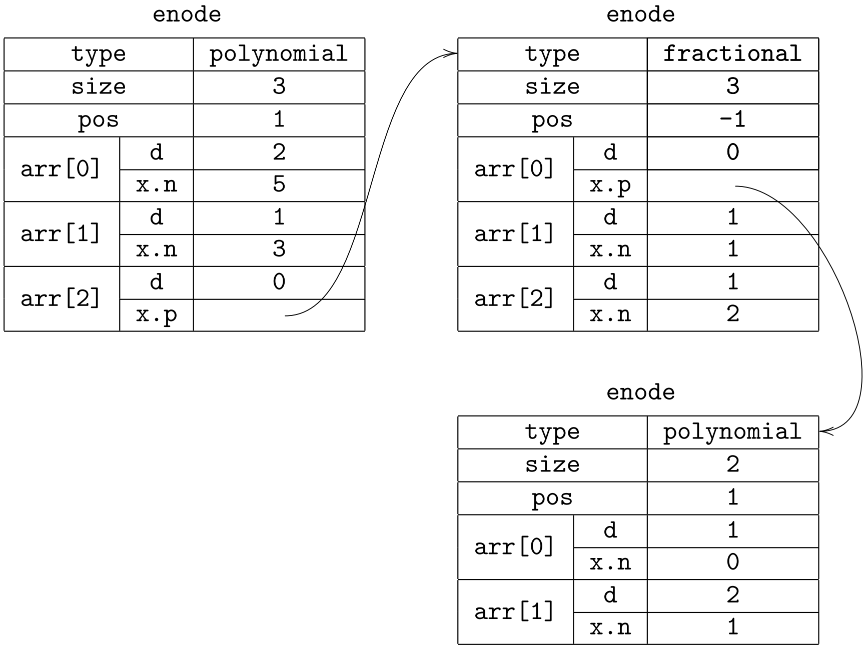
1.3 Data Structures for Quasi-polynomials

Internally, we do not represent our quasi-polynomials as step-polynomials, but, similarly to Loechner (1999), as polynomials with periodic numbers for coefficients. However, we also allow our periodic numbers to be represented by fractional parts of degree-1 polynomials rather than an explicit enumeration using the `periodic` type. By default, the current version of `barvinok` uses `periodics`, but this can be changed through the `--enable-fractional` configure option. In the latter case, the quasi-polynomial using 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`.

---

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

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

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 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.

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.

1.4 Operations on Quasi-polynomials

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

```
void eadd(const evalue *e1,evalue *res);
void emul (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 1.3, 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);
```

The function `esum` 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`.

1.5 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 `short_rat`

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

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

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.

Example 3 Figure 3 shows the internal representation of

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

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);
    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 = NULL);
    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).

1.6 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 1.2.

```
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 1.2.

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_pip`.

```
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_pip(Polyhedron *P,
          unsigned exist, unsigned nparam, unsigned MaxRays);
evalue* barvinok_enumerate_pip_with_options(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_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_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.

1.7 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). The basis vectors are stored in the rows of the matrix returned. This function currently uses `GLPK` (Makhorin 2006) to perform the linear optimizations and so is only available if you have `GLPK`.

```
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 and therefore also requires `GLPK`.

1.8 `bernstein` Data Structures and Functions

The `bernstein` library used `GiNaC` data structures to represent the data it manipulates. In particular, a polynomial is stored in a `GiNaC::ex`, a list of variable or parameter names is stored in a `GiNaC::exvector`, while the parametric vertices or generators are stored in a `GiNaC::matrix`, where the rows refer to the generators and the columns to the coordinates of each generator.

```
namespace bernstein {
GiNaC::exvector constructParameterVector(
    const char * const *param_names, unsigned nbParams);
GiNaC::exvector constructVariableVector(unsigned nbVariables,
                                        const char *prefix);
}
```

The functions `constructParameterVector` and `constructVariableVector` construct a list of variable names either from a list of `char *`s or by suffixing `prefix` with a number starting from 0. Such lists are needed for the functions `domainVertices`, `bernsteinExpansion` and `evalue_bernstein_coefficients`.

