Normaliz 3.6.1

Winfried Bruns, Tim Rï¿½mer, Richard Sieg and Christof Sï¿½ger

Normaliz 2 team member: Bogdan Ichim

http://normaliz.uos.de https://github.com/Normaliz

mailto:normaliz@uos.de

https://hub.docker.com/u/normaliz/dashboard/

https://mybinder.org/v2/gh/Normaliz/NormalizJupyter/master

Contents

1. Introduction

1.1. The objectives of Normaliz

The program Normaliz is a tool for computing the Hilbert bases and enumerative data of rational cones and, more generally, sets of lattice points in rational polyhedra. The mathematical background and the terminology of this manual are explained in Appendix A. For a thorough treatment of the mathematics involved we refer the reader to [5]. The terminology follows [5]. For algorithms of Normaliz see [6], [7], [8], [9], [10], and [11]. Some new developments are briefly explained in this manual.

Both polyhedra and lattices can be given by

(1) systems of generators and/or
(2) constraints.

Since version 3.1, cones need not be pointed and polyhedra need not have vertices, but are allowed to contain a positive-dimensional affine subspace.

In addition to generators and constraints, affine monoids can be defined by lattice ideals, in other words, by binomial equations.

In order to describe a rational polyhedron by *generators*, one specifies a finite set of vertices $x_1,\dots,x_n \in \mathbb{Q}^d$ and a set $y_1,\dots,y_m \in \mathbb{Z}^d$ generating a rational cone C. The polyhedron defined by these generators is

$$P = \operatorname{conv}(x_1,\dots,x_n) + C, \qquad C = \mathbb{R}_+ y_1 + \cdots + \mathbb{R}_+ y_m.$$

An affine lattice defined by generators is a subset of $\mathbb{Z}^d$ given as

$$L = w + L_0, \qquad L_0 = \mathbb{Z} z_1 + \cdots + \mathbb{Z} z_r, \qquad w, z_1,\dots,z_r \in \mathbb{Z}^d.$$

Constraints defining a polyhedron are affine-linear inequalities with integral coefficients, and the constraints for an affine lattice are affine-linear diophantine equations and congruences. The conversion between generators and constraints is an important task of Normaliz.

The first main goal of Normaliz is to compute a system of generators for

$$P \cap L.$$

The minimal system of generators of the monoid $M = C \cap L_0$ is the Hilbert basis $\operatorname{Hilb}(M)$ of M. The homogeneous case, in which $P = C$ and $L = L_0$, is undoubtedly the most important one, and in this case $\operatorname{Hilb}(M)$ is the system of generators to be computed. In the general case the system of generators consists of $\operatorname{Hilb}(M)$ and finitely many points $u_1,\dots,u_s \in P \cap L$ such that

$$P \cap L = \bigcup_{j=1}^{s} u_j + M.$$

The second main goal are enumerative data that depend on a grading of the ambient lattice. Normaliz computes the Hilbert series and the Hilbert quasipolynomial of the monoid or set

of lattice points in a polyhedron. In combinatorial terminology: Normaliz computes Ehrhart series and quasipolynomials of rational polyhedra. Normaliz also computes weighted Ehrhart series and Lebesgue integrals of polynomials over rational polytopes.

Normaliz now has a variant called QNormaliz. Its basic number class are elements of an algebraic extension of $\mathbb{Q}$ inside $\mathbb{R}$. This extends the scope of Normaliz to algebraic polyhedra, but computations that depend on the finite generation of the monoid of lattice points in a cone are of course not possible.

The computation goals of Normaliz can be set by the user. In particular, they can be restricted to subtasks, such as the lattice points in a polytope or the leading coefficient of the Hilbert (quasi)polynomial.

Performance data of Normaliz can be found in [8].

Acknowledgement. In 2013–2016 the development of Normaliz has been supported by the DFG SPP 1489 "Algorithmische und experimentelle Methoden in Algebra, Geometrie und Zahlentheorie".

1.2. Platforms, implementation and access from other systems

Executables for Normaliz are provided for Mac OS, Linux and MS Windows. If the executables prepared cannot be run on your system, then you can compile Normaliz yourself (see Section 10). The statically linked Linux binaries provided by us can be run in the Linux subsystem of MS Windows 10. NA Docker image of Normaliz is available.

Normaliz is written in C++, and should be compilable on every system that has a GCC compatible compiler. It uses the standard packages Boost and GMP (see Section 10). The parallelization is based on OpenMP. CoCoaLib [1] and Flint [15] are optional packages.

Normaliz consists of two parts: the front end " normaliz" for input and output and the C++ library "libnormaliz" that does the computations.

Normaliz can be accessed from the interactive general purpose system PYTHON via the interface PYNORMALIZ written by Sebastian Gutsche with contributions by Justin Shenk and Richard Sieg.

Normaliz can also be accessed from the following systems:

- SINGULAR via the library `normaliz.lib`,
- MACAULAY2 via the package `Normaliz.m2`,
- COCOA via an external library and libnormaliz,
- GAP via the GAP package NORMALIZINTERFACE [12] which uses libnormaliz,
- POLYMAKE (thanks to the POLYMAKE team),
- SAGEMATH via PyNormaliz (on preparation, thanks to Matthias Kï¿½ppe).

The Singular and Macaulay2 interfaces are contained in the Normaliz distribution. At present, their functionality is limited to Normaliz 2.10. Nevertheless they profit from newer versions.

Furthermore, Normaliz is used by the B. Burton's system REGINA and in SECDEC by S. Borowka et al.

Normaliz does not have its own interactive shell. We recommend the access via PyNormaliz, GAP or SageMath for interactive use.

1.3. Major changes relative to version 3.1.1

In 3.2.0;

(1) installation via `autotools` (written by Matthias Kï¿½ppe),
(2) automatic choice of algorithmic variants (can be switched off),
(3) sparse format for vectors and matrices,
(4) constraints in symbolic form,
(5) Hilbert series with denominator corresponding to a homogeneous system of parameters,
(6) choice of an output directory,
(7) improvements of `libnormaliz`: extension of enumeration `ConeProperty`, constructors based on `Matrix<Integer>`, additional functions for retrieval of results,
(8) a better implementation of `Approximate` and its use in the inhomogeneous case,
(9) option `Symmetrize` that produces symmetrized input for nmzIntegrate and runs nmzIntegrate on this input,
(10) QNormaliz, a version of Normaliz using coordinates in an extension of $\mathbb{Q}$ (restricted to convex hull computations and triangulation).

In 3.3.0:

(1) inclusion of NmzIntegrate in Normaliz as a part of libnormaliz,
(2) fractions in input files,
(3) controlled interruption of Normaliz.

In 3.4.0:

(1) limit of parallelization via libnormaliz,
(2) floating point numbers in input,
(3) project-and-lift algorithm for lattice points in polytopes, also with a floating point variant,
(4) significantly improved subdivision, equivalent replacement for Scip,
(5) fast Gorenstein test,
(6) restriction of number of significant coefficients of quasipolynomial,
(7) definition of semi-open parallelepipeds in input and output of their lattice points.

3.4.1 is a pure bugfix.

In 3.5.0:

(1) Use of LLL reduced coordinates in project-and-lift,
(2) expansion of Hilbert series and weghted Ehrhart series for a range of degrees,
(3) projection of cones and polyhedra,
(4) Euclidean volumes,
(5) optional library Flint for univariate polynomial arithmetic,
(6) Docker image,

(7) install scripts.

In 3.5.1:

(1) Unique output of bases of sublattices and equations (necessary for tests on old systems where LLL can produce different results).

In 3.5.2:

(1) Descent algorithm for multiplicities and volumes of polytopes.

3.5.3 is identical to 3.5.2 in the user interface.

In 3.5.4:

(1) Triangulation, cone and Stanley decomposition available for cones over polytopes defined by inhomogeneous data.
(2) Computation goal EhrhartSeries introduced for polytopes defined by inhomogeneous data.

In 3.6.0:

(1) QNormaliz for algebraic polytopes part of the release.
(2) Computation goal EhrhartQuasipolynomial introduced, use of KeepOrder extended, and Hilbert vasis of recession cone can be set as precomputed data.
(3) Improved installation.

In 3.6.1:

(1) Improvements in build scripts.
(2) Introduction of NoGradingDenom and better treatment of polytopes.
(3) Refinements in the computation of voumes.

See the file `CHANGELOG` in the basic package for more information on the history of Normaliz.

1.4. Future extensions

(1) Computation and exploitation of automorphism groups,
(2) multigraded Hilbert series,
(3) access from further systems,
(4) Grï¿½bner and Graver bases.

1.5. Download and installation

In order to install Normaliz you should first download the basic package. Follow the instructions in

`http://normaliz.uos.de/download/.`

They guide you to our GitHub repository

`https://github.com/Normaliz/Normaliz/releases.`

There you will also find executables for Linux, Mac OS and MS Windows. Unzip the basic package and the executable for your system in a directory of your choice. In it, a directory `normaliz-3.6.1` (called Normaliz directory in the following) is created with several subdirectories.

An alternative to the basic package and the (system dependent) executable is the

Docker image `normaliz/normaliz`

that is automatically downloaded from the Docker repository if you ask for it. (In the docker container, the Normaliz directory is called `Normaliz`, independently of the version number.)

See Section 9 for more details on the distribution and the Docker image, and consult Section 10 if you want to compile Normaliz yourself.

1.6. Exploring Normaliz online

You can explore Normaliz online at

`https://mybinder.org/v2/gh/Normaliz/NormalizJupyter/master`.

The button 'New" offers you a terminal. Choose it, and you will be in a Docker container based on the Normaliz Docker image. Your username is `norm`, and Normaliz is cotained in the subdirectory `Normaliz` of your home directory. Moreover, it is installed, and can be invoked by the commad `normaliz` from anywhere. Just type

```
normaliz -c Normaliz/example/rational
```

to run a small computation. You van also have a python shell and run PyNoormaliz or study the tutorial of PyNormaliz (a Jupyter notebook).

It is possible to upload and download files, but please refrain from using Bimder as a platform for heavy computations.

2. Normaliz by examples

2.1. Terminology

For the precise interpretation of parts of the Normaliz output some terminology is necessary, but this section can be skipped at first reading, and the user can come back to it when it becomes necessary. We will give less formal descriptions along the way.

As pointed out in the introduction, Normaliz "computes" intersections $P \cap L$ where P is a rational polyhedron in $\mathbb{R}^d$ and L is an affine sublattice of $\mathbb{Z}^d$. It proceeds as follows:

(1) If the input is inhomogeneous, then it is homogenized by introducing a homogenizing coordinate: the polyhedron P is replaced by the cone $C(P)$: it is the closure of $\mathbb{R}_+(P \times$

$\{1\}$) in $\mathbb{R}^{d+1}$. Similarly L is replaced by $\widetilde{L} = \mathbb{Z}(L \times \{1\})$. In the homogeneous case in which P is a cone and L is a subgroup of $\mathbb{Z}^d$, we set $C(P) = P$ and $\widetilde{L} = L$.

(2) The computations take place in the *efficient lattice*

$$\mathbb{E} = \widetilde{L} \cap \mathbb{R}C(P).$$

where $\mathbb{R}C(P)$ is the linear subspace generated by $C(P)$. The internal coordinates are chosen with respect to a basis of $\mathbb{E}$. The *efficient cone* is

$$\mathbb{C} = \mathbb{R}_+(C(P) \cap \mathbb{E}).$$

(3) Inhomogeneous computations are truncated using the dehomogenization (defined implicitly or explicitly).

(4) The final step is the conversion to the original coordinates. Note that we must use the coordinates of $\mathbb{R}^{d+1}$ if homogenization has been necessary, simply because some output vectors may be non-integral otherwise.

Normaliz computes inequalities, equations and congruences defining $\mathbb{E}$ and $\mathbb{C}$. The output contains only those constraints that are really needed. They must always be used jointly: the equations and congruences define $\mathbb{E}$, and the equations and inequalities define $\mathbb{C}$. Altogether they define the monoid $M = \mathbb{C} \cap \mathbb{E}$. In the homogeneous case this is the monoid to be computed. In the inhomogeneous case we must intersect M with the dehomogenizing hyperplane to obtain $P \cap L$.

In this section, only pointed cones (and polyhedra with vertices) will be discussed. Nonpointed cones will be addressed in Section 6.14.

2.2. Practical preparations

You may find it comfortable to run Normaliz via the GUI jNormaliz [3]. In the Normaliz directory open jNormaliz by clicking `jNormaliz.jar` in the appropriate way. (We assume that Java is installed on your machine.) In the jNormaliz file dialogue choose one of the input files in the subdirectory `example`, say `small.in`, and press `Run`. In the console window you can watch Normaliz at work. Finally inspect the output window for the results.

The menus and dialogues of jNormaliz are self explanatory, but you can also consult the documentation [3] via the help menu.

Remark The jNormaliz drop down menus do presently not cover all options of Normaliz. But since all computation goals and algorithmic variants can be set in the input file, there is no real restriction in using jNormaliz. The only option not reachable by jNormaliz is the output directory (see Section 5.5).

Moreover, one can, and often will, run Normaliz from the command line. This is fully explained in Section 5. At this point it is enough to call Normaliz by typing

```
normaliz -c <project>
```

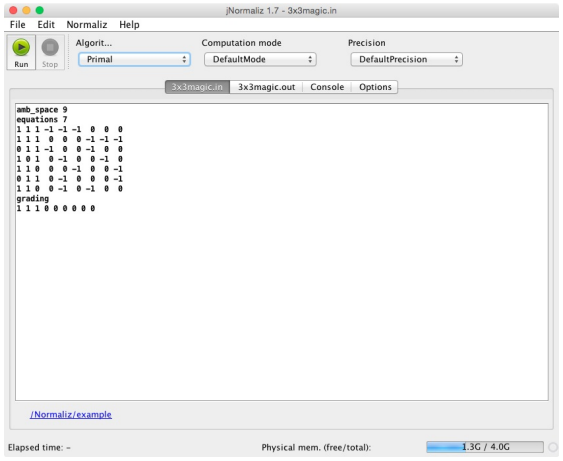


Figure 1: jNormaliz

where `<project>` denotes for the project to be computed. Normaliz will load the file `<project>.in`. The option `-c` makes Normaliz to write a progress report on the terminal. Normaliz writes its results to `<project>.out`.

Note that you may have to prefix `normaliz` by a path name, and `<project>` must contain a path to the input file if it is not in the current directory. Suppose the Normaliz directory is the current directory and we are using a Linux or Mac system. Then

```
./normaliz -c example/small
```

will run `small.in` from the directory `example`. On Windows we must change this to

```
.\normaliz -c example\small
```

The commands given above will run Normaliz with the at most 8 parallel threads. For the very small examples in this tutorial you may want to add `-x=1` to suppress parallelization. For large examples, you can increase the number of parallel threads by `-x=<N>` where `<N>` is the number of threads that you want to suggest. See Section 5.3.

As long as you don't specify a computation goal on the command line or in the input file, Normaliz will use the *default computation goals*:

```
HilbertBasis
HilbertSeries
ClassGroup
```

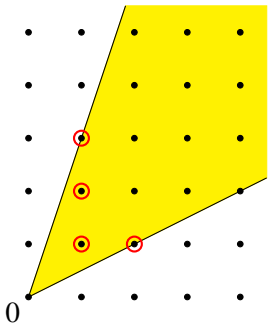
The computation of the Hilbert series requires the explicit or implicit definition of a grading. Normaliz does only complain that a computation goal cannot be reached if the goal has been set explicitly. For example, if you say `HilbertSeries` and there is no grading, an exception will be thrown and Normaliz terminates, but an output file with the already computed data will be written.

Normaliz will always print the results that are obtained on the way to the computation goals and do not require extra effort.

Appendix B helps you to read the console output that you have demanded by the option `-c`.