```
namespace bernstein {
GiNaC::matrix domainVertices(Param_Polyhedron *PP, Param_Domain *Q,
                             const GiNaC::exvector& params);
}
```

The function `domainVertices` constructs a matrix representing the generators (in this case vertices) of the `Param_Polyhedron` `PP` for the `Param_Domain` `Q`, to be used in a call to `bernsteinExpansion`. The elements of `params` are used in the resulting matrix to refer to the parameters.

```
namespace bernstein {
GiNaC::lst bernsteinExpansion(const GiNaC::matrix& vert,
                              const GiNaC::ex& poly,
                              const GiNaC::exvector& vars,
                              const GiNaC::exvector& params);
}
```

The function `bernsteinExpansion` computes the Bernstein coefficients of the polynomial `poly` over the parametric polytope that is the convex hull of the rows in `vert`. The vectors `vars` and `params` identify the variables (i.e., the coordinates of the space in which the parametric polytope lives) and the parameters, respectively.

```
namespace bernstein {

typedef std::pair< Polyhedron *, GiNaC::lst > guarded_lst;

struct piecewise_lst {
    const GiNaC::exvector vars;
    std::vector<guarded_lst> list;
    /*  0: just collect terms
     *  1: remove obviously smaller terms (maximize)
     * -1: remove obviously bigger terms (minimize)
     */
    int sign;

    piecewise_lst(const GiNaC::exvector& vars);
    piecewise_lst& combine(const piecewise_lst& other);
    void maximize();
    void simplify_domains(Polyhedron *ctx, unsigned MaxRays);
    GiNaC::numeric evaluate(const GiNaC::exvector& values);
    void add(const GiNaC::ex& poly);
}

}
```

A `piecewise_list` structure represents a list of (disjoint) polyhedral domains, each with an associated `GiNaC::lst` of polynomials. The `vars` member contains the variable names of the dimensions of the polyhedral domains.
`piecewise_lst::combine` computes the common refinement of the polyhedral domains in `this` and `other` and associates to each of the resulting subdomains the union of the sets of polynomials associated to the domains from `this` and `other` that contain the subdomain. If the `sign`s of the `piecewise_list`s are not zero, then the (obviously) redundant elements of these sets are removed from the union. The result is stored in `this`.
`piecewise_lst::maximize` removes polynomials from domains that evaluate to a value that is smaller than or equal to the value of some other polynomial associated to the same domain for each point in the domain.
`piecewise_lst::evaluate` "evaluates" the `piecewise_list` by looking for the domain (if any) that contains the point given by `values` and computing the maximal value attained by any of the associated polynomials evaluated at that point.
`piecewise_lst::add` adds the polynomial `poly` to each of the polynomial associated to each of the domains.
`piecewise_lst::simplify_domains` "simplifies" the domains by removing the constraints that are implied by the constraints in `ctx`, basically by calling `PolyLib`'s `DomainSimplify`. Note that you should only do this at the end of your computation. In particular, you do not want to call this method before calling `piecewise_lst::maximize`, since this method will then have less information on the domains to exploit.

```
namespace barvinok {
bernstein::piecewise_lst *evalue_bernstein_coefficients(
    bernstein::piecewise_lst *pl_all, evalue *e,
    Polyhedron *ctx, const GiNaC::exvector& params);
bernstein::piecewise_lst *evalue_bernstein_coefficients(
    bernstein::piecewise_lst *pl_all, evalue *e,
    Polyhedron *ctx, const GiNaC::exvector& params,
    barvinok_options *options);
}
```

The `evalue_bernstein_coefficients` function will compute the Bernstein coefficients of the piecewise parametric polynomial stored in the `evalue e`. The `params` vector specifies the names to be used for the parameters, while the context `Polyhedron ctx` specifies extra constraints on the parameters. The dimension of `ctx` needs to be the same as the length of `params`. The `evalue e` is assumed to be of type `partition` and each of the domains in this `partition` is interpreted as a parametric polytope in the given parameters. The procedure will compute the Bernstein coefficients of the associated polynomial over each such parametric polytope. The resulting `bernstein::piecewise_lst` collects the Bernstein coefficients over all parametric polytopes in `e`. If `pl_all` is not `NULL` then this list will be combined with the list computed by calling `piecewise_lst::combine`. If `bernstein_optimize` is set to `BV_BERNSTEIN_MAX` in `options`, then this combination will remove obviously redundant Bernstein coefficients with respect to upper bound computation and similarly for `BV_BERNSTEIN_MIN`. The default (`BV_BERNSTEIN_NONE`) is to only remove duplicate Bernstein coefficients.

2 Applications included in the `barvinok` distribution

This section describes some application programs provided by the `barvinok` library, available from `http://freshmeat.net/projects/barvinok/`. 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

2.1 `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 1.1. 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 $S = \{\, s \mid s \geq 0 \wedge 2s \leq 13 \,\}$ 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:2  Lines:0
Constraints 2 3
Inequality: [   1   0 ]
Inequality: [  -2  13 ]
Rays 2 3
Vertex: [   0 ]/1
Vertex: [  13 ]/2
   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
```