2.3. A cone in dimension 2

We want to investigate the cone $C = \mathbb{R}_+(2,1) + \mathbb{R}_+(1,3) \subset \mathbb{R}^2$:



This cone is defined in the input file `2cone.in`:

```
amb_space 2
cone 2
1 3
2 1
```

The input tells Normaliz that the ambient space is $\mathbb{R}^2$, and then a cone with 2 generators is defined, namely the cone C from above.

The figure indicates the Hilbert basis, and this is our first computation goal.

If you prefer to consider the columns of a matrix as input vectors (or have a matrix in this format from another system) you can use the input

```
amb_space 2
cone transpose 2
1 2
3 1
```

Note that the number `2` following `transpose` is now the number of *columns*. Later on we will also show the use of formatted matrices.

2.3.1. The Hilbert basis

In order to compute the Hilbert basis, we run Normaliz from jNormaliz or by

```
./normaliz -c example/2cone
```

and inspect the output file:

```
4 Hilbert basis elements
2 extreme rays
2 support hyperplanes
```

Self explanatory so far.

```
embedding dimension = 2
rank = 2 (maximal)
external index = 1
internal index = 5
original monoid is not integrally closed
```

The embedding dimension is the dimension of the space in which the computation is done. The rank is the rank of the lattice $\mathbb{E}$ (notation as in Section 2.1). In fact, in our example $\mathbb{E} = \mathbb{Z}^2$, and therefore has rank 2.

For subgroups $G \subset U \subset \mathbb{Z}^d$ we denote the order of the torsion subgroup of U/G by the *index* of G in U. The *external index* is the index of the lattice $\mathbb{E}$ in $\mathbb{Z}^d$. In our case $\mathbb{E} = \mathbb{Z}^d$, and therefore the external index is 1. Note: the external index is 1 exactly when $\mathbb{E}$ is a direct summand of $\mathbb{Z}^d$.

For this example and many others the *original monoid* is well defined: the generators of the cone used as input are contained in $\mathbb{E}$. (This need not be the case if $\mathbb{E}$ is a proper sublattice of $\mathbb{Z}^d$, and we let the original monoid be undefined in inhomogeneous computations.) Let G be the subgroup generated by the original monoid. The *internal index* is the index of G in $\mathbb{E}$.

The original monoid is *integrally closed* if and only if the it contains the Hilbert basis, and this is evidently false for our example. We go on.

```
size of triangulation   = 1
resulting sum of |det|s = 5
```

The primal algorithm of Normaliz relies on a (partial) triangulation. In our case the triangulation consists of a single simplicial cone, and (the absolute value of) its determinant is 5.

```
No implicit grading found
```

If you do not define a grading explicitly, Normaliz tries to find one itself: the grading is defined if and only if there is a linear form γ on $\mathbb{E}$ under which all extreme rays of the efficient cone $\mathbb{C}$ have value 1, and if so, γ is the implicit grading. Such does not exist in our case.

The last information before we come to the vector lists:

```
rank of class group = 0
finite cyclic summands:
5: 1
```

The class group of the monoid M has rank 0, in other words, it is finite. It has one finite cyclic summand of order 5.

This is the first instance of a multiset of integers displayed as a sequence of pairs

```
<n>: <m>
```

Such an entry says: the multiset contains the number `<n>` with multiplicity `<m>`.

Now we look at the vector lists (typeset in two columns to save space):

```
4 Hilbert basis elements:       2 extreme rays:
1 1                             1 3
1 2                             2 1
1 3
2 1                             2 support hyperplanes:
                                -1  2
                                 3 -1
```

The support hyperplanes are given by the linear forms (or inner normal vectors):

$$\begin{aligned} -x_1 + 2x_2 &\geq 0, \\ 3x_1 - x_2 &\geq 0. \end{aligned}$$

If the order is not fixed for some reason, Normaliz sorts vector lists as follows : (1) by degree if a grading exists and the application makes sense, (2) lexicographically.

2.3.2. The cone by inequalities

Instead by generators, we can define the cone by the inequalities just computed (`2cone_ineq.in`). We use this example to show the input of a formamtted matrix:

```
amb_space auto
inequalities
[[-1  2] [3 -1]]
```

A matrix of input type `inequalities` contains *homogeneous* inequalities. Normaliz can determine the dimension of the ambient space from the formatted matrix. Therefore we can declare the ambient space as being "auto determined" (but `amb_space 2` is not forbidden).

We get the same result as with `2cone.in` except that the data depending on the original monoid cannot be computed: the internal index and the information on the original monoid are missing since there is no original monoid.

2.3.3. The interior

Now we want to compute the lattice points in the interior of our cone. If the cone C is given by the inequalities $\lambda_i(x) \geq 0$ (within $\mathrm{aff}(C)$), then the interior is given by the inequalities $\lambda_i(x) > 0$. Since we are interested in lattice points, we work with the inequalities $\lambda_i(x) \geq 1$.

The input file `2cone_int.in` says

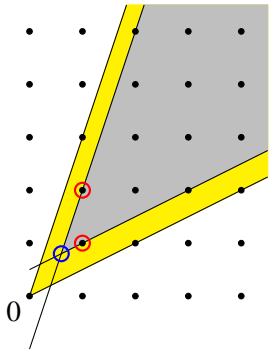
```
amb_space 2
strict_inequalities 2
```

```
-1  2
3 -1
```

The strict inequalities encode the conditions

$$-x_1 + 2x_2 \geq 1,$$
$$3x_1 - x_2 \geq 1.$$

This is our first example of inhomogeneous input.



Alternatively we could use the following two equivalent input files, in a more intuitive notation:

```
amb_space 2
constraints 2
-1  2 > 0
3 -1 > 0
```

```
amb_space 2
constraints 2
-1  2 >= 1
3 -1 >= 1
```

There is an even more intuitive way to type the input file using symbolic constraints that we will introduce in Section 2.6.2.

Normaliz homogenizes inhomogeneous computations by introducing an auxiliary homogenizing coordinate x_{d+1}. The polyhedron is obtained by intersecting the homogenized cone with the hyperplane $x_{d+1} = 1$. The recession cone is the intersection with the hyperplane $x_{d+1} = 0$. The recession monoid is the monoid of lattice points in the recession cone, and the set of lattice points in the polyhedron is represented by its system of module generators over the recession monoid.

Note that the homogenizing coordinate serves as the denominator for rational vectors. In our example the recession cone is our old friend that we have already computed, and therefore we need not comment on it.

```
2 module generators
4 Hilbert basis elements of recession monoid
1 vertices of polyhedron
2 extreme rays of recession cone
3 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 2
```

The only surprise may be the embedding dimension: Normaliz always takes the dimension of the space in which the computation is done. It is the number of components of the output vectors. Because of the homogenization it has increased by 1.

```
size of triangulation   = 1
resulting sum of |det|s = 25
```

In this case the homogenized cone has stayed simplicial, but the determinant has changed.

```
dehomogenization:
0 0 1
```

The dehomogenization is the linear form δ on the homogenized space that defines the hyperplanes from which we get the polyhedron and the recession cone by the equations $\delta(x) = 1$ and $\delta(x) = 0$, respectively. It is listed since one can also work with a user defined dehomogenization.

```
module rank = 1
```

This is the rank of the module of lattice points in the polyhedron over the recession monoid. In our case the module is an ideal, and so the rank is 1.

The output of inhomogeneous computations is always given in homogenized form. The last coordinate is the value of the dehomogenization on the listed vectors, 1 on the module generators, 0 on the vectors in the recession monoid:

```
2 module generators:          4 Hilbert basis elements of recession monoid:
1 1 1                         1 1 0
1 2 1                         1 2 0
                              1 3 0
                              2 1 0
```

The module generators are $(1,1)$ and $(1,2)$.

```
1 vertices of polyhedron:
3 4 5
```

Indeed, the polyhedron has a single vertex, namely $(3/5,4/5)$.

```
2 extreme rays of recession cone:     3 support hyperplanes of polyhedron (homogenized):
```

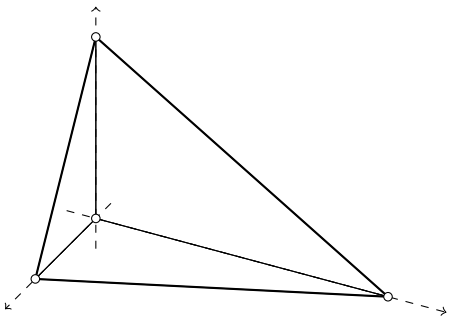
```
1 3 0                                   -1  2 -1
2 1 0                                    0  0  1
                                         3 -1 -1
```

Two support hyperplanes are exactly those that we have used to define the polyhedron – and it has only 2. But Normaliz always outputs the support hyperplanes that are needed for the cone that one obtains by homogenizing the polyhedron, as indicated by 'homogenized". The homogenizing variable is always ≥ 0. In this case the support hyperplane $(0,0,1)$ is essential for the description of the cone. Note that it need not always appear.

2.4. A lattice polytope

The file `polytope.in` contains

```
amb_space 4
polytope 4
0 0 0
2 0 0
0 3 0
0 0 5
```



This is a good place to mention that Normaliz also accepts matrices (and vectors) in sparse format:

```
amb_space 4
polytope 4 sparse
;
1:2;
2:3;
3:5;
```

Each input row, concluded by `;`, lists the indices and the corresponding nonzero values in that row of the matrix.

The Ehrhart monoid of the integral polytope with the 4 vertices

$$(0,0,0)\,,\quad (2,0,0)\,,\quad (0,3,0)\quad \text{and}\quad (0,0,5)$$

in $\mathbb{R}^3$ is to be computed. The generators of the Ehrhart monoid are obtained by attaching a further coordinate 1 to the vertices, and this explains `amb_space 4`. In fact, the input type `polytope` is not only a convenient version of

```
cone 4
0 0 0 1
2 0 0 1
0 3 0 1
0 0 5 1
```

It also sets the he grading to be the last coordinate. See 3.10 below for general information on gradings.

Running `normaliz` produces the file `polytope.out`:

```
19 Hilbert basis elements
18 lattice points in polytope (Hilbert basis elements of degree 1)
4 extreme rays
4 support hyperplanes

embedding dimension = 4
rank = 4 (maximal)
external index = 1
internal index = 30
original monoid is not integrally closed
```

Perhaps a surprise: the lattice points of the polytope do not yield all Hilbert basis elements.

```
size of triangulation   = 1
resulting sum of |det|s = 30
```

Nothing really new so far. The grading appears in the output file:

```
grading:
0 0 0 1

degrees of extreme rays:
1: 4
```

Again we encounter the notation `<n>: <m>`: we have 4 extreme rays, all of degree 1.

```
Hilbert basis elements are not of degree 1
```

Perhaps a surprise: the polytope is not integrally closed as defined in [5]. Now we see the enumerative data defined by the grading:

```
multiplicity = 30

Hilbert series:
1 14 15
denominator with 4 factors:
1: 4

degree of Hilbert Series as rational function = -2

Hilbert polynomial:
1 4 8 5
with common denominator = 1
```

The polytope has $\mathbb{Z}^3$-normalized volume 30 as indicated by the multiplicity (see Section 6.1.1

for a discussion of volumes and multiplicities). The Hilbert (or Ehrhart) function counts the lattice points in kP, $k \in \mathbb{Z}_+$. The corresponding generating function is a rational function $H(t)$. For our polytope it is

$$\frac{1+14t+15t^2}{(1-t)^4}.$$

The denominator is given in multiset notation: `1: 4` say that the factor $(1-t^1)$ occurs with multiplicity 4.

The Ehrhart polynomial (again we use a more general term in the output file) of the polytope is

$$p(k) = 1+4k+8k^2+5k^3\,.$$

In our case it has integral coefficients, a rare exception. Therefore one usually needs a denominator.

Everything that follows has already been explained.

```
rank of class group = 0
finite cyclic summands:
30: 1

**********************************************************************

18 lattice points in polytope (Hilbert basis elements of degree 1):
0 0 0 1
...
2 0 0 1

1 further Hilbert basis elements of higher degree:
1 2 4 2

4 extreme rays:                 4 support hyperplanes:
0 0 0 1                         -15 -10 -6 30
0 0 5 1                         0   0  1  0
0 3 0 1                         0   1  0  0
2 0 0 1                         1   0  0  0
```

The support hyperplanes give us a description of the polytope by inequalities: it is the solution of the system of the 4 inequalities

$$x_3 \geq 0\,, \quad x_2 \geq 0\,, \quad x_1 \geq 0 \quad \text{and} \quad 15x_1+10x_2+6x_3 \leq 30\,.$$

2.4.1. Only the lattice points

Suppose we want to compute only the lattice points in our polytope. In the language of graded monoids these are the degree 1 elements, and so we add `Deg1Elements` to our input file (`polytope_deg1.in`):

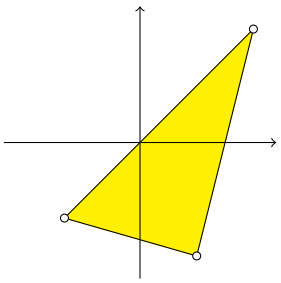
```
amb_space 4
polytope 4
0 0 0
2 0 0
0 3 0
0 0 5
Deg1Elements
/* This is our first explicit computation goal*/
```

We have used this opportunity to include a comment in the input file. The computation of lattice points in a polytope will be taken up again in Sections 2.13 and 6.2.

We lose all information on the Hilbert series, and from the Hilbert basis we only retain the degree 1 elements.

2.5. A rational polytope

The type `polytope` can (now) be used for rational polytopes as well.



We want to investigate the Ehrhart series of the triangle P with vertices

$$(1/2,1/2),\ (-1/3,-1/3),\ (1/4,-1/2).$$

For this example the procedure above yields the input file `rational.in`:

```
amb_space 3
polytope 3
1/2 1/2
-1/3 -1/3
1/4 -1/2
HilbertSeries
```

From the output file we only list the data of the Ehrhart series.

```
multiplicity = 5/8
multiplicity (float) = 0.625

Hilbert series:
```

```
1 0 0 3 2 -1 2 2 1 1 1 1 2
denominator with 3 factors:
1: 1  2: 1  12: 1

degree of Hilbert Series as rational function = -3

Hilbert series with cyclotomic denominator:
-1 -1 -1 -3 -4 -3 -2
cyclotomic denominator:
1: 3  2: 2  3: 1  4: 1

Hilbert quasi-polynomial of period 12:
0:   48 28 15                        7:   23 22 15
1:   11 22 15                        8:   16 28 15
2:  -20 28 15                        9:   27 22 15
3:   39 22 15                        10:   -4 28 15
4:   32 28 15                        11:    7 22 15
5:   -5 22 15                        with common denominator = 48
6:   12 28 15
```

The multiplicity is a rational number. Since in dimension 2 the normalized area (of full-dimensional polytopes) is twice the Euclidean area, we see that P has Euclidean area $5/16$. If the multiplicity is not integral, we also print it in floating point format, This is certainly superfluous for a fraction like $5/8$, but very handy if the numerator and the denominator have many digits.

Unlike in the case of a lattice polytope, there is no canonical choice of the denominator of the Ehrhart series. Normaliz gives it in 2 forms. In the first form the numerator polynomial is

$$1+3t^3+2t^4-t^5+2t^6+2t^7+t^8+t^9+t^{10}+t^{11}+2t^{12}$$

and the denominator is

$$(1-t)(1-t^2)(1-t^{12}).$$

As a rational function, $H(t)$ has degree -3. This implies that $3P$ is the smallest integral multiple of P that contains a lattice point in its interior.

Normaliz gives also a representation as a quotient of coprime polynomials with the denominator factored into cyclotomic polynomials. In this case we have

$$H(t)=-\frac{1+t+t^2+t^3+4t^4+3t^5+2t^6}{\zeta_1^3\zeta_2^2\zeta_3\zeta_4}$$

where ζ_i is the i-th cyclotomic polynomial ($\zeta_1=t-1$, $\zeta_2=t+1$, $\zeta_3=t^2+t+1$, $\zeta_4=t^2+1$).

Normaliz transforms the representation with cyclotomic denominator into one with denominator of type $(1-t^{e_1})\cdots(1-t^{e_r})$, $r=$ rank, by choosing e_r as the least common multiple of all the orders of the cyclotomic polynomials appearing, e_{r-1} as the lcm of those orders that have multiplicity ≥ 2 etc.

There are other ways to form a suitable denominator with 3 factors $1-t^e$, for example $g(t) = (1-t^2)(1-t^3)(1-t^4) = -\zeta_1^3\zeta_2^2\zeta_3\zeta_4$. Of course, $g(t)$ is the optimal choice in this case. However, P is a simplex, and in general such optimal choice may not exist. We will explain the reason for our standardization below.

Let $p(k)$ be the number of lattice points in kP. Then $p(k)$ is a quasipolynomial:

$$p(k) = p_0(k) + p_1(k)k + \cdots + p_{r-1}(k)k^{r-1},$$

where the coefficients depend on k, but only to the extent that they are periodic of a certain period $\pi \in \mathbb{N}$. In our case $\pi = 12$ (the lcm of the orders of the cyclotomic polynomials).

The table giving the quasipolynomial is to be read as follows: The first column denotes the residue class j modulo the period and the corresponding line lists the coefficients $p_i(j)$ in ascending order of i, multiplied by the common denominator. So

$$p(k) = 1 + \frac{7}{12}k + \frac{5}{16}k^2, \qquad k \equiv 0 \pmod{12},$$

etc. The leading coefficient is the same for all residue classes and equals the Euclidean volume (in this case).

Our choice of denominator for the Hilbert series is motivated by the following fact: e_i is the common period of the coefficients $p_{r-i}, \ldots, p_{r-1}$. The user should prove this fact or at least verify it by several examples.

Especially in the case of a simplex the representation of the Hilbert series shown so far may not be the expected one. In fact, there is a representation in which the exponents of t in the denominator are the degrees of the integral extreme generators. So one would expect the denominator to be $(1-t^2)(1-t^3)(1-t^4)$ in our case. The generalization to the nonsimplicial case uses the degrees of a homogeneous system of parameters (see [5, p. 200]). Normaliz can compute such a denominator if the computation goal `HSOP` is set (`rationalHSOP.in`):

```
Hilbert series (HSOP):
1 1 1 3 4 3 2
denominator with 3 factors:
2: 1  3: 1  4: 1
```

Note that the degrees of the elements in a homogeneous system of parameters are by no means unique and that there is no optimal choice in general. To find a suitable sequence of degrees Normaliz must compute the face lattice of the cone to some extent. Therefore be careful not to ask for `HSOP` if the cone has many support hyperplanes.

2.5.1. The Ehrhart series with `vertices`?

It is tempting to define the polytope by the input type `vertices`. This choice makes the compuation inhomogeneous, a mode that is mainly meant for (potentially) unbounded polyhedra. But it can be used for polytopes as well, and with this input type you can compute aall of the data that we have seen above. You must ask for the `EhrhartSeries` instead of the `HilbertSeries`. The file `rational_inhom.in` is

```
amb_space 2
vertices 3
1/2 1/2 1
-1/3 -1/3 1
1/4 -1/2 1
EhrhartSeries
```

Nevertheless, there is also use for `HilbertSeries` in the inhomogeneous case. But then the grading must be defined on the affine space of the polytope (and not on the cone over the polytope). See Sections 6.1 and 6.10.2.

2.5.2. The rational polytope by inequalities

We extract the support hyperplanes of our polytope from the output file and use them as input (`poly_ineq.in`):

```
amb_space 3
inequalities 3
-8  2 3
1 -1 0
2  7 3
grading
unit_vector 3
HilbertSeries
```

At this point we have to help Normaliz because it has no way to guess that we want to investigate the polytope defined by the inequalities and the choice $x_3 = 1$. This is achieved by the specification of the grading that maps every vector to its third coordinate.

This is the first time that we used the shortcut `unit_vector <n>` which represents the nth unit vector $e_n \in \mathbb{R}^d$ and is only allowed for input types which require a single vector.

These data tell us that the polytope, as a subset of $\mathbb{R}^2$, is defined by the inequalities

$$\begin{aligned} -8x_1 + 2x_2 + 3 &\geq 0, \\ x_1 - x_2 + 0 &\geq 0, \\ 2x_1 + 7x_2 + 3 &\geq 0. \end{aligned}$$

These inequalities are inhomogeneous, but we are using the homogeneous input type `inequalities` which amounts to introducing the grading variable x_3 as explained above.

The inequalities as written above look somewhat artificial. It is certainly more natural to write them in the form

$$\begin{aligned} 8x_1 - 2x_2 &\leq 3, \\ x_1 - x_2 &\geq 0, \\ 2x_1 + 7x_2 &\geq -3. \end{aligned}$$

and for the direct transformation into Normaliz input we have introduced the type `hom_constraints`. The prefix `hom` indicates that we want homogeneous inequalities whereas plain `constraints` that we have already seen in Section 2.3.3 give inhomogeneous inequalities. The file `poly_hom_const.in` contains

```
amb_space 3
hom_constraints 3
8 -2 <= 3
1 -1 >= 0
2 7  >= -3
grading
unit_vector 3
HilbertSeries
```

You can of course also switch to inhomogeneous input using `inhom_inequalities` or `constrints` in the same way as `polytope` can be replaced by `vertices`.

2.6. Magic squares

Suppose that you are interested in the following type of "square"

x_1	x_2	x_3
x_4	x_5	x_6
x_7	x_8	x_9

and the problem is to find nonnegative values for $x_1,\dots,x_9$ such that the 3 numbers in all rows, all columns, and both diagonals sum to the same constant $\mathscr{M}$. Sometimes such squares are called *magic* and $\mathscr{M}$ is the *magic constant*. This leads to a linear system of equations

$$\begin{aligned}
x_1+x_2+x_3 &= x_4+x_5+x_6;\\
x_1+x_2+x_3 &= x_7+x_8+x_9;\\
x_1+x_2+x_3 &= x_1+x_4+x_7;\\
x_1+x_2+x_3 &= x_2+x_5+x_8;\\
x_1+x_2+x_3 &= x_3+x_6+x_9;\\
x_1+x_2+x_3 &= x_1+x_5+x_9;\\
x_1+x_2+x_3 &= x_3+x_5+x_7.
\end{aligned}$$

This system is encoded in the file `3x3magic.in`:

```
amb_space 9
equations 7
1 1 1 -1 -1 -1  0  0  0
1 1 1  0  0  0 -1 -1 -1
0 1 1 -1  0  0 -1  0  0
```

```
1 0 1  0 -1  0  0 -1  0
1 1 0  0  0 -1  0  0 -1
0 1 1  0 -1  0  0  0 -1
1 1 0  0 -1  0 -1  0  0
grading
sparse 1:1 2:1 3:1;
```

The input type `equations` represents *homogeneous* equations. The first equation reads

$$x_1 + x_2 + x_3 - x_4 - x_5 - x_6 = 0,$$

and the other equations are to be interpreted analogously. The magic constant is a natural choice for the grading. It is given in sparse form, equivalent to the dense form

```
grading
1 1 1 0 0 0 0 0 0
```

It seems that we have forgotten to define the cone. This may indeed be the case, but doesn't matter: if there is no input type that defines a cone, Normaliz chooses the positive orthant, and this is exactly what we want in this case.

The output file contains the following:

```
5 Hilbert basis elements
5 lattice points in polytope (Hilbert basis elements of degree 1)
4 extreme rays
4 support hyperplanes

embedding dimension = 9
rank = 3
external index = 1

size of triangulation   = 2
resulting sum of |det|s = 4

grading:
1 1 1 0 0 0 0 0 0
with denominator = 3
```

The input degree is the magic constant. However, as the denominator 3 shows, the magic constant is always divisible by 3, and therefore the effective degree is $\mathscr{M}/3$. This degree is used for the multiplicity, the Hilbert series, and the Hilbert basis elements of degree 1, and other date deopending on the degree.

By introducing the grading denominator, Normaliz has changed the grading defined by you, and you may not like this. There is a way out: add the option `NoGradingDenom`. We will fdiscuss the consequneces below.

```
degrees of extreme rays:
1: 4
```

```
Hilbert basis elements are of degree 1
```

This was not to be expected (and is no longer true for 4×4 squares).

```
multiplicity = 4

Hilbert series:
1 2 1
denominator with 3 factors:
1: 3

degree of Hilbert Series as rational function = -1

Hilbert polynomial:
1 2 2
with common denominator = 1
```

The Hilbert series is

$$\frac{1+2t+t^2}{(1-t)^3}.$$

The Hilbert polynomial is

$$P(k) = 1+2k+2k^2,$$

and after substituting $\mathscr{M}/3$ for k we obtain the number of magic squares of magic constant $\mathscr{M}$, provided 3 divides $\mathscr{M}$. (If $3 \nmid \mathscr{M}$, there is no magic square of magic constant $\mathscr{M}$.)

```
rank of class group = 1
finite cyclic summands:
2: 2
```

So the class group is $\mathbb{Z} \oplus (\mathbb{Z}/2\mathbb{Z})^2$.

```
5 lattice points in polytope (Hilbert basis elements of degree 1):
0 2 1 2 1 0 1 0 2
1 0 2 2 1 0 0 2 1
1 1 1 1 1 1 1 1 1
1 2 0 0 1 2 2 0 1
2 0 1 0 1 2 1 2 0

0 further Hilbert basis elements of higher degree:
```

The 5 elements of the Hilbert basis represent the magic squares

2	0	1
0	1	2
1	2	0

1	0	2
2	1	0
0	2	1

1	1	1
1	1	1
1	1	1

1	2	0
0	1	2
2	0	1

0	2	1
2	1	0
1	0	2

All other solutions are linear combinations of these squares with nonnegative integer coefficients. One of these 5 squares is clearly in the interior:

```
4 extreme rays:                 4 support hyperplanes:
0 2 1 2 1 0 1 0 2               -2 -1 0 0  4 0 0 0 0
1 0 2 2 1 0 0 2 1               0 -1 0 0  2 0 0 0 0
1 2 0 0 1 2 2 0 1               0  1 0 0  0 0 0 0 0
2 0 1 0 1 2 1 2 0               2  1 0 0 -2 0 0 0 0
```

These 4 support hyperplanes cut out the cone generated by the magic squares from the linear subspace they generate. Only one is reproduced as a sign inequality. This is due to the fact that the linear subspace has submaximal dimension and there is no unique lifting of linear forms to the full space.

```
6 equations:                3 basis elements of generated lattice:
1 0 0 0 0  1 -2 -1  1       1 0 -1 -2 0  2  1  0 -1
0 1 0 0 0  1 -2  0  0       0 1 -1 -1 0  1  1 -1  0
0 0 1 0 0  1 -1 -1  0       0 0  3  4 1 -2 -1  2  2
0 0 0 1 0 -1  2  0 -2
0 0 0 0 1 -1  1  0 -1
0 0 0 0 0  3 -4 -1  2
```

So one of our equations has turned out to be superfluous (why?). Note that also the equations are not reproduced exactly. Finally, Normaliz lists a basis of the efficient lattice $\mathbb{E}$ generated by the magic squares.

Note that the equations and the lattice basis are not uniquely determined. We transform their matrices into reduced row echelon form to force unique output files.

2.6.1. Blocking the grading denominator

As mentioned above, one can block the grading denominator and force Normaliz to use the input grading. For the magic squares we augment the input file as follows (`3x3magicNGD.in`):

```
amb_space 9
equations 7
1 1 1 -1 -1 -1  0  0  0
...
1 1 0  0 -1  0 -1  0  0
grading
sparse 1:1 2:1 3:1;
NoGradingDenom
```

The consequences:

```
grading:
1 1 1 0 0 0 0 0 0
```

```
degrees of extreme rays:
3: 4

multiplicity = 4/9
multiplicity (float) = 0.444444444444

Hilbert series:
1 0 0 2 0 0 1
denominator with 3 factors:
3: 3

degree of Hilbert Series as rational function = -3

The numerator of the Hilbert series is symmetric.

Hilbert series with cyclotomic denominator:
-1 0 0 -2 0 0 -1
cyclotomic denominator:
1: 3  3: 3

Hilbert quasi-polynomial of period 3:
0:  9 6 2
1:  0 0 0
2:  0 0 0
with common denominator = 9

rank of class group = 1
finite cyclic summands:
2: 2

**************************************************************************

0 lattice points in polytope (Hilbert basis elements of degree 1):
```

It is easy to relate the data with the grading denominator to those without. You must decide yourself what you prefer. One aspect is whether one prefers intrinsic data (with grading denominator) to extrinsic ones that depend on the embedding (without the grading denominator). We will discuss the topic again in Section 6.1.

2.6.2. With even corners

We change our definition of magic square by requiring that the entries in the 4 corners are all even. Then we have to augment the input file by the following (`3x3magiceven.in`):

```
congruences 4 sparse
1:1 10:2;
```

```
3:1 10:2;
7:1 10:2;
9:1 10:2;
```

This sparse form is equivalent to the dense form

```
congruences 4
1 0 0 0 0 0 0 0 0 2
0 0 1 0 0 0 0 0 0 2
0 0 0 0 0 0 1 0 0 2
0 0 0 0 0 0 0 0 1 2
```

The first 9 entries in each row represent the coefficients of the coordinates in the homogeneous congruences, and the last is the modulus:

$$x_1 \equiv 0 \mod 2$$

is the first congruence etc.

We could also define these congruences as symbolic constraints:

```
constraints 4 symbolic
x[1] ~ 0(2);
x[3] ~ 0(2);
x[7] ~ 0(2);
x[9] ~ 0(2);
```

The output changes accordingly:

```
9 Hilbert basis elements
0 lattice points in polytope (Hilbert basis elements of degree 1)
4 extreme rays
4 support hyperplanes

embedding dimension = 9
rank = 3
external index = 4

size of triangulation   = 2
resulting sum of |det|s = 8

grading:
1 1 1 0 0 0 0 0 0
with denominator = 3

degrees of extreme rays:
2: 4

multiplicity = 1
```

```
Hilbert series:
1 -1 3 1
denominator with 3 factors:
1: 1  2: 2

degree of Hilbert Series as rational function = -2

Hilbert series with cyclotomic denominator:
-1 1 -3 -1
cyclotomic denominator:
1: 3  2: 2

Hilbert quasi-polynomial of period 2:
0:   2 2 1
1:  -1 0 1
with common denominator = 2
```

After the extensive discussion in Section 2.5 it should be easy for you to write down the Hilbert series and the Hilbert quasipolynomial. (But keep in mind that the grading has a denominator.)

```
rank of class group = 1
finite cyclic summands:
4: 2

**********************************************************************

0 lattice points in polytope (Hilbert basis elements of degree 1):

9 further Hilbert basis elements of higher degree:
...

4 extreme rays:
0 4 2 4 2 0 2 0 4
2 0 4 4 2 0 0 4 2
2 4 0 0 2 4 4 0 2
4 0 2 0 2 4 2 4 0
```

We have listed the extreme rays since they have changed after the introduction of the congruences, although the cone has not changed. The reason is that Normaliz always chooses the extreme rays from the efficient lattice $\mathbb{E}$.

```
4 support hyperplanes:
...

6 equations:
...                              3 basis elements of generated lattice:
```

```
                                  1 0 -1 -2 0  2  1  0 -1
2 congruences:                    0 1 -1 -1 0  1  1 -1  0
1 0 0 0 0 0 0 0 0 2               0 0  3  4 1 -2 -1  2  2
0 1 0 0 1 0 0 0 0 2
```

The rank of the lattice has of course not changed, but after the introduction of the congruences the basis has changed.

2.6.3. The lattice as input

It is possible to define the lattice by generators. We demonstrate this for the magic squares with even corners. The lattice has just been computed (`3x3magiceven_lat.in`):

```
amb_space 9
lattice 3
 0  1 2 3 1 -1 0  1 2
 2 -1 2 1 1  1 0  3 0
 0  3 0 1 1  1 2 -1 2
grading
1 1 1 0 0 0 0 0 0
```

It produces the same output as the version starting from equations and congruences.