2.2 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 1.6.) 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
POLYHEDRON Dimension:5
           Constraints:6  Equations:1  Rays:5  Lines:0
Constraints 6 7
Equality:    [   1   1   1   0  -1   0 ]
Inequality: [   0   1   1   1  -1   0 ]
Inequality: [   0   1   0   0   0   0 ]
Inequality: [   0   0   1   0   0   0 ]
```

```
Inequality: [   0  -2  -2   0   1   0 ]
Inequality: [   0   0   0   0   0   1 ]
Rays 5 7
Ray:    [   1   0   1   1   2 ]
Ray:    [   1   1   0   1   2 ]
Vertex: [   0   0   0   0   0 ]/1
Ray:    [   0   0   0   1   0 ]
Ray:    [   1   0   0   1   1 ]
POLYHEDRON Dimension:2
           Constraints:1  Equations:0  Rays:3  Lines:2
Constraints 1 4
Inequality: [   0   0   1 ]
Rays 3 4
Line:   [   1   0 ]
Line:   [   0   1 ]
Vertex: [   0   0 ]/1
         - P + Q  >= 0
         2P - Q  >= 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 )
 )
 )
         Q  >= 0
         P - Q  -1 >= 0
          1 >= 0

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

The output corresponds to

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

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
POLYHEDRON Dimension:4
           Constraints:5  Equations:0  Rays:5  Lines:0
Constraints 5 6
Inequality: [  -1  -1  -1   1   0 ]
Inequality: [   1  -1   0   0   0 ]
Inequality: [   0   1  -1   0   0 ]
Inequality: [   0   0   1   0  -1 ]
Inequality: [   0   0   0   0   1 ]
Rays 5 6
Ray:    [   1   1   1   3 ]
Ray:    [   1   1   0   2 ]
Ray:    [   1   0   0   1 ]
Ray:    [   0   0   0   1 ]
Vertex: [   1   1   1   3 ]/1
POLYHEDRON Dimension:1
           Constraints:1  Equations:0  Rays:2  Lines:1
Constraints 1 3
Inequality: [   0   1 ]
Rays 2 3
Line:   [   1 ]
Vertex: [   0 ]/1
(n^3)/((1-n) * (1-n) * (1-n^2) * (1-n^3))
```

Options:

`--floor`	`-f`	convert `fractionals` to `floorings`
`--convert`	`-c`	convert `fractionals` to `periodics`
`--series`	`-s`	compute rational generating function instead of piecewise step-polynomial
`--explicit`	`-e`	convert computed rational generating function to a piecewise step-polynomial

2.3 `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
POLYHEDRON Dimension:4
           Constraints:5  Equations:1  Rays:4  Lines:0
Constraints 5 6
Equality:   [   1  -6  -9   0   7 ]
Inequality: [   0   1   0   0  -1 ]
Inequality: [   0  -1   0   0   8 ]
Inequality: [   0   0   1   0  -1 ]
Inequality: [   0   0  -1   1   0 ]
Rays 4 6
Vertex: [  50   8   1   1 ]/1
Ray:    [   0   0   0   1 ]
Ray:    [   9   0   1   1 ]
Vertex: [   8   1   1   1 ]/1
exist: 2, nparam: 1
         P  -3 >= 0
          1 >= 0

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

( 8 * P + 0 )
```

Options:

`--floor`	`-f`	convert `fractionals` to `floorings`
`--convert`	`-c`	convert `fractionals` to `periodics`
`--omega`	`-o`	use `Omega` as a preprocessor
`--pip`	`-p`	call `barvinok_enumerate_pip` instead of `barvinok_enumerate_e`

2.4 `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

2.5 `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

2.6 `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`.

2.7 `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

2.8 `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 `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).

4 `Omega` interface

The `barvinok` distribution includes an interface to `Omega` (Kelly et al. 1996b) `occ`, an extension of `oc` (Kelly et al. 1996a). The extension adds the operations shown in Figure 4. Here are some examples:

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

P := {[i,j] : 0 <= i < 4*n-1 and 0 <= j < n and
              n-1 <= i+j <= 3*n-2 };
C1 := {[i,j] : 0 <= i < 4*n-1 and 0 <= j < n and
               2*n-1 <= i+j <= 4*n-2 and i <= 2*n-1 };

count_lexsmaller P within C1;

vertices C1;

bmax { [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 ) };
```

Name	Syntax	Explanation
Card	`card` r	Computes the number of integer points in r and prints the result to standard output
Ranking	`ranking` r	Computes the rank function of r and prints the result to standard output (Loechner et al. 2002; Turjan et al. 2002)
Predecessors	`count_lexsmaller` r `within` d	Computes a function from the elements of d to the number of elements of r that are lexicographically smaller than that element and prints the result to standard output.
Vertices	`vertices` r	Computes the parametric vertices of r using `PolyLib` (Loechner 1999).
Bernstein	`bmax` f	Computes the Bernstein coefficients of the function f over its domain and removes the redundant coefficients by calling `piecewise_lst::maximize`. The results are printed to standard output. See the example for how to specify the function f.

Figure 4: Extra relational operations of `occ`

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*

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

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

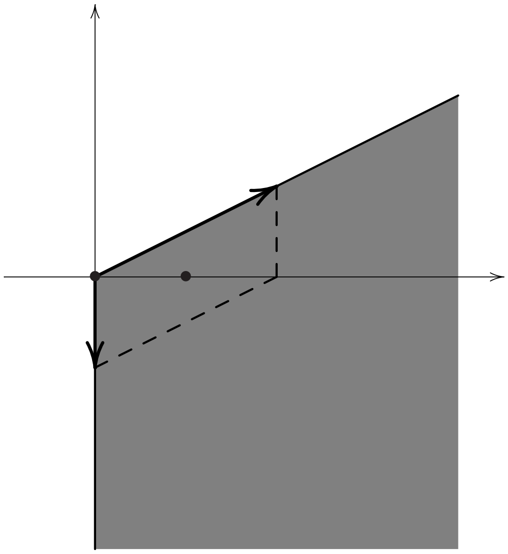


Figure 5: The integer points in the fundamental parallelepiped of K