`lattice` has a variant that takes the saturation of the sublattice generated by the input vectors (`3x3magic_sat.in`):

```
amb_space 9
saturation 3
 0  1 2 3 1 -1 0  1 2
 2 -1 2 1 1  1 0  3 0
 0  3 0 1 1  1 2 -1 2
grading
1 1 1 0 0 0 0 0 0
```

Clearly, we remove the congruences by this choice and arrive at the output of `3x3magic.in`.

2.7. Decomposition in a numerical semigroup

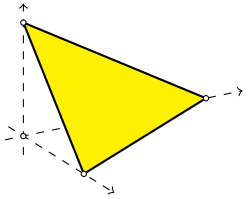
Let $S = \langle 6, 10, 15 \rangle$, the numerical semigroup generated by $6, 10, 15$. How can 97 be written as a sum in the generators?

In other words: we want to find all nonnegative integral solutions to the equation

$$6x_1 + 10x_2 + 15x_3 = 97.$$

Input (`NumSemi.in`):

```
amb_space 3
constraints 1 symbolic
6x[1] + 10x[2] + 15x[3] = 97;
```



The equation cuts out a triangle from the positive orthant.

The set of solutions is a module over the monoid M of solutions of the homogeneous equation $6x_1 + 10x_2 + 15x_3 = 0$. So $M = 0$ in this case.

```
6 lattice points in polytope (module generators):
2 1 5 1
2 4 3 1
2 7 1 1
7 1 3 1
7 4 1 1
12 1 1 1

0 Hilbert basis elements of recession monoid:
```

The last line is as expected, and the 6 lattice popints (or module generators) are the goal of the computation.

Normaliz is smart enough to recognize that it must compute the lattice points in a polygon, and does exactly this. You can recognize it in the console output: Normaliz 3.6.1 has used the project-and-lift algorithm. We will discuss it further in Section 2.13 and Section 6.2.1.

For those who like to play: add the option `--NoProjection` to the command line. Then the terminal output will change; Normaliz computes the lattice points as a truncated Hilbert basis via a triangulation (only one simplicial cone in this case).

2.8. A job for the dual algorithm

We increase the size of the magic squares to 5×5. Normaliz can do the same computation as for 3×3 squares, but this will take some minutes. Suppose we are only interested in the Hilbert basis, we should use the dual algorithm for this example. (The dual algorithm goes back to Pottier [17].) The input file is `5x5dual.in`:

```
amb_space 25
equations 11
1 1 1 1 1 -1 -1 -1 -1 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
...
1 1 1 1 0  0  0  0 -1  0  0  0 -1  0  0  0 -1  0  0  0 -1  0  0  0  0
grading
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
HilbertBasis
```

The input file does not say anything about the dual algorithm mentioned in the section title. With this input it is chosen automatically. See Section 6.5 for a discussion of when this happens. But you can insist on the dual algorithm by adding `DualMode` to the input (or `-d` to the command line). Or, if you want to compare it to the primal algorithm add `PrimalMode` (or `-P` to the command line).

The Hilbert basis contains 4828 elements, too many to be listed here.

With the file `5x5.in` you can compute the Hilbert basis and the Hilbert series, and the latter with HSOP:

```
Hilbert series (HSOP):
1 15 356 4692 36324 198467 ... 198467 36324 4692 356 15 1
denominator with 15 factors:
1: 5  2: 3  6: 2  12: 1  60: 2  420: 1  1260: 1

degree of Hilbert Series as rational function = -5

The numerator of the Hilbert Series is symmetric.
```

In view of the length of the numerator of the Hilbert series it may be difficult to observe the symmetry. So Normaliz does it for you. The symmetry shows that the monoid is Gorenstein, but if you are only interested in the Gorenstein property, there is a much faster way to check it (see Section 6.7).

The size 6×6 is out of reach for the Hilbert series, but the Hilbert basis can be computed (in the automatically chosen dual mode). It takes some hours.

2.9. A dull polyhedron

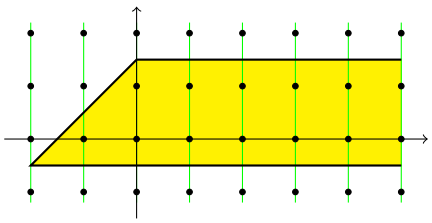
We want to compute the polyhedron defined by the inequalities

$$\begin{aligned} \xi_2 &\geq -1/2, \\ \xi_2 &\leq 3/2, \\ \xi_2 &\leq \xi_1 + 3/2. \end{aligned}$$

They are contained in the input file `InhomIneq.in`:

```
amb_space 2
constraints 3
 0 1 >= -1/2
 0 1 <=  3/2
-1 1 <=  3/2
grading
unit_vector 1
```

The grading says that we want to count points by the first coordinate, namely along the green kines:



It yields the output

```
2 module generators
1 Hilbert basis elements of recession monoid
2 vertices of polyhedron
1 extreme rays of recession cone
3 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 1

size of triangulation   = 1
resulting sum of |det|s = 8

dehomogenization:
0 0 1

grading:
1 0 0
```

The interpretation of the grading requires some care in the inhomogeneous case. We have extended the input grading vector by an entry 0 to match the embedding dimension. For the computation of the degrees of *lattice points* in the ambient space you can either use only the first 2 coordinates or take the full scalar product of the point in homogenized coordinates and the extended grading vector.

```
module rank = 2
multiplicity = 2
```

The module rank is 2 in this case since we have two "layers" in the solution module that are parallel to the recession monoid. This is of course also reflected in the Hilbert series.

```
Hilbert series:
1 1
denominator with 1 factors:
1: 1

shift = -1
```

We haven't seen a shift yet. It is always printed (necessarily) if the Hilbert series does not start

in degree 0. In our case it starts in degree -1 as indicated by the shift -1. We thus get the Hilbert series

$$t^{-1}\,\frac{t+t}{1-t} = \frac{t^{-1}+1}{1-t}.$$

Note: We used the opposite convention for the shift in Normaliz 2.

Note that the Hilbert (quasi)polynomial is always computed for the unshifted monoid defined by the input data. (This was different in previous versions of Normaliz.)

```
degree of Hilbert Series as rational function = -1

Hilbert polynomial:
2
with common denominator = 1

********************************************************************

2 module generators:
-1 0 1
0 1 1

1 Hilbert basis elements of recession monoid:
1 0 0

2 vertices of polyhedron:
-4 -1 2
0  3 2

1 extreme rays of recession cone:
1 0 0

3 support hyperplanes of polyhedron (homogenized):
0 -2 3
0  2 1
2 -2 3
```

The dual algorithm that was used in Section 2.8 can also be applied to inhomogeneous computations. We would of course loose the Hilbert series. In certain cases it may be preferable to suppress the computation of the vertices of the polyhedron if you are only interested in the integer points; see Section 4.6.

2.9.1. Defining it by generators

If the polyhedron is given by its vertices and the recession cone, we can define it by these data (`InhomIneq_gen.in`):

```
amb_space 2
vertices 2
-4 -1 2
0  3 2
cone 1
1 0
grading
unit_vector 1
```

The output is identical to the version starting from the inequalities.

2.10. The Condorcet paradox

In social choice elections each of the k voters picks a linear preference order of the n candidates. There are $n!$ such orders. The election result is the vector $(x_1,\dots,x_N)$,N=n!, in which x_i is the number of voters that have chosen the i-th preference order in, say, lexicographic enumeration of these orders. In the following we assume the *impartial anonymous culture* according to which every preference order has the same basic weight of $1/n!$.

We say that candidate A *beats* candidate B if the majority of the voters prefers A to B. As the Marquis de *Condorcet* (and others) observed, "beats" is not transitive, and an election may exhibit the *Condorcet paradox*: there is no Condorcet winner. (See [11] and the references given there for more information.)

We want to find the probability for $k\to\infty$ that there is a Condorcet winner for $n=4$ candidates. The event that A is the Condorcet winner can be expressed by linear inequalities on the election outcome (a point in 24-space). The wanted probability is the lattice normalized volume of the polytope cut out by the inequalities at $k=1$. The file `Condorcet.in`:

```
amb_space 24
inequalities 3
1 1 1 1 1 1 -1 -1 -1 -1 -1 -1   1  1 -1 -1  1 -1   1  1 -1 -1  1 -1
1 1 1 1 1 1  1  1 -1 -1  1 -1  -1 -1 -1 -1 -1 -1   1  1  1 -1 -1 -1
1 1 1 1 1 1  1  1  1 -1 -1 -1   1  1  1 -1 -1 -1  -1 -1 -1 -1 -1 -1
nonnegative
total_degree
Multiplicity
```

The first inequality expresses that A beats B, the second and the third say that A beats C and D. (So far we do not exclude ties, and they need not be excluded for probabilities as $k\to\infty$.)

In addition to these inequalities we must restrict all variables to nonnegative values, and this is achieved by adding the attribute `nonnegative`. The grading is set by `total_degree`. It replaces the grading vector with 24 entries 1. Finally `Multiplicity` sets the computation goal.

From the output file we only mention the quantity we are out for:

```
multiplicity = 1717/8192
multiplicity (float) = 0.209594726562
```

Since there are 4 candidates, the probability for the existence of a Condorcet winner is $1717/2048 = 0.209595$.

We can refine the information on the Condorcet paradox by computing the Hilbert series. Either we delete `Multiplicity` from the input file or, better, we add `--HilbertSeries` (or simply `-q`) on the command line. The result:

```
Hilbert series:
1 5 133 363 4581 8655 69821 100915 ... 12346 890 481 15 6
denominator with 24 factors:
1: 1  2: 14  4: 9

degree of Hilbert Series as rational function = -25
```

If your executable of Normaliz was built with CoCoALib (ee Section 10), for example the executablse for Linux or Mac OS from our distribution or in the Docker image, it uses symmetrization for the computation of the Hilbert series. If not, then simply disregard any remark on symmetrization. Everything runs very quickly also without it.

If symmetrization has been used, you will also find a file `Condorcet.symm.out` in your directory. It contains the data computed for the symmetrization. You need not care at this point. We take continue the discussion of symmetrization in Section 6.8.

2.10.1. Excluding ties

Now we are more ambitious and want to compute the Hilbert series for the Condorcet paradox, or more precisely, the number of election outcomes having A as the Condorcet winner depending on the number k of voters. Moreover, as it is customary in social choice theory, we want to exclude ties. The input file changes to `CondorcetSemi.in`:

```
amb_space 24
excluded_faces 3
1 1 1 1 1 1 -1 -1 -1 -1 -1 -1   1  1 -1 -1  1 -1   1  1 -1 -1  1 -1
1 1 1 1 1 1  1  1 -1 -1  1 -1  -1 -1 -1 -1 -1 -1   1  1  1 -1 -1 -1
1 1 1 1 1 1  1  1  1 -1 -1 -1   1  1  1 -1 -1 -1  -1 -1 -1 -1 -1 -1
nonnegative
total_degree
HilbertSeries
```

We could omit `HilbertSeries`, and the computation would include the Hilbert basis. The type `excluded_faces` only affects the Hilbert series. In every other respect it is equivalent to `inequalities`.

From the file `CondorcetSemi.out` we only display the Hilbert series:

```
Hilbert series:
6 15 481 890 12346 ... 100915 69821 8655 4581 363 133 5 1
denominator with 24 factors:
1: 1  2: 14  4: 9

shift = 1

degree of Hilbert Series as rational function = -24
```

Surprisingly, this looks like the Hilbert series in the previous section read backwards, roughly speaking. This is true, and one can explain it as we will see below.

It is justified to ask why we don't use `strict_inequalities` instead of `excluded_faces`. It does of course give the same Hilbert series. However, Normaliz cannot (yet) apply symmetrization in inhomogeneous computations. Moreover, the algorithmic approach is different, and according to our experience `excluded_faces` is more efficient, independently of symmetrization.

2.10.2. At least one vote for every preference order

Suppose we are only interested in elections in which every preference order is chosen by at least one voter. This can be modeled as follows (`Condorcet_one.in`):

```
amb_space 24
inequalities 3
1 1 1 1 1 1 -1 -1 -1 -1 -1 -1   1  1 -1 -1  1 -1   1  1 -1 -1  1 -1
1 1 1 1 1 1  1  1 -1 -1  1 -1  -1 -1 -1 -1 -1 -1   1  1  1 -1 -1 -1
1 1 1 1 1 1  1  1  1 -1 -1 -1   1  1  1 -1 -1 -1  -1 -1 -1 -1 -1 -1
strict_signs
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
total_degree
HilbertSeries
```

The entry 1 at position i of the vector `strict_signs` imposes the inequality $x_i \geq 1$. A -1 would impose the inequality $x_i \leq -1$, and the entry 0 imposes no condition on the i-th coordinate.

```
Hilbert series:
1 5 133 363 4581 8655 69821 100915 ... 12346 890 481 15 6
denominator with 24 factors:
1: 1  2: 14  4: 9

shift = 24

degree of Hilbert Series as rational function = -1
```

Again we encounter (almost) the Hilbert series of the Condorcet paradox (without side con-

ditions). It is time to explain this coincidence. Let C be the Condorcet cone defined by the nonstrict inequalities, M the monoid of lattice points in it, $I_1 \subset M$ the ideal of lattice points avoiding the 3 facets defined by ties, I_2 the ideal of lattice points with strictly positive coordinates, and finally I_3 the ideal of lattice points in the interior of C. Moreover, let $\mathbb{1} \in \mathbb{Z}^{24}$ be the vector with all entries 1.

Since $\mathbb{1}$ lies in the three facets defining the ties, it follows that $I_2 = M + \mathbb{1}$. This explains why we obtain the Hilbert series of I_2 by multiplying the Hilbert series of M by t^{24}, as just observed. Generalized Ehrhart reciprocity (see [5, Theorem 6.70]) then explains the Hilbert series of I_1 that we observed in the previous section. Finally, the Hilbert series of I_3 that we don't have displayed is obtained from that of M by "ordinary" Ehrhart reciprocity. But we can also obtain I_1 from I_3: $I_1 = I_3 - \mathbb{1}$, and generalized reciprocity follows from ordinary reciprocity in this very special case. (Also see W. Bruns and B. Ichim, "Computations of volumes and Ehrhart series in four candidates elections", `arXiv:1704.00153`.)

The essential point in these arguments (apart from reciprocity) is that $\mathbb{1}$ lies in all support hyperplanes of C except the coordinate hyperplanes.

You can easily compute the Hilbert series of I_3 by making all inequalities strict.

As the terminal output shows, symmetrization has not been applied for the reason mentioned above: `strict_signs` is an inhomogeneous input type. It would of course be possible to encode the strict signs as `excluded_faces`. Then the sparse format of matrices is very handy:

```
excluded_faces 24
1:1;
1:2;
...
1:24;
```

This is a shorthand for the unit matrix.

2.11. Testing normality

We want to test the monoid $A_{4\times4\times3}$ defined by $4 \times 4 \times 3$ contingency tables for normality (see [7] for the background). The input file is `A443.in`:

```
amb_space 40
cone_and_lattice 48
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
HilbertBasis
```

Why `cone_and_lattice`? Well, we want to find out whether the monoid is normal, i.e., whether $M = C(M) \cap \mathrm{gp}(M)$. If M is even integrally closed in $\mathbb{Z}^{24}$, then it is certainly integrally closed in the evidently smaller lattice $\mathrm{gp}(M)$, but the converse does not hold in general, and therefore we work with the lattice generated by the monoid generators.

It turns out that the monoid is indeed normal:

```
original monoid is integrally closed
```

Actually the output file reveals that M is even integrally closed in $\mathbb{Z}^{24}$: the external index is 1, and therefore $\mathrm{gp}(M)$ is integrally closed in $\mathbb{Z}^{24}$.