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 $0 \le k_j < s_j$. $\square$

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 (1), 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 4 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. 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.,

$$K_j = \operatorname{pos} \left(\{ \mathbf{u}_i \}_{i=1}^d \setminus \{ \mathbf{u}_j \} \cup \{ \mathbf{w} \} \right). \tag{2}$$

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

$$\sum_{i=0}^k \beta_i \mathbf{u}_i = \sum_{i=k+1}^{d'} \alpha_i \mathbf{u}_i, \tag{3}$$

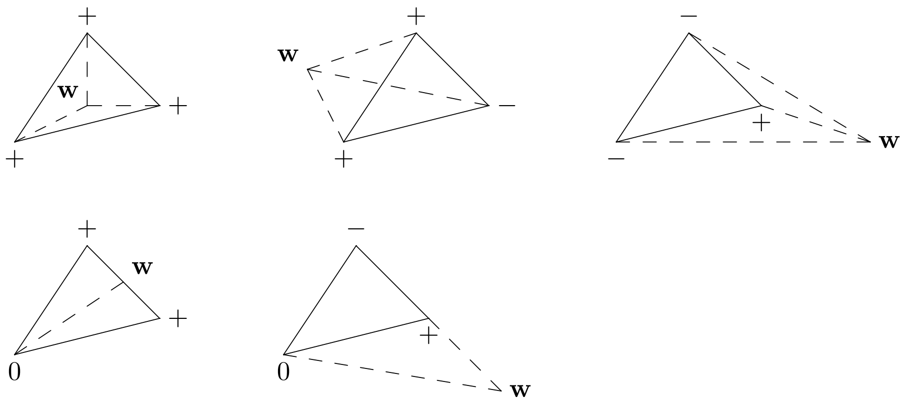


Figure 6: Possible locations of $\mathbf{w}$ with respect to the rays of a 3-dimensional cone. The figure shows a section of the cones.

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

$$\frac{\alpha_j}{\alpha_j + \alpha_l}\mathbf{u}_j + \frac{\alpha_l}{\alpha_j + \alpha_l}\mathbf{u}_l \in H \qquad \text{for } \alpha_i \alpha_l > 0. \tag{4}$$

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

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

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 (3).) The cones $\{\, K_i \,\}_{i=0}^{k}$ and $\{\, K_i \,\}_{i=k+1}^{d'}$ therefore both form a triangulation of K' and hence

$$[K'] = [K] + \sum_{i=1}^{k} [K_i] - \sum_{j \in J_1} [F_j] = \sum_{i=k+1}^{d'} [K_i] - \sum_{j \in J_2} [F_j] \tag{5}$$

or

$$[K] = \sum_{i=1}^{d'} \varepsilon_i \, [K_i] + \sum_j \delta_j \, [F_j], \tag{6}$$

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 6 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 (6). Here we will apply the following proposition.

Proposition 5.3 (Köppe and Verdoolaege (2007)) *Let*

$$\sum_{i \in I_1} \epsilon_i [P_i] + \sum_{i \in I_2} \delta_k [P_i] = 0 \tag{7}$$

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*

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

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*