The output files also shows that there is a grading on $\mathbb{Z}^{24}$ under which all our generators have degree 1. We could have seen this ourselves: Every generator has exactly one entry 1 in the first 16 coordinates. (This is clear from the construction of M.)

A noteworthy detail from the output file:

```
size of partial triangulation   = 48
```

It shows that Normaliz uses only a partial triangulation in Hilbert basis computations; see [7].

It is no problem to compute the Hilbert series as well if you are interested in it. Simply add `-q` to the command line or remove `HilbertBasis` from the input file. Then a full triangulation is needed (size $2,654,272$).

Similar examples are `A543`, `A553` and `A643`. The latter is not normal, as we will see below. Even on a standard PC or laptop, the Hilbert basis computation does not take very long because Normaliz uses only a partial triangulation. The Hilbert series can still be determined, but the computation time will grow considerably since the it requires a full triangulation. See [8] for timings.

2.11.1. Computing just a witness

If the Hilbert basis is large and there are many support hyperplanes, memory can become an issue for Normaliz, as well as computation time. Often one is only interested in deciding whether the given monoid is integrally closed (or normal). In the negative case it is enough to find a single element that is not in the original monoid – a witness disproving integral closedness. As soon as such a witness is found, Normaliz stops the Hilbert basis computation (but will continue to compute other data if they are asked for). We look at the example `A643.in` (for which the full Hilbert basis is not really a problem):

```
amb_space 54
cone_and_lattice 72
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 ...
...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 ...
IsIntegrallyClosed
```

Don't add `HilbertBasis` because it will overrule `IsIntegrallyClosed`!

The output:

```
72 extreme rays
153858 support hyperplanes
```

```
embedding dimension = 54
rank = 42
external index = 1
internal index = 1
original monoid is not integrally closed
witness for not being integrally closed:
0 0 1 0 1 1 1 1 0 0 1 0 0 1 0 1 0 1 1 0 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 ...

grading:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 ...

degrees of extreme rays:
1: 72

**********************************************************************

72 extreme rays:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 ...
...
```

If you repeat such a computation, you may very well get a different witness if several parallel threads find witnesses. Only one of them is delivered.

2.12. Convex hull computation/vertex enumeration

Normaliz computes convex hulls as should be very clear by now, and the only purpose of this section is to emphasize that Normaliz can be restricted to this task by setting an explicit computation goal. By convex hull computation we mean the determination of the support hyperplanes of a polyhedron is given by generators (or vertices). The converse operation is vertex enumeration. Both amount to the dualization of a cone, and can therefore be done by the same algorithm.

As an example we take the input file `cyclicpolytope30-15.in`, the cyclic polytope of dimension 15 with 30 vertices (suggested by D. Avis and Ch. Jordan):

```
/* cyclic polytope of dimension 15 with 30 vertices */
amb_space 16
polytope 30
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768
...
30 900 27000 810000 ... 478296900000000000000 1434890700000000000000
SupportHyperplanes
```

Already the entries of the vertices show that the computation cannot be done in 64 bit arithmetic. But you need not be worried. Just start Normaliz as usual. It will simply switch to

infinite precision by itself, as shown by the terminal output (use the option `-c` or `--Verbose`).

```
                                                      \.....|
                  Normaliz 3.2.0                       \....|
                                                        \...|
    (C) The Normaliz Team, University of Osnabrueck      \..|
                  January  2017                           \.|
                                                           \|
************************************************************
Compute: SupportHyperplanes
Could not convert 15181127029874798299.
Arithmetic Overflow detected, try a bigger integer type!
Restarting with a bigger type.
************************************************************
starting primal algorithm (only support hyperplanes) ...
Generators sorted lexicographically
Start simplex 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
gen=17, 72 hyp
gen=18, 240 hyp
gen=19, 660 hyp
gen=20, 1584 hyp
gen=21, 3432 hyp
gen=22, 6864 hyp
gen=23, 12870 hyp
gen=24, 22880 hyp
gen=25, 38896 hyp
gen=26, 63648 hyp
gen=27, 100776 hyp
gen=28, 155040 hyp
gen=29, 232560 hyp
gen=30, 341088 hyp
Pointed since graded
Select extreme rays via comparison ... done.
------------------------------------------------------------
transforming data... done.
```

Have a look at the output file if you are not afraid of 341088 linear forms.

If you have looked closely at the terminal output above, you should have stumbled on the lines

```
Could not convert 15181127029874798299.
Arithmetic Overflow detected, try a bigger integer type!
```

They show that Normaliz has tried the computation in 64 bit integers, but encountered a number that is too large for this precision. It has automatically switched to infinite precision. (See Section 4.4 for more information on integer types.)

2.13. Lattice points in a polytope and its Euclidean volume

The computation of lattice points in a polytope can be viewed as a truncated Hilbert basis computation, and we have seen in preceding examples. But Normaliz can be restricted to their computation, with homogeneous as well as with inhomogeneous input. Let us look at `ChF_8_1024.in`:

```
amb_space 8
constraints 16
0.10976576 0.2153132834 ... 0.04282847494 >= -1/2
...
0.10976576 -0.2153132834 ... -0.04282847494 >= -1/2
0.10976576 0.2153132834 ... 0.04282847494 <= 1/2
0.10976576 -0.2153132834 ...-0.04282847494 <= 1/2
LatticePoints
ProjectionFloat
```

This example comes from numerical analysis; see Ch. Kacwin, J. Oettershagen and T. Ullrich, On the orthogonality of the Chebyshev-Frolov lattice and applications, Monatsh. Math. 184 (2017), 425–441). Its origin explains the decimal fractions in the input. Normaliz converts them immediately into ordinary fractions of type numerator/denominator, and then makes the input integral as usual.

In the output file you can see to what integer vectors Normaliz has converted the inequalities of the input file:

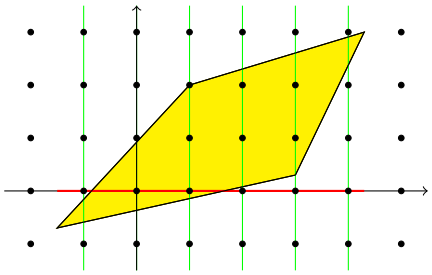
```
16 support hyperplanes of polyhedron (homogenized):
  5488288000  10765664170  ...  2141423747 25000000000
  ...
 -5488288000  10765664170 ...   2141423747 25000000000
```

The option `ProjectionFloat` indicates that we want to compute the lattice points in the polytope defined by the inequalities and that we want to use the floating point variant of the project-and-lift algorithm; `Projection` would make Normaliz use its ordinary arithmetic in this algorithm. For our example the difference in time is not really significant, but when you try `VdM_16_1048576.in`, it becomes very noticeable. Let us have a look at the relevant part of then terminal output:

```
Polyhedron is parallelotope
Computing lattice points by project-and-lift
LLL based on support hyperplanes
Projection
embdim 9 inequalities 16
embdim 8 inequalities 56
embdim 7 inequalities 112
embdim 6 inequalities 140
embdim 5 inequalities 112
embdim 4 inequalities 56
```

```
embdim 3 inequalities 16
embdim 2 inequalities 2
Lifting
embdim 2 Deg1Elements 5
embdim 3 Deg1Elements 21
embdim 4 Deg1Elements 73
embdim 5 Deg1Elements 195
embdim 6 Deg1Elements 365
embdim 7 Deg1Elements 629
embdim 8 Deg1Elements 907
embdim 9 Deg1Elements 1067
Project-and-lift complete
```

We start with embedding dimension 9 since we need a homogenizing coordinate in inhomogeneous computations. Then the polytope is successively projected onto a coordinate hyperplane until we reach a line segment given by 2 inequalities. In the second part Normaliz lifts the lattice points back through all projections. The following figure illustrates the procedure for a polygon that is projected to a line segment.



The green lines show the fibers over the lattice points in the (red) line segment. Note that not every lattice point in the projection must be liftable to a lattice point in the next higher dimension.

In `ChF_8_1024.out` we see

```
1067 lattice points in polytope (module generators):
-4  0  0  0  0  0  0  0 1
-3  0  0  0 -1  0  0  0 1
-3  0  0  0  0  0  0  0 1
...
3  0  0  0  0  0  0  0 1
3  0  0  0  1  0  0  0 1
4  0  0  0  0  0  0  0 1
```

Normaliz finds out that our polytope is in fact a parallelotope. This allows Normaliz to suppress the computation of its vertices. We are not interested in them, and they look frightening when written as ordinary fractions (computed with the additional option `SupportHyperplanes`). This is only the first vertex, the denominator is the number in the last row:

```
256 vertices of polyhedron:
```

```
-7831972155307708173239167258085974255845869779051329651906336771582421875
-2560494334732147696394408175864650673712115229853232268085759500000000000
241193292411744825003604124168323708374286000514244717129567484500000000
-217068228389985295036766378136729994606584469799021447894240025000000000
184601354007762175056223233356965155155965920765943807476092280050000000
-145040353166280163458776558695633828794386588673702458271863175000000000
99905532871877331630351926862909103889365678465423944402406122000000000
-509313990522468215816366827427428831508901797188810249435062450000000000
229248633580316965731682361560246162542228357108960340867209201212984250
...
```

Not all polytopes are parallelotopes, and in most cases Normaliz must compute the vertices or extreme rays as an auxiliary step, even if we are not interested in them. You can always add the option

NoExtRaysOutput

if you want to suppress their output. (The numerical information on the number of extreme rays etc. will however be included in the output file if it is available.) Similarly one can suppress the output of support hyperplanes by

NoSuppHypsOutput

On the other hand, the information provided by the vertices or support hyperplanes may be important. Instead of the unreadable integer output shown above, you can ask for

VerticesFloat

Then the vertices of polyhedra are printed in floating point format:

```
256 vertices of polyhedron:
-3.41637   -1.11691     1.0521  ... 0.435796  -0.222167          1
-3.41637  -0.946868   0.435796  ...  -1.0521   0.632677          1
...
```

Notre that they can only be printed if a polyhedron is defined. This is always the case in inhomogeneous computations, but in the homogeneous case a grading is necessary.

Similarly we can get the support hyperplanes in floating point format (they are only defined up to a positive scalar multiple) by

SuppHypsFloat

resulting in

```
16 support hyperplanes of polyhedron (homogenized):
-0.219532  -0.430627  -0.405641  ...  -0.168022 -0.0856569          1
-0.219532  -0.365068  -0.168022  ...   0.405641    0.24393          1
...
```

By its construction, our polytope should have Euclidean volume 1024. We can confirm this number by computing the volume, using the option

Volume, -V

We get

```
volume (normalized) = 205078125000...00/49670537275735342575...58763
volume (normalized, float) =41287680.0308
volume (Euclidean) = 1024.00000076
```

The result makes us happy, despite of the small inaccuracy of the floating point computation on which the Euclidean volume is based. See Section 6.1.1 for a discussion of volumes and multiplicities.

2.14. The integer hull

The integer hull of a polyhedron P is the convex hull of the set of lattice points in P (despite of its name, it usually does not contain P). Normaliz computes by first finding the lattice points and then computing the convex hull. The computation of the integer hull is requested by the computation goal `IntegerHull`.

The computation is somewhat special since it creates a second cone (and lattice) C_{int}. In homogeneous computations the degree 1 vectors generate C_{int} by an input matrix of type `cone_and_lattice`. In inhomogeneous computations the module generators and the Hilbert basis of the recession cone are combined and generate C_{int}. Therefore the recession cone is reproduced, even if the polyhedron should not contain a lattice point.

The integer hull computation itself is always inhomogeneous. The output file for C_{int} is `<project>.IntHull.out`.

As a very simple example we take `rationalIH.in` (`rational.in` augmented by `IntegerHull`):

```
amb_space 3
cone 3
1 1 2
-1 -1 3
1 -2 4
grading
unit_vector 3
HilbertSeries
IntegerHull
```

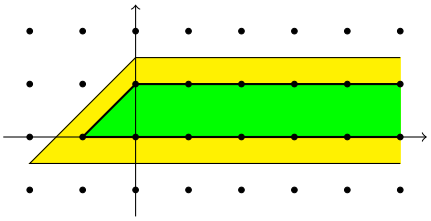
It is our rational polytope from Section 2.5. We know already that the origin is the only lattice point it contains. Nevertheless let us have a look at `rationalIH.IntHull.out`:

```
1 vertices of polyhedron
0 extreme rays of recession cone
1 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 0
rank of recession monoid = 0  (polyhedron is polytope)
internal index = 1
```

```
**************************************************************************

1 vertices of polyhedron:
 0 0 1

0 extreme rays of recession cone:

1 support hyperplanes of polyhedron (homogenized):
0 0 1

2 equations:
 1 0 0
 0 1 0

1 basis elements of generated lattice:
 0 0 1
```

Since the lattice points in P are already known, the goal was to compute the constraints defining the integer hull. Note that all the constraints defining the integer hull can be different from those defining P. In this case the integer hull is cit out by the 2 equations.

As a second example we take the polyhedron of Section 2.9. The integer hull is the "green" polyhedron:



The input is `InhomIneqIH.in` (`InhomIneq.in` augmented by `IntegerHull`). The data of the integer hull are found in `InhomIneqIH.IntHull.out`:

```
...
2 vertices of polyhedron:
-1 0 1
0 1 1

1 extreme rays of recession cone:
1 0 0

3 support hyperplanes of polyhedron (homogenized):
0 -1 1
0  1 0
```

```
1 -1 1
```

2.15. Inhomogeneous congruences

We want to compute the nonnegative solutions of the simultaneous inhomogeneous congruences

$$x_1 + 2x_2 \equiv 3 \quad (7),$$
$$2x_1 + 2x_2 \equiv 4 \quad (13)$$

in two variables. The input file `InhomCong.in` is

```
amb_space 2
constraints 2 symbolic
x[1] + 2x[2] ~ 3 (7);
2x[1] +  2x[2] ~ 4 (13);
```

This is an example of input of symbolic constraints. We use ~ as the best ASCII character for representing the congruence sign $\equiv$.

Alternatively one can use a matrix in the input As for which we must move the right hand side over to the left.

```
amb_space 2
inhom_congruences 2
1 2 -3 7
2 2 -4 13
```

It is certainly harder to read.

The first vector list in the output:

```
3 module generators:
0 54 1
1  1 1
80  0 1
```

Easy to check: if $(1,1)$ is a solution, then it must generate the module of solutions together with the generators of the intersections with the coordinate axes. Perhaps more difficult to find:

```
6 Hilbert basis elements of recession monoid:
0 91 0
1 38 0
3 23 0                                          1 vertices of polyhedron:
5  8 0                                          0 0 91
12  1 0
91  0 0
```

Strange, why is $(0,0,1)$, representing the origin in $\mathbb{R}^2$, not listed as a vertex as well? Well the vertex shown represents an extreme ray in the lattice $\mathbb{E}$, and $(0,0,1)$ does not belong to $\mathbb{E}$.

```
2 extreme rays of recession cone:
0 91 0
91  0 0

3 support hyperplanes of polyhedron (homogenized)
0 0 1
0 1 0
1 0 0

1 congruences:
58 32 1 91
```

Normaliz has simplified the system of congruences to a single one.

```
3 basis elements of generated lattice:
 1 0  33
 0 1 -32
 0 0  91
```

Again, don't forget that Normaliz prints a basis of the efficient lattice $\mathbb{E}$.

2.15.1. Lattice and offset

The set of solutions to the inhomogeneous system is an affine lattice in $\mathbb{R}^2$. The lattice basis of $\mathbb{E}$ above does not immediately let us write down the set of solutions in the form $w+L_0$ with a subgroup L_0, but we can easily transform the basis of $\mathbb{E}$: $(1,1,1)$ is in $\mathbb{E}$ and we use it to reduce the third column of the other two basis elements to 0. Try the file `InhomCongLat.in`:

```
amb_space 2
offset
1 1
lattice 2
5 8
-12 -1
```

2.15.2. Variation of the signs

Suppose we want to solve the system of congruences under the condition that both variables are negative (`InhomCongSigns.in`):

```
amb_space 2
inhom_congruences 2
1 2 -3 7
```

```
2 2 -4 13
signs
-1 -1
```

The two entries of the sign vector impose the sign conditions $x_1 \leq 0$ and $x_2 \leq 0$.

From the output we see that the module generators are more complicated now:

```
4 module generators:
-11   0 1
-4  -7 1
-2 -22 1
0 -37 1
```

The Hilbert basis of the recession monoid is simply that of the nonnegative case multiplied by -1.

2.16. Integral closure and Rees algebra of a monomial ideal

Next, let us discuss the example `MonIdeal.in` (typeset in two columns):

```
amb_space 5
rees_algebra 9
1 2 1 2                 1 0 3 4
3 1 1 3                 5 1 0 1
2 5 1 0                 2 4 1 5
0 2 4 3                 2 2 2 4
0 2 3 4
```

The input vectors are the exponent vectors of a monomial ideal I in the ring $K[X_1, X_2, X_3, X_4]$. We want to compute the normalization of the Rees algebra of the ideal. In particular we can extract from it the integral closure of the ideal. Since we must introduce an extra variable T, we have `amb_space 5`.

In the Hilbert basis we see the exponent vectors of the X_i, namely the unit vectors with last component 0. The vectors with last component 1 represent the integral closure $\bar{I}$ of the ideal. There is a vector with last component 2, showing that the integral closure of I^2 is larger than $\bar{I}^2$.

```
16 Hilbert basis elements:
0 0 0 1 0
...
5 1 0 1 1
6 5 2 2 2

11 generators of integral closure of the ideal:
0 2 3 4
...
5 1 0 1
```

The output of the generators of $\bar{I}$ is the only place where we suppress the homogenizing variable for "historic" reasons. If we extract the vectors with last component 1 from the extreme rays, then we obtain the smallest monomial ideal that has the same integral closure as I.

```
10 extreme rays:
0 0 0 1 0
...
5 1 0 1 1
```

The support hyperplanes which are not just sign conditions describe primary decompositions of all the ideals $\overline{I^k}$ by valuation ideals. It is not hard to see that none of them can be omitted for large k (for example, see: W. Bruns and G. Restuccia, Canonical modules of Rees algebras. J. Pure Appl. Algebra 201, 189–203 (2005)).

```
23 support hyperplanes:
0 0 0  0   1
0 ...
6 0 1  3 -13
```

2.16.1. Only the integral closure of the ideal

If only the integral closure of the ideal is to be computed, one can choose the input as follows (`IntClMonId.in`):

```
amb_space 4
vertices 9
1 2 1 2 1
...
2 2 2 4 1
cone 4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

The generators of the integral closure appear as module generators in the output and the generators of the smallest monomial ideal with this integral closure are the vertices of the polyhedron.

2.17. Starting from a binomial ideal

As an example, we consider the binomial ideal generated by

$$X_1^2X_2 - X_4X_5X_6,\ \ X_1X_4^2 - X_3X_5X_6,\ \ X_1X_2X_3 - X_5^2X_6.$$

We want to find an embedding of the toric ring it defines and the normalization of the toric ring. The input vectors are obtained as the differences of the two exponent vectors in the

binomials. So the input ideal `lattice_ideal.in` is

```
amb_space 6
lattice_ideal 3
2 1  0 -1 -1  -1
1 0 -1  2 -1  -1
1 1  1  0 -2  -1
```

In order to avoid special input rules for this case in which our object is not defined as a subset of an ambient space, but as a quotient of type *generators/relations*, we abuse the name `amb_space`: it determines the space in which the input vectors live.

We get the output

```
6 original generators of the toric ring
```

namely the residue classes of the indeterminates.

```
9 Hilbert basis elements
9 lattice points in polytope (Hilbert basis elements of degree 1)
```

So the toric ring defined by the binomials is not normal. Normaliz found the standard grading on the toric ring. The normalization is generated in degree 1, too (in this case).

```
5 extreme rays
5 support hyperplanes

embedding dimension = 3
rank = 3 (maximal)
external index = 1
internal index = 1
original monoid is not integrally closed
```

We saw that already.

```
size of triangulation   = 5
resulting sum of |det|s = 10

grading:
-2 1 1
```

This is the grading on the ambient space (or polynomial ring) defining the standard grading on our subalgebra. The enumerative data that follow are those of the normalization!

```
degrees of extreme rays:
1: 5

Hilbert basis elements are of degree 1

multiplicity = 10
```

```
Hilbert series:
1 6 3
denominator with 3 factors:
1: 3

degree of Hilbert Series as rational function = -1

Hilbert polynomial:
1 3 5
with common denominator = 1

rank of class group = 2
class group is free

**************************************************************************

6 original generators:
1 0 0
2 3 5
0 0 1
1 1 2
0 1 3
3 1 0
```

This is an embedding of the toric ring defined by the binomials. There are many choices, and Normaliz has taken one of them. You should check that the generators in this order satisfy the binomial equations. Turning to the ring theoretic interpretation, we can say that the toric ring defined by the binomial equations can be embedded into $K[Y_1, Y_2, Y_3]$ as a monomial subalgebra that is generated by $Y_1^0Y_2^0Y_3^1, \ldots, Y_1^1Y_2^0Y_3^3$.

Now the generators of the normalization:

```
9 lattice points in polytope (Hilbert basis elements of degree 1):
                                          5 extreme rays:
0 0 1                                     0 0 1
0 1 3                                     0 1 3
1 0 0                                     1 0 0
1 1 2                                     2 3 5
1 2 4                                     3 1 0
2 1 1
2 2 3                                    5 support hyperplanes:
2 3 5                                    0   0 15
3 1 0                                    0   1 0
                                         1   0 0
                                         2  -3 1
                                         5 -15 7
```

```
0 further Hilbert basis elements of higher degree:
```

3. The input file

The input file `<project>.in` consists of one or several items. There are several types of items:

(1) definition of the ambient space,
(2) matrices with integer or rational entries (depending on the type),
(3) vectors with integer entries,
(4) constraints in or symbolic format,
(5) a polynomial,
(6) computation goals and algorithmic variants,
(7) numerical parameters,
(8) comments.

An item cannot include another item. In particular, comments can only be included between other items, but not within another item. Matrices and vectors can have two different formats, plain and formatted.

Matrices and vectors are classified by the following attributes:

(1) generators, constraints, accessory,
(2) cone/polyhedron, (affine) lattice,
(3) homogeneous, inhomogeneous.

In this classification, equations are considered as constraints on the lattice because Normaliz treats them as such – for good reason: it is very easy to intersect a lattice with a hyperplane.

The line structure is irrelevant for the interpretation of the input, but it is advisable to use it for the readability of the input file.

The input syntax of Normaliz 2 can still be used. It is explained in Appendix C.

3.1. Input items

3.1.1. The ambient space and lattice

The ambient space is specified as follows:

```
amb_space <d>
```

where `<d>` stands for the dimension d of the ambient vector space $\mathbb{R}^d$ in which the geometric objects live. The *ambient lattice* $\mathbb{A}$ is set to $\mathbb{Z}^d$.

Alternatively one can define the ambient space implicitly by

```
amb_space auto
```

In this case the dimension of the ambient space is determined by Normaliz from the first formatted vector or matrix in the input file. It is clear that any input item that requites the knowledge of the dimension can only follow the first formatted vector or matrix.

In the following the letter d will always denote the dimension set with `amb_space`.

An example:

```
amb_space 5
```

indicates that polyhedra and lattices are subobjects of $\mathbb{R}^5$. The ambient lattice is $\mathbb{Z}^5$.

The first non-comment input item must specify the ambient space.

3.1.2. Plain vectors

A plain vector is built as follows:

```
<T>
<x>
```

Again `<T>` denotes the type and `<x>` is the vector itself. The number of components is determined by the type of the vector and the dimension of the ambient space. At present, all vectors have length d.

Example:

```
grading
1 0 0
```

Normaliz allows also the input of sparse vectors. Sparse input is signalized by the key word `sparse` as the first entry. It is followed by entries of type `<col>:<val>` where `<pos>` denotes the column and `<val>` the value in that column. (The unspecified columns have entry 0.) A sparse vector is terminated by the character `;` .

Example:

```
grading
sparse 1:1;
```

For certain vectors there also exist shortcuts. Examples:

```
total_degree
unit_vector 25
```

3.1.3. Formatted vectors

A formatted vector is built as follows:

```
<T>
[ <x> ]
```

where <T> denotes the type and <x> is the vector itself. The components can be separated by white space, commas or semicolons. An example showing all possibilities (not recommended):

```
grading
[1,0; 0 5]
```

3.1.4. Plain matrices

A plain matrix is built as follows:

```
<T> <m>
<x_1>
...
<x_m>
```

Here <T> denotes the type of the matrix, <m> the number of rows, and <x_1>,...,<x_m> are the rows. Some types allow rational and floating point matrix entries, others are restricted to integers; see Sections 3.1.9 and 3.1.10.

The number of columns is implicitly defined by the dimension of the ambient space and the type of the matrix. Example (with amb_space 3):

```
cone 3
1/3 2 3
4 5 6
11 12/7 13/21
```

Normaliz allows the input of matrices in transposed form:

```
<T> transpose <n>
<x_1>
...
<x_m>
```

Note that <n> is now the number of *columns* of the matrix that follows it (assumed to be the number of input vectors). The number of rows is determined by the dimension of the ambient space and the type of the matrix. Example:

```
cone transpose 3
1  0   3/2
0 1/9   4
```

is equivalent to

```
cone 3
 1  0
 0 1/9
3/2 4
```

Like vectors, matrices have a sparse input variant, again signalized by the key word `sparse`. The rows are sparse vectors with entries `<col>:<val>`, and each row is concluded by the character `;`.

Example:

```
inequalities 3 sparse
1:1;
2:1;
3:1;
```

chooses the 3×3 unit matrix as a matrix of type `inequalities`. Note that also in case of transposed matrices, sparse entry is row by row.

Matrices may have zero rows. Such empty matrices like

```
inhom_inequalities 0
```

can be used to make the input inhomogeneous (Section 3.1.14) or to avoid the automatic choice of the positive orthant in certain cases (Section 3.1.15). (The empty `inhom_inequalities` have both effects simultaneously.) Apart from these effects, empty matrices have no influence on the computation.

3.1.5. Formatted matrices

A formatted matrix is built as follows:

```
<T>
[ [<x_1>]
...
[<x_m>]  ]
```

Here `<T>` denotes the type of the matrix and `<x_1>`,...,`<x_m>` are vectors. Legal separators are white space, commas and semicolons. An example showing all possibilities (not really recommended):

```
cone [
[ 2 1][3/7 4];
[0 1],
[9 10] [11 12/13]
]
```

Similarly as plain matrices, formatted matrices can be given in transposed form, and they can be empty.

3.1.6. Constraints in tabular format

This input type is somewhat closer to standard notation than the encoding of constraints in matrices. The general type of equations and inequalities is

```
<x> <rel> <int>;
```

where <x> denotes a vector of length d, <rel> is one of the relations =, <=, >=, <, > and <int> is an integer.

Congruences have the form

```
<x> ~ <int> (<mod>);
```

where <mod> is a nonzero integer.

Examples:

```
1/2 -2 >= 5
1 -1/7 = 0
-1 1 ~ 7 (9)
```

Note: all numbers and relation signs must be separated by white space.

3.1.7. Constraints in symbolic format

This input type is even closer to standard notation than the encoding of constraints in matrices or in tabular format. It is especially useful if the constraints are sparse. Instead of assigning a value to a coordinate via its position in a vector, it uses coordinates named x[<n>] where <n> is the index of the coordinate. The index is counted from 1.

The general type of equations and inequalities is

```
<lhs> <rel> <rhs>;
```

where <lhs> and <rhs> denote linear function of the x<n> with integer coefficients. As above, <rel> is one of the relations =, <=, >=, <, >. (Both <lhs> and <rhs> must be nonempty.) Note the terminating semicolon.

Congruences have the form

```
<lhs> ~ <rhs> (<mod>);
```

where <mod> is a nonzero integer.

Examples:

```
1/3x[1] >= 2x[2] + 5;
x[1]+1=1/4x[2] ;
-x[1] + x[2] ~ 7 (9);
```

There is no need to insert white space for separation, but it may be inserted anywhere where it does not disrupt numbers or relation signs.

3.1.8. Polynomials

For the computation of weighted Ehrhart series and integrals Normaliz needs the input of a polynomial with rational coefficients. The polynomial is first read as a string. For the computation the string is converted by the input function of CoCoALib [1]. Therefore any string representing a valid CoCoA expression is allowed. However the names of the indeterminates are fixed: `x[1]`,...,`x[<N>` where `<N>]` is the value of `amb_space`. The polynomial must be concluded by a semicolon.

Example:

```
(x[1]+1)*(x[1]+2)*(x[1]+3)*(x[1]+4)*(x[1]+5)*
(x[2]+1)*(x[3]+1)*(x[4]+1)*(x[5]+1)*(x[6]+1)*(x[7]+1)*
(x[8]+1)*(x[8]+2)*(x[8]+3)*(x[8]+4)*(x[8]+5)*1/14400;

(x[1]*x[2]*x[3]*x[4])^2*(x[1]-x[2])^2*(x[1]-x[3])^2*
(x[1]-x[4])^2*(x[2]-x[3])^2*(x[2]-x[4])^2*(x[3]-x[4])^2;
```

3.1.9. Rational numbers

Rational numbers are allowed in input matrices, but not in all. They are *not* allowed in vectors and in matrices containing lattice generators and in congruences, namely in

```
lattice     cone_and_lattice   normalization  offset         open_facets
congruences inhom_congruences  rees_algebra   lattice_ideal
grading     dehomogenization   signs          strict_signs
```

They are allowed in `saturation` since it defines the intersection of the vector space generated by the rows of the matrix with the integral lattice.

Note: Only positive numbers are allowed as denominators. Negative denominators may result in a segmentation fault. Illegal formats may result in

```
std::exception caught... "mpz_set_str" ...  exiting.
```

Normaliz first reduces the input numbers to lowest terms. Then each row of a matrix is multiplied by the least common multiple of the denominators of its entries. In all applications in which the original monoid generators play a role, one should use only integers in input matrices to avoid any ambiguity.

3.1.10. Decimal fractions and floating point numbers

Normaliz accepts decimal fractions and floating point numbers in its input files. These are precisely converted to ordinary fractions (or integers). Examples:

```
1.1 --> 11/10    0.5 --> 1/2    -.1e1 --> -1
```

It is not allowed to combine an ordinary fraction and a decimal fraction in the same number. In other words, expressions like `1.0/2` are not allowed.

3.1.11. Computation goals and algorithmic variants

These are single or compound words, such as

```
HilbertBasis
Multiplicity
```

The file can contain several computation goals, as in this example.

3.1.12. Comments

A comment has the form

```
/* <text> */
```

where `<text>` stands for the text of the comment. It can have arbitrary length and stretch over several lines. Example:

```
/* This is a comment
*/
```

Comments are only allowed at places where also a new keyword would be allowed, especially not between the entries of a matrix or a vector. Comments can not be nested.

3.1.13. Restrictions

Input items can almost freely be combined, but there are some restrictions:

(1) The types

`cone, cone_and_lattice, polytope, rees_algebra`

exclude each other mutually.

(2) The input type `subspace` excludes `polytope` and `rees_algebra`.

(3) The types

`lattice`, `saturation`, `cone_and_lattice`

exclude each other mutually.

(4) `polytope` can not be combined with `grading`.

(5) The only type that can be combined with `lattice_ideal` is `grading`.

(6) The following types cannot be combined with inhomogeneous types or `dehomogenization`:

`polytope, rees_algebra, excluded_faces`

(7) The following types cannot be combined with inhomogeneous types:

`dehomogenization, support_hyperplanes`

(8) Special restrictions apply for the input type `open_facets`; see Section 3.12.

A non-restriction: the same type can appear several times. This is useful if one wants to combine different formats, for example