$$\begin{aligned} \tilde{P}_i = \Big\{ \mathbf{x} \in \mathbb{Q}^d \mid \langle \mathbf{b}^*_{i,j}, \mathbf{x} \rangle \geq \beta_{i,j} \text{ for } j \in J_i \text{ with } \langle \mathbf{b}^*_{i,j}, \mathbf{y} \rangle > 0, \\ \langle \mathbf{b}^*_{i,j}, \mathbf{x} \rangle > \beta_{i,j} \text{ for } j \in J_i \text{ with } \langle \mathbf{b}^*_{i,j}, \mathbf{y} \rangle < 0 \Big\}. \end{aligned} \tag{8}$$

Then

$$\sum_{i \in I_1} \epsilon_i [\tilde{P}_i] = 0. \tag{9}$$

When applying this proposition to (6), we obtain

$$\left[\tilde{K}\right] = \sum_{i=1}^{d'} \varepsilon_i \left[\tilde{K}_i\right], \tag{10}$$

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 (2007).

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., $\alpha_i\alpha_j > 0$ (4), 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 7 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), 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,

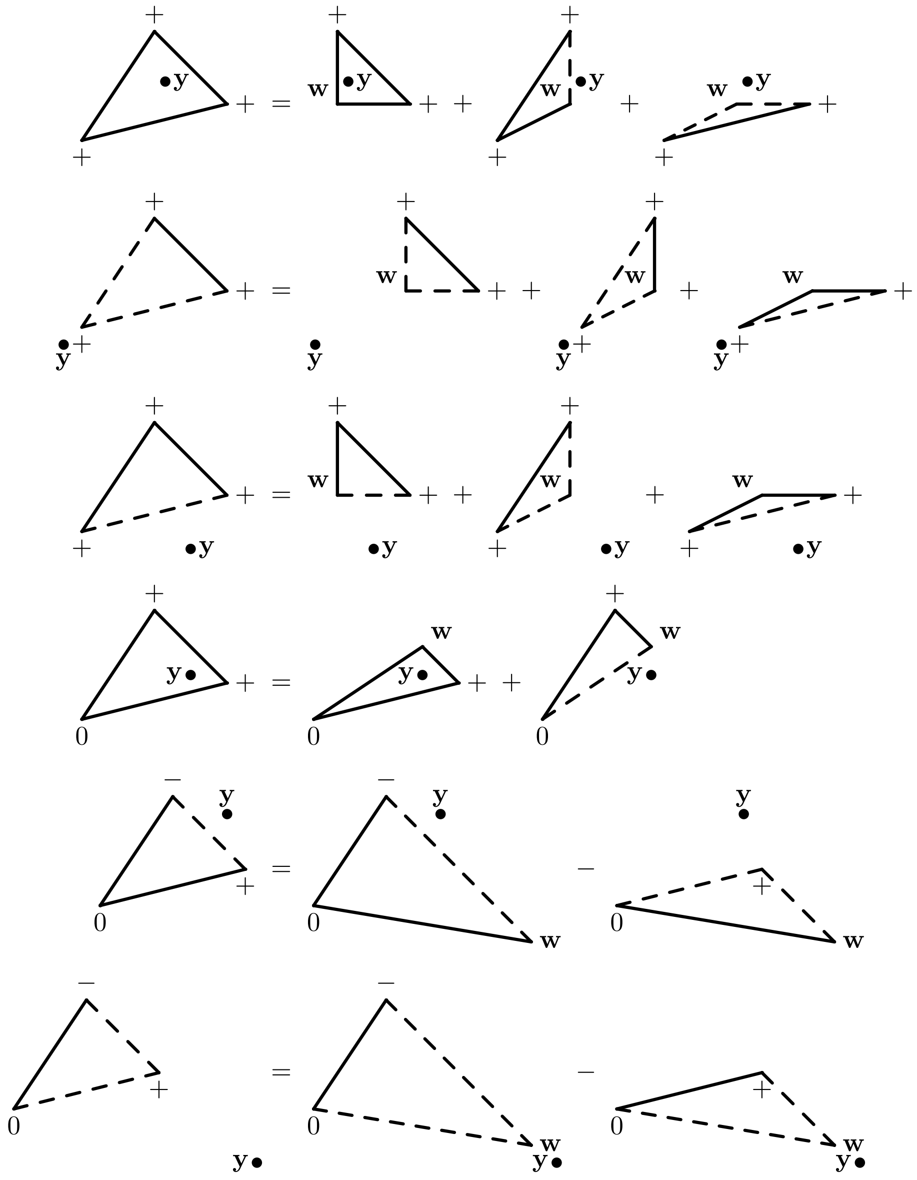

+ y + + = + w y + + + w y + + w y + +
+ y + = + w + + + w + + w +
+ + y = + w y + + w + y + w + y
+ y 0 = w y 0 + + w y 0
− y 0 + = − y 0 w − y 0 + w
− 0 + y = − 0 w y − 0 + w y

Figure 7: Examples of decompositions in primal space.

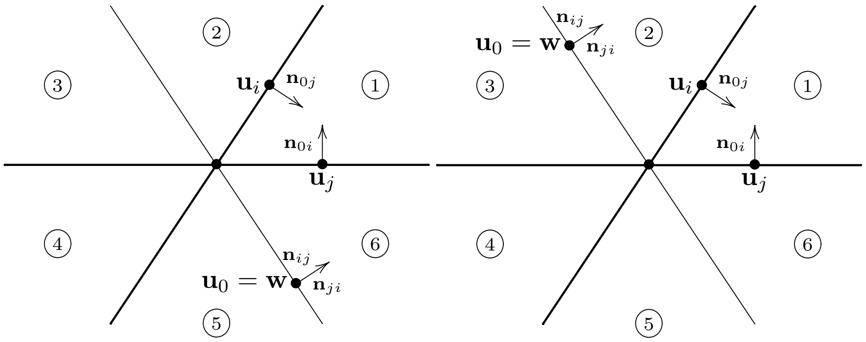


Figure 8: Possible locations of $\mathbf{w}$ with respect to $\mathbf{u}_i$ and $\mathbf{u}_j$, projected onto a plane orthogonal to the other rays, when $\alpha_i \alpha_j < 0$.

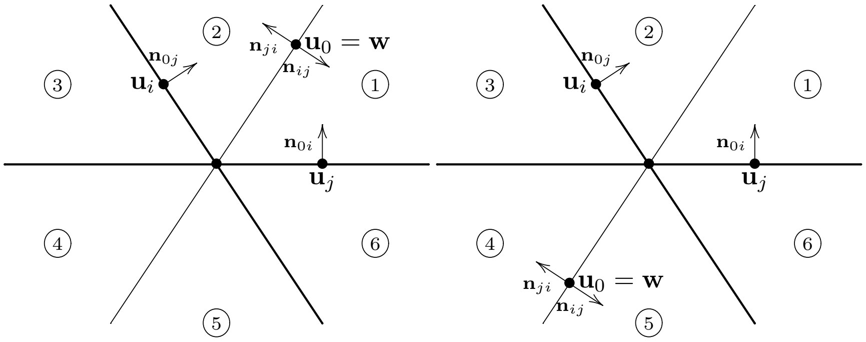


Figure 9: Possible locations of $\mathbf{w}$ with respect to $\mathbf{u}_i$ and $\mathbf{u}_j$, projected onto a plane orthogonal to the other rays, when $\alpha_i \alpha_j > 0$.

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 shown in Figure 8. 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 9 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.

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$[\tilde{K}][\mathbf{u}_j]$
```