```
inequalities 2 sparse
1:1;
1:1 3:-1;
inequalities 2
1 1 0 1
1 -1 -1 0
```

3.1.14. Homogeneous and inhomogeneous input

Apart from the restrictions listed in the previous section, homogeneous and inhomogeneous types can be combined as well as generators and constraints. A single inhomogeneous type or `dehomogenization` in the input triggers an inhomogeneous computation. The input item of inhomogeneous type may be an empty matrix.

3.1.15. Default values

If there is no lattice defining item, Normaliz (virtually) inserts the the unit matrix as an input item of type `lattice`. If there is no cone defining item, the unit matrix is (additionally) inserted as an input item of type `cone`.

If the input is inhomogeneous, then Normaliz provides default values for vertices and the offset as follows:

(1) If there is an input matrix of lattice type lattice, but no `offset`, then the offset 0 is inserted.
(2) If there is an input matrix of type cone, but no `vertices`, then the vertex 0 is inserted.

An important point. If the input does not contain any cone generators or inequalities, Normaliz automatically assumes that you want to compute in the positive orthant. In order to avoid this choice you can add an empty matrix of inequalities. This will not affect the results, but avoid the sign restriction.

3.1.16. Normaliz takes intersections (almost always)

The input may contain several cone defining items and several lattice defining items.

The sublattice L defined by the lattice input items is the *intersection* of the sublattices defined by the single items. The polyhedron P is defined as the intersection of all polyhedra defined by the single polyhedron defining items. The object then computed by Normaliz is

$$P \cap L.$$

There are three notable exceptions to the rule that Normaliz takes intersections:

(1) `vertices` and `cone` form a unit. Together they define a polyhedron.
(2) The same applies to `offset` and `lattice` that together define an affine lattice.
(3) The `subspace` is added to `cone` or `cone_and_lattice`.

3.2. Homogeneous generators

3.2.1. Cones

The main type is `cone`. The other two types are added for special computations.

`cone` is a matrix with d columns. Every row represents a vector, and they define the cone generated by them. Section 2.3, `2cone.in`

`subspace` is a matrix with d columns. The linear subspace generated by the rows is added to the cone. Section 6.14.4.

`polytope` is a matrix with $d-1$ columns. It is internally converted to `cone` extending each row by an entry 1. Section 2.4, `polytope.in`. This input type automatically sets `NoGradingDenom` and defines the grading $(0,\dots,0,1)$.

`rees_algebra` is a matrix with $d-1$ columns. It is internally converted to type `cone` in two steps: (i) each row is extended by an entry 1 to length d. (ii) The first $d-1$ unit vectors of length d are appended. Section 2.16, `MonIdeal.in`.

`extreme_rays` is a matrix with d columns. It requires homogeneous input. It is the input type for precomputed extreme rays. Normaliz takes their correctness for granted, but they do not define the cone. Section 6.19.2, `2cone_extt.in`.

Moreover, it is possible to define a cone and a lattice by the same matrix:

`cone_and_lattice` The vectors of the matrix with d columns define both a cone and a lattice. Section 2.11, `A443.in`.

If `subspace` is used in combination with `cone_and_lattice`, then the sublattice generated by its rows is added to the lattice generated by `cone_and_lattice`.

The Normaliz 2 types `integral_closure` and `normalization` can still be used. They are synonyms for `cone` and `cone_and_lattice`, respectively.

3.2.2. Lattices

There are 3 types:

`lattice` is a matrix with d columns. Every row represents a vector, and they define the lattice generated by them. Section 2.6.3, `3x3magiceven_lat.in`

`saturation` is a matrix with d columns. Every row represents a vector, and they define the *saturation* of the lattice generated by them. Section 2.6.3, `3x3magic_sat.in`.

`cone_and_lattice` See Section 3.2.1.

3.3. Homogeneous Constraints

3.3.1. Cones

inequalities is a matrix with d columns. Every row $(\xi_1,\dots,\xi_d)$ represents a homogeneous inequality

$$\xi_1 x_1 + \cdots + \xi_d x_d \geq 0, \qquad \xi_i \in \mathbb{Z},$$

for the vectors $(x_1,\dots,x_d) \in \mathbb{R}^d$. Sections 2.3.2, 2.5.2 , `2cone_ineq.in`, `poly_ineq.in`

signs is a vector with d entries in $\{-1,0,1\}$. It stands for a matrix of type `inequalities` composed of the sign inequalities $x_i \geq 0$ for the entry 1 at the i-th component and the inequality $x_i \leq 0$ for the entry -1. The entry 0 does not impose an inequality. See 2.15.2, `InhomCongSigns.in`.

nonnegative It stands for a vector of type `sign` with all entries equal to 1. See Section 2.10, `Condorcet.in`.

excluded_faces is a matrix with d columns. Every row $(\xi_1,\dots,\xi_d)$ represents an inequality

$$\xi_1 x_1 + \cdots + \xi_d x_d > 0, \qquad \xi_i \in \mathbb{Z},$$

for the vectors $(x_1,\dots,x_d) \in \mathbb{R}^d$. It is considered as a homogeneous input type though it defines inhomogeneous inequalities. The faces of the cone excluded by the inequalities are excluded from the Hilbert series computation, but `excluded_faces` behaves like `inequalities` in every other respect . Section 2.10.1, `CondorcetSemi.in`.

support_hyperplanes is a matrix with d columns. It requires homogeneous input. It is the input type for precomputed support hyperplanes. Note that it overrides all other inequalities in the input, but `excluded_faces` still exclude the faces defined by them. Section 6.19.1, `2cone_supp.in`.

3.3.2. Lattices

equations is a matrix with d columns. Every row $(\xi_1,\dots,\xi_d)$ represents an equation

$$\xi_1 x_1 + \cdots + \xi_d x_d = 0, \qquad \xi_i \in \mathbb{Z},$$

for the vectors $(x_1,\dots,x_d) \in \mathbb{R}^d$. Section 2.6, `3x3magic.in`

congruences is a matrix with $d+1$ columns. Each row $(\xi_1,\dots,\xi_d,c)$ represents a congruence

$$\xi_1 z_1 + \cdots + \xi_d z_d \equiv 0 \mod c, \qquad \xi_i, c \in \mathbb{Z},$$

for the elements $(z_1,\dots,z_d) \in \mathbb{Z}^d$. Section 2.6.2, `3x3magiceven.in`.

3.4. Inhomogeneous generators

3.4.1. Polyhedra

vertices is a matrix with $d+1$ columns. Each row $(p_1,\dots,p_d,q)$, $q>0$, specifies a generator of a polyhedron (not necessarily a vertex), namely

$$v_i = \left(\frac{p_1}{q},\dots,\frac{p_n}{q}\right), \qquad p_i \in \mathbb{Z}, q \in \mathbb{Z}_{>0},$$

Section 2.9.1, `InhomIneq_gen.in`

Note: `vertices` and `cone` together define a polyhedron. If `vertices` is present in the input, then the default choice for `cone` is the empty matrix.

The Normaliz 2 input type `polyhedron` can still be used.

3.4.2. Affine lattices

offset is a vector with d entries. It defines the origin of the affine lattice. Section 2.15.1, `InhomCongLat.in`.

Note: `offset` and `lattice` (or `saturation`) together define an affine lattice. If `offset` is present in the input, then the default choice for `lattice` is the empty matrix.

3.5. Inhomogeneous constraints

3.5.1. Polyhedra

inhom_inequalities is a matrix with $d+1$ columns. We consider inequalities

$$\xi_1 x_1 + \cdots + \xi_d x_d \geq \eta, \qquad \xi_i, \eta \in \mathbb{Z},$$

rewritten as

$$\xi_1 x_1 + \cdots + \xi_d x_d + (-\eta) \geq 0$$

and then represented by the input vectors

$$(\xi_1,\dots,\xi_d,-\eta).$$

Section 2.9, `InhomIneq.in`.

strict_inequalities is a matrix with d columns. We consider inequalities

$$\xi_1 x_1 + \cdots + \xi_d x_d \geq 1, \qquad \xi_i \in \mathbb{Z},$$

represented by the input vectors

$$(\xi_1,\dots,\xi_d).$$

Section 2.3.3, `2cone_int.in`.

strict_signs is a vector with d components in $\{-1,0,1\}$. It is the "strict" counterpart to `signs`. An entry 1 in component i represents the inequality $x_i > 0$, an entry -1 the opposite inequality, whereas 0 imposes no condition on x_i. 2.10.2, `Condorcet_one.in`

3.5.2. Affine lattices

inhom_equations is a matrix with $d+1$ columns. We consider equations

$$\xi_1 x_1 + \cdots + \xi_d x_d = \eta, \qquad \xi_i, \eta \in \mathbb{Z},$$

rewritten as

$$\xi_1 x_1 + \cdots + \xi_d x_d + (-\eta) = 0$$

and then represented by the input vectors

$$(\xi_1, \ldots, \xi_d, -\eta).$$

See 2.7`NumSemi.in`.

inhom_congruences We consider a matrix with $d+2$ columns. Each the row $(\xi_1, \ldots, \xi_d, -\eta, c)$ represents a congruence

$$\xi_1 z_1 + \cdots + \xi_d z_d \equiv \eta \mod c, \qquad \xi_i, \eta, c \in \mathbb{Z},$$

for the elements $(z_1, \ldots, z_d) \in \mathbb{Z}^d$. Section 2.15, `InhomCongSigns.in`.

3.6. Tabular constraints

constraints allows the input of equations, inequalities and congruences in a format that is close to standard notation. As for matrix types the keyword `constraints` is followed by the number of constraints. The syntax of tabular constraints has been described in Section 3.2.1. If $(\xi_1, \ldots, \xi_d)$ is the vector on the left hand side and η the integer on the right hand side, then the constraint defines the set of vectors $(x_1, \ldots, x_d)$ such that the relation

$$\xi_1 x_1 + \ldots \xi_d x_d \ \texttt{rel}\ \eta$$

is satisfied, where `rel` can take the values $=, \leq, \geq, <, >$ with the represented by input strings `=,<=,>=,<,>`, respectively.

The input string `~` represents a congruence $\equiv$ and requires the additional input of a modulus. It represents the congruence

$$\xi_1 x_1 + \ldots \xi_d x_d \equiv \eta \pmod{c}.$$

Sections 2.3.3, `2cone_int.in`, 2.6.2, `3x3magiceven.in`, 2.9, `InhomIneq.in`.

A right hand side $\neq 0$ makes the input inhomogeneous, as well as the relations $<$ and $>$. Strict inequalities are always understood as conditions for integers. So

$$\xi_1 x_1 + \cdots + \xi_d x_d < \eta$$

is interpreted as

$$\xi_1 x_1 + \ldots \xi_d x_d \leq \eta - 1,$$

3.6.1. Forced homogeneity

It is often more natural to write constraints in inhomogeneous form, even when one wants the computation to be homogeneous. The type `constraints` does not allow this. Therefore we have introduced

`hom_constraints` for the input of equations, non-strict inequalities and congruences in the same format as `constraints`, except that these constraints are meant to be for a homogeneous computation. It is clear that the left hand side has only $d-1$ entries now. See Section 2.5.2, `poly_hom_const.in`.

3.7. Symbolic constraints

The input syntax is

`constraints <n> symbolic` where `<n>` is the number of constraints in symbolic form that follow.

The constraints have the form described in Section 3.1.7. Note that every symbolic constraint (including the last) must be terminated by a semicolon.

See 2.7, `NumSemi.in`, 2.15, `InhomCong.in`.

The interpretation of homogeneity follows the same rules as for tabular constraints. The variant `hom_constraints` is allowed and works as for tabular constraints.

3.8. Relations

Relations do not select a sublattice of $\mathbb{Z}^d$ or a subcone of $\mathbb{R}^d$, but define a monoid as a quotient of $\mathbb{Z}^d_+$ modulo a system of congruences (in the semigroup sense!).

The rows of the input matrix of this type are interpreted as generators of a subgroup $U \subset \mathbb{Z}^d$, and Normaliz computes an affine monoid and its normalization as explained in Section A.5.

Set $G = \mathbb{Z}^d/U$ and $L = G/\mathrm{torsion}(G)$. Then the ambient lattice is $\mathbb{A} = \mathbb{Z}^r$, $r = \operatorname{rank} L$, and the efficient lattice is L, realized as a sublattice of $\mathbb{A}$. Normaliz computes the image of $\mathbb{Z}^d_+$ in L and its normalization.

`lattice_ideal` is a matrix with d columns containing the generators of the subgroup U. Section 2.17, `lattice_ideal.in`.

The type `lattice_ideal` cannot be combined with any other input type (except `grading`)—such a combination would not make sense. (See Section 3.10.1 for the use of a grading in this case.)

3.9. Unit vectors

A grading or a dehomogenization is often given by a unit vector:

unit_vector <n> represents the nth unit vector in $\mathbb{R}^d$ where n is the number given by <n>.

This shortcut cannot be used as a row of a matrix. It can be used whenever a single vector is asked for, namely after `grading`, `dehomogenization`, `signs` and `strict_signs`. See Section 2.5, `rational.in`

3.10. Grading

This type is accessory. A $\mathbb{Z}$-valued grading can be specified in two ways:

(1) *explicitly* by including a grading in the input, or

(2) *implicitly*. In this case Normaliz checks whether the extreme integral generators of the monoid lie in an (affine) hyperplane A given by an equation $\lambda(x) = 1$ with a $\mathbb{Z}$-linear form λ. If so, then λ is used as the grading.

Implicit gradings are only possible for homogeneous computations.

If the attempt to find an implicit grading causes an arithmetic overflow and `verbose` has been set (say, by the option `-c`), then Normaliz issues the warning

```
Giving up the check for a grading
```

If you really need this check, rerun Normaliz with a bigger integer type.

Explicit definition of a grading:

grading is a vector of length d representing the linear form that gives the grading. Section 2.5, `rational.in`.

total_degree represents a vector of length d with all entries equal to 1. Section 2.10, `Condorcet.in`.

Before Normaliz can apply the degree, it must be restricted to the effective lattice $\mathbb{E}$. Even if the entries of the grading vector are coprime, it often happens that all degrees of vectors in $\mathbb{E}$ are divisible by a greatest common divisor $g > 1$. Then g is extracted from the degrees, and it will appear as `denominator` in the output file.

Normaliz checks whether all generators of the (recession) monoid have positive degree (after passage to the quotient modulo the unit group in the nonpointed case). Vertices of polyhedra may have degrees ≤ 0.

3.10.1. With lattice_ideal input

In this case the unit vectors correspond to generators of the monoid. Therefore the degrees assigned to them must be positive. Moreover, the vectors in the input represent binomial relations, and these must be homogeneous. In other words, both monomials in a binomial must have the same degree. This amounts to the condition that the input vectors have degree 0. Normaliz checks this condition.

3.11. Dehomogenization

Like `grading` this is an accessory type.

Inhomogeneous input for objects in $\mathbb{R}^d$ is homogenized by an additional coordinate and then computed in $\mathbb{R}^{d+1}$, but with the additional condition $x_{d+1} \geq 0$, and then dehomogenizing all results: the substitution $x_{d+1} = 1$ acts as the *dehomogenization*, and the inhomogeneous input types implicitly choose this dehomogenization.

Like the grading, one can define the dehomogenization explicitly:

`dehomogenization` is a vector of length d representing the linear form δ.

The dehomogenization can be any linear form δ satisfying the condition $\delta(x) \geq 0$ on the cone that is truncated. (In combination with constraints, the condition $\delta(x) \geq 0$ is automatically satisfied since δ is added to the constraints.)

The input type `dehomogenization` can only be combined with homogeneous input types, but makes the computation inhomogeneous, resulting in inhomogeneous output. The polyhedron computed is the intersection of the cone $\mathbb{C}$ (and the lattice $\mathbb{E}$) with the hyperplane given by $\delta(x) = 1$, and the recession cone is $\mathbb{C} \cap \{x : \delta(x) = 0\}$.

A potential application is the adaptation of other input formats to Normaliz. The output must then be interpreted accordingly.

Section 6.12, `dehomogenization.in`.

3.12. Open facets

The input type `open_facets` is similar to `strict_inequalities`. However, it allows to apply strict inequalities that are not yet known. This makes only sense for simplicial polyhedra where a facet can be identified by the generator that does *not* lie in it.

`open_facets` is a vector with entries $\in \{0, 1\}$.

The restrictions for the use of open facets are the following:

(1) Only the input types `cone`, `vertices` and `grading` can appear together with `open_facets`.
(2) The vectors in `cone` are linearly independent.
(3) There is at most one vertex.

The number of vectors in `cone` may be smaller than d, but `open_facets` must have d entries.

`open_facets` make the computation inhomogeneous. They are interpreted as follows. Let v be the vertex—if there are no `vertices`, then v is the origin. The shifted $C' = v + C$ is cut out by affine-linear inequalities $\lambda_i(x) \geq 0$ with coprime integer coefficients. We number these in such a way that $\lambda_i(v + c_i) \neq 0$ for the generators c_i of C (in the input order), $i = 1, \ldots, n$. Then all subsequent computations are applied to the shifted cone $C'' = v' + C$ defined by the inequalities

$$\lambda_i(x) \geq u_i$$

where the vector $(u_1, \ldots, u_d)$ is given by `open_facets`. (If $\dim C < d$, then the entries u_j with $j > \dim C$ are ignored.)

That 1 indicates 'open" is in accordance with its use for the disjoint decomposition; see Section 6.15.2. Section 6.18 discusses an example.

3.13. Coordinates for projection

The coordinates of a projection of the cone can be chosen by

projection_coordinates . It is a 0-1 vector of length d.

The entries 1 mark the coordinates of the image of the projection. The other coordinates give the kernel of the projection. See Section 6.13 for an exasmple.

3.14. Numerical parameters

Certain numerical parameters used by Normaliz can (only) be set in the input file.

3.14.1. Degree bound for series expansion

It can be set by