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 simple cones do not intersect at their facets. Again, we apply Proposition 5.3 with $\mathbf{y}$ an interior point of the cone (Section 5.1).

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 5.4 *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 5.5 *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 5.6 *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 5.5, 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 1.3). 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 5.6 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 5.7 *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 $g_{\mathbf{i}} \in \mathbb{Q}[T_1, \dots, T_n]$ 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[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

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

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 decription 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 computer 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 computes 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

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

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.

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; 2005)
- Computation and Manipulation of Enumerators of Integer Projections of Parametric Polytopes (Verdoolaege, Woods, Bruynooghe, and Cools; 2005)
- Incremental Loop Transformations and Enumeration of Parametric Sets (Verdoolaege; 2005)
- Counting with rational generating functions (Verdoolaege and Woods; 2005)
- Symbolic Polynomial Maximization over Convex Sets and its Application to Memory Requirement Estimation (Clauss, Fernández, Gabervetsky, and Verdoolaege; 2006)
- Counting integer points in parametric polytopes using Barvinok's rational functions (Verdoolaege, Seghir, Beyls, Loechner, and Bruynooghe; 2007b)
- Approximating the Number of Integer Points in Parametric Polytopes by Polynomials (Meister and Verdoolaege; 2007)
- 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; 2007)

6.2 Publications Refering to the Library

This is a list of some reports and publications refering to the `barvinok` library.

- Theorems of Brion, Lawrence, and Varchenko on rational generating functions for cones (Beck, Haase, and Sottile; 2005)
- 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)
- On Ehrhart Polynomials and Probability Calculations in Voting Theory (Lepelley, Louichi, and Smaoui; 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)

References

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. [45]

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. [29]

Barvinok, A. I. (1994). Computing the Ehrhart polynomial of a convex lattice polytope. *Dicrete Comput. Geom. 12*, 35–48. [31]

Barvinok, A. I. (2006). Computing the Ehrhart quasi-polynomial of a rational simplex. *Math. Comp. 75*, 1449–1466. [45]

Beck, M., C. Haase, and F. Sottile (2005). Theorems of Brion, Lawrence, and Varchenko on rational generating functions for cones. [45]

Beyls, K. and E. D'Hollander (2005, 4). Generating cache hints for improved program efficiency. *Journal of Systems Architecture 51*(4), 223–250. [45]

Bik, A. J. C. (1996). *Compiler Support for Sparse Matrix Computations*. Ph. D. thesis, University of Leiden, The Netherlands. [40]

Brion, M. (1988). Points entiers dans les polyèdres convexes. *Annales Scientifiques de l'École Normale Supérieure. Quatrième Série 21*(4), 653–663. [31]

Büeler, B., A. Enge, and K. Fukuda (2000). Exact volume computation for polytopes: A practical study. DMV Seminar Band 29. [42]

Clauss, P. and V. Loechner (1998, July). Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing 19*(2), 179–194. [3]

Clauss, P., F. J. Fernández, D. Gabervetsky, 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. [44]

Cohen, J. and T. Hickey (1979). Two algorithms for determining volumes of convex polyhedra. *J. ACM 26*(3), 401–414. [42]

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). [16]

De Loera, J. A. (1995, May). *Triangulations of Polytopes and Computational Algebra*. Ph. D. thesis, Cornell University. [42]

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/`. [20]

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. [31]

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. [29]

Devos, H., S. Verdoolaege, J. Van Campenhout, and D. Stroobandt (2007). Bounds on quasi-polynomials for static program analysis. manuscript in preparation. [44]

Ehrhart, E. (1977). *Polynômes arithmétiques et Méthode des Polyèdres en Combinatoire*, Volume 35 of *International Series of Numerical Mathematics*. Basel/Stuttgart: Birkhauser Verlag. [37, 38]

Feautrier, P. (1988). Parametric integer programming. *Operationnelle/Operations Research 22*(3), 243–268. [25]

Feautrier, P. (2006). Solving systems of affine (in)equalities: PIP's user's guide. [25]

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. [26]

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. [25]

Kelly, W., V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott (1996a, November). The Omega calculator and library. Technical report, University of Maryland. [27]

Kelly, W., V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott (1996b, November). The Omega library. Technical report, University of Maryland. [27]

Köppe, M. (2007). A primal Barvinok algorithm based on irrational decompositions. *SIAM Journal on Discrete Mathematics 21*(1), 220–236. [31, 45]

Köppe, M. and S. Verdoolaege (2007). Computing parametric rational generating functions with a primal barvinok algorithm. manuscript in preparation. [33, 45]

Lasserre, J. B. and E. S. Zeron (2005). An alternative algorithm for counting lattice points in a convex polytope. *Math. Oper. Res. 30.* [45]

Lee, C. W. (1991). Regular triangulations of convex polytopes. *Applied Geometry and Discrete Mathematics — The Victor Klee Festschrift 4*, 443–456. [42]

Lepelley, D., A. Louichi, and H. Smaoui (2006, March). On Ehrhart polynomials and probability calculations in voting theory. Technical Report 200610, CERESUR / CREM. [45]

Loechner, V. (1997). *Contribution à l'étude des polyèdres paramétrés et applications en parallélisation automatique*. Ph. D. thesis, University Louis Pasteur, Strasbourg. [21]

Loechner, V. (1999, March). Polylib: A library for manipulating parameterized polyhedra. Technical report, ICPS, Université Louis Pasteur de Strasbourg, France. [3, 4, 5, 7, 21, 28, 38]

Loechner, V. and D. K. Wilde (1997, December). Parameterized polyhedra and their vertices. *International Journal of Parallel Programming 25*(6), 525–549. [3]

Loechner, V., B. Meister, and P. Clauss (2002). Precise data locality optimization of nested loops. *J. Supercomput. 21*(1), 37–76. [28]

Makhorin, A. (2006, July). Gnu linear programming kit, reference manual, version 4.11. [16]

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. [13, 41]

Meister, B. and S. Verdoolaege (2007). Approximating the number of integer points in parametric polytopes by polynomials. manuscript in preparation. [44]

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. [45]

Rabl, T. (2006, January). Volume calculation and estimation of parameterized integer polytopes. Master's thesis. [42, 45]

Scarf, H. E. (1981, March). Production sets with indivisibilities-part II: The case of two activities. *Econometrica 49*(2), 395–423. [15]

Scarf, H. E. and K. M. Woods (2006). Neighborhood complexes and generating functions for affine semigroups. *Discrete & Computational Geometry 35*(3), 385–403. [15]

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. [9]

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. [44]

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.* [45]

Shoup, V. (2004). NTL. Available from `http://www.shoup.net/ntl/`. [11]

Stanley, R. P. (1986). *Enumerative Combinatorics*, Volume 1. Cambridge University Press. [37]

Stanley, R. P. (1993). A monotonicity property of h-vectors and h*-vectors. *European Journal of Combinatorics 14*(3), 251–258. [26]

Turjan, A., B. Kienhuis, and E. Deprettere (2002, July). A compile time based approach for solving out-of-order communication in Kahn process networks. In *IEEE 13th International Conference on Aplication-specific Systems, Architectures and Processors (ASAP'2002).* [28]

Verdoolaege, S. (2005, April). *Incremental Loop Transformations and Enumeration of Parametric Sets.* Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium. [7, 9, 10, 11, 13, 15, 16, 20, 24, 44]

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. [44]

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.* [44]

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. [44]

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 Em-*

bedded Systems, Washington D.C., pp. 248–258. [44]

Verdoolaege, S., K. Beyls, M. Bruynooghe, and F. Catthoor (2005). 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. [44]

Verdoolaege, S. and K. M. Woods (2005, April). Counting with rational generating functions. [44]

Verdoolaege, S., K. M. Woods, M. Bruynooghe, and R. Cools (2005). Computation and manipulation of enumerators of integer projections of parametric polytopes. Report CW 392, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium. [44]

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*. [45]

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*. [45]

Verdoolaege, S., R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe (2007b). Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*. accepted for publication. [44]

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. [3]

Woods, K. M. (2006, June). personal communication. [38]

List of Acronyms

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