```
expansion_degree <n>
```

where <n> is the number of coefficients to be computed and printed. See Section 6.10.

3.14.2. Number of significant coefficients of the quasipolynomial

It can be set by

```
nr_coeff_quasipol <n>
```

where <n> is the number of highest coefficients to be printed. See Section 6.11.

3.15. Pointedness

Since version 3.1 Normaliz can also compute nonpointed cones and polyhedra without vertices.

3.16. The zero cone

The zero cone with an empty Hilbert basis is a legitimate object for Normaliz. Nevertheless a warning message is issued if the zero cone is encountered.

4. Computation goals and algorithmic variants

The library `libnormaliz` contains a class `ConeProperties` that collects computation goals, algorithmic variants and additional data that are used to control the work flow in `libnormaliz` as well as the communication with other programs. The latter are not important for the Normaliz user, but are listed as a reference for `libnormaliz`. See Appendix D for a description of `libnormaliz`.

All computation goals and algorithmic variants can be communicated to Normaliz in two ways:

(1) in the input file, for example `HilbertBasis`,
(2) via a verbatim command line option, for example `--HilbertBasis`.

For the most important choices there are single letter command line options, for example `-N` for `HilbertBasis`. The single letter options ensure backward compatibility to Normaliz 2. In jNormaliz they are also accessible via their full names.

Some computation goals apply only to homogeneous computations, and some others make sense only for inhomogeneous computations.

Some single letter command line options combine two or more computation goals, and some algorithmic variants imply computation goals.

4.1. Default choices and basic rules

If several computation goals are set, all of them are pursued. In particular, computation goals in the input file and on the command line are accumulated. But

`--ignore, -i` on the command line switches off the computation goals and algorithmic variants set in the input file.

The default computation goal is set if neither the input file nor the command line contains a computation goal or an algorithmic variant that implies a computation goal. It is

`SupportHyperplanes` + `HilbertBasis` + `HilbertSeries` .

In the homogeneous case, `ClassGroup` is icluded as well.

If set explicitly in the input file or on the command line the following adds these computation goals:

`DefaultMode`

`DefaultMode` can be set explicitly in addition to other computation goals. If it is set, implicitly or explicitly, Normaliz will not complain about unreachable computation goals.

4.2. The choice of algorithmic variants

For its main computation goals Normaliz has algorithmic variants. It tries to choose the variant that seems best for the given input data. This automatic choice may however be a bad one. Therefore the user can completely control which algorithmic variant is used.

4.2.1. Primal vs. dual

For the computation of Hilbert bases Normaliz has two algorithms, the primal algorithm that is based on triangulations, and the dual algorithm that is of type "pair completion". We have seen both in Section 2. Roughly speaking, the primal algorithm is the first choice for generator input, and the dual algorithm is usually better for constraints input. The choice also applies to the computation of degree 1 elements. However, for them the default choice is project-and-lift. See Section 6.2.1. The conditions under which the dual algorithm is chosen are specified in Section 6.5.

The choice of the algorithm can be fixed or blocked:

`DualMode, -d` activates the dual algorithm for the computation of the Hilbert basis and degree 1 elements. Includes `HilbertBasis`, unless `Deg1Elements` is set. It overrules `IsIntegrallyClosed`.

`PrimalMode, -P` blocks the use of the dual algorithm.

The automatic choice can of course fail. See Section 6.5 for an example for which it is bad.

4.2.2. Lattice points in polytopes

For this task Normaliz has several methods. They are discussed in Section 6.2. The default choice is the project-and-lift algorithm. It can be chosen explicitly:

`Projection, -j`

`NoProjection` blocks it.

Alternative choices are

`ProjectionFloat, -J` , project-and-lift with floating point arithmetic,

`PrimalMode, -P` , triangulation based method,

`Approximate, -r` , approximation of rational polytopes followed by triangulation and

`DualMode, -d` , dual algorithm.

`PrimalMode` and `DualMode` do not imply `Deg1Elements` since they can also be used for Hilbert bases.

The following options modify `Projection` and `ProjectionFloat`:

`NoLLL` blocks the use of LLL reduced coordinates,

`PrimalMode, -P` blocks relaxation.

Both LLL and relaxation are switched on by default. See Section 6.2.3.

4.2.3. Bottom decomposition

Bottom decomposition is a way to produce an optimal triangulation for a given set of generators. It is discussed in Section 6.3. The criterion for its automatic choice is explained there. It can be forced or blocked:

BottomDecomposition, -b tells Normaliz to use bottom decomposition in the primal algorithm.

NoBottomDec, -o forbids Normaliz to use bottom decomposition in the primal algorithm, even if it would otherwise be chosen because of large roughness (see Section 6.3).

An option to be mentioned in this context is

KeepOrder, -k forbids Normaliz to reorder the generators of the efficient cone $\mathbb{C}$. Only useful if original monoid generators are defined. Also blocks `BottomDecomposition`.

`KeepOrder` is only allowed if `OriginalMonoidGenerators` are defined. It is rarely a good idea to set `KeepOrder` (try it). It is primarily used internally when data must be computed in an auxiliary cone.

4.2.4. Multilicity and volume

For the computation of multiplicities Normaliz offers has two main algorithms:

(1) the computation and evalutaion of a full triangulation,
(2) descent in the face lattice (see Section 6.6).

Moreover, (1) has a variant via symmetrization (see below).

Normaliz procceds as follows: if the conditions for its use are satisfied, it tries the descent algorithm. These conditions are the same as for the choice of the dual algorithm in Hilbert basis or lattice point computations (see Section 6.5). Next it will check whether the conditions for symmetrization are satisfied (unlikely, if descent has not been chosen), and finally it falls back on (1).

Descent, -F chooses the descent algorithm for the multiplicity (or volume) computation. (Does not imply `Multiplicity`or `Volume` !).

NoDescent blocks it.

Note: `PrimalMode` does not block the descent algorithm (as it did in previous versions).

4.2.5. Symmetrization

In rare cases Normaliz can use symmetrization in the computation of multiplicities or Hilbert series. If applicable, this is a very strong tool. We have mentioned it in Section 2.10 and will discuss it in Section 6.8. It will be chosen automatically, but can also be forced or blocked:

Symmetrize, -Y lets Normaliz compute the multiplicity and/or the Hilbert series via symmetrization (or just compute the symmetrized cone).

NoSymmetrization blocks symmetrization.

4.2.6. Subdivision of simplicial cones

Normaliz tries to subdivide "large" simplicial cones; see Section 6.4. If your executable is built with SCIP, you can set

`SCIP`

However, in general, Normaliz' own method is faster and more reliable.

Subdivision requires enlarging the set of generators and can lead to a nested triangulation (see Sections 6.4 and 6.15.1). The subdivision can be blocked by

`NoSubdivision`

4.2.7. Options for the grading

By setting

`NoGradingDenom`

you can force Normaliz not to change the original grading if it would otherwise divide it by the grading denominator. It is implied by several computation goals for polytopes. See Section 6.1.

By

`GradingIsPositive`

the user guarantees that the grading is positive. This option can be useful in rare cases if Normaliz would otherwise compute extreme rays only to check the positivity of the grading.

4.3. Computation goals

The computation goal `Sublattice` does not imply any other computation goal. Most other computation goals include `Sublattice` and `SupportHyperplanes`. The exceptions are:

(1) certain computations based on the dual algorithm or i see Section 4.6;
(2) `Projection` or `ProjectionFloat` app,ied to parallelotopes; see Section 4.6;
(3) computations done completely by symmetrization.

If you are in doubt whether your desired data will be computed, add an explicit computation goal.

4.3.1. Lattice data

`Sublattice, -S` (upper case S) asks Normaliz to compute the coordinate transformation to and from the efficient sublattice.

4.3.2. Support hyperplanes and extreme rays

`SupportHyperplanes, -s` triggers the computation of support hyperplanes and extreme rays.

Normaliz tries to find a grading.

VerticesFloat converts the format of the vertices to floating point. It implies SupportHyperplanes.

SuppHypsFloat converts the format of the support hyperplanes to floating point. It implies SupportHyperplanes.

Note that VerticesFloat iand SuppHypsFloat are not pure output options. They are computation goals, and therefore break implicit DefaultMode.

ProjectCone Normaliz projects the cone defined by the input data onto a subspace generated by selected coordinate vectors and computes the image with the goal SupportHyperplanes.

4.3.3. Hilbert basis and lattice points

HilbertBasis, -N triggers the computation of the Hilbert basis. In inhomogeneous computations it asks for the Hilbert basis of the recession monoid *and* the module generators.

Deg1Elements, -1 restricts the computation to the degree 1 elements of the Hilbert basis in homogeneous computations (where it requires the presence of a grading).

LatticePoints is identical to Deg1Eelements in the homogenous case, but implies NoGradingDenom. In inhomogeneous computations it is a synonym for HilbertBasis.

ModuleGeneratorsOverOriginalMonoid, -M computes a minimal system of generators of the integral closure over the original monoid (see Section 6.17). Requires the existence of original monoid generators.

The boolean valued computation goal IsIntegrallyClosed is also related to the Hilbert basis; see Section 4.3.10.

4.3.4. Enumerative data

The computation goals in this section require a grading. They include SupportHyperplanes.

HilbertSeries,-q triggers the computation of the Hilbert series.

EhrhartSeries computes the Ehrhart series of a polytope, regardless of whether it is defined by homogeneous or inhomogeneous input. In the homogeneous case it is equivalent to HilbertSeries + NoGradingDenom, but not in the inhomogeneous case. See the deiscussion in Section 6.1. Can be combined with HSOP.

Multiplicity, -v restricts the computation to the multiplicity.

Volume, -V computes the lattice normalized and the Euclidean volume of a polytope given by homogeneous or inhomogeneous input (implies Multiplicity in the homogeneous case, but also sets NoGradingDenom).

HSOP lets Normaliz compute the degrees in a homogeneous system of parameters and the induced representation of the Hilbert series.

NoPeriodBound This option removes the period bound that Normaliz sets for the computation of the Hilbert quasipolynomial (presently 10^6).

4.3.5. Combined computation goals

Can only be set by single letter command line options:

-n `HilbertBasis` + `Multiplicity`
-h `HilbertBasis` + `HilbertSeries`
-p `Deg1Elements` + `HilbertSeries`

4.3.6. The class group

`ClassGroup, -C` is self explanatory, includes `SupportHyperplanes`. Not allowed in inhomogeneous computations.

4.3.7. Integer hull

`IntegerHull, -H` computes the integer hull of a polyhedron. Implies the computation of the lattice points in it.

More precisely: in homogeneous computations it implies `Deg1Elements`, in inhomogeneous computations it implies `HilbertBasis`. See Section 2.14.

4.3.8. Triangulation and Stanley decomposition

`Triangulation, -T` makes Normaliz compute, store and export the full triangulation.
`ConeDecomposition, -D` Normaliz computes a disjoint decomposition of the cone into semiopen simplicial cones. Implies `Triangulation`.
`TriangulationSize, -t` makes Normaliz count the simplicial cones in the full triangulation.
`TriangulationDetSum` makes Normaliz additionally sum the absolute values of their determinants.
`StanleyDec, -y` makes Normaliz compute, store and export the Stanley decomposition. Only allowed in homogeneous computations.

The triangulation and the Stanley decomposition are treated separately since they can become very large and may exhaust memory if they must be stored for output.

Note that these decompositions cannot be computed for a polyhedron that is unbounded (modulo its maximal subspace).

4.3.9. Weighted Ehrhart series and integrals

`WeightedEhrhartSeries, -E` makes Normaliz compute a generalized Ehrhart series.
`VirtualMultiplicity, -L` makes Normaliz compute the virtual multiplicity of a weighted Ehrhart series.
`Integral, -I` makes Normaliz compute an integral over a polytope. Implies `NoGradingDenom`.

These computation goals require a homogeneous computation.

Don't confuse these options with symmetrization. The latter symmetrizes (if possible) the given data and uses `-E` or `-L` internally on the symmetrized object. The options `-E,-I,-L` ask for the input of a polynomial. See Section 3.1.8.

4.3.10. Boolean valued computation goals

They tell Normaliz to find out the answers to the questions they ask. Two of them are more important than the others since they may influence the course of the computations:

`IsIntegrallyClosed, -w` : is the original monoid integrally closed? Normaliz stops the Hilbert basis computation as soon as it can decide whether the original monoid contains the Hilbert basis (see Section 2.11.1). If the answer is 'no", Normaliz computes a witness, an element of the integral closure that is not contained in the original monoid.

`IsPointed` : is the efficient cone $\mathbb{C}$ pointed? This computation goal is sometimes useful to give Normaliz a hint that a nonpointed cone is to be expected. See Section 6.14.3.

For the following we only need the support hyperplanes and the lattice:

`IsGorenstein, -G` : is the monoid of lattice points Gorenstein? In addition to answering this question, Normaliz also computes the generator of the interior of the monoid (the canonical module) if the monoid is Gorenstein.

The remaining ones:

`IsDeg1ExtremeRays` : do the extreme rays have degree 1?

`IsDeg1HilbertBasis` : do the Hilbert basis elements have degree 1?

`IsReesPrimary` : for the input type `rees_algebra`, is the monomial ideal primary to the irrelevant maximal ideal?

The last three computation goals are not really useful for Normaliz since they will be answered automatically. Note that they may trigger extensive computations.

4.4. Integer type

There is no need to worry about the integer type chosen by Normaliz. All preparatory computations use infinite precision. The main computation is then tried with 64 bit integers. If it fails, it will be restarted with infinite precision.

Infinite precision does not mean that overflows are completely impossible. In fact, Normaliz requires numbers of type ''degree" fit the type `long` (typically 64 bit on 64 bit systems). If an overflow occurs in the computation of such a number, it cannot be remedied.

The amount of computations done with infinite precision is usually very small, but the transformation of the computation results from 64 bit integers to infinite precision may take some time. If you need the highest possible speed, you can suppress infinite precision completely by

`LongLong`

With this option, Normaliz cannot restart a failed computation.

On the other hand, the 64 bit attempt can be bypassed by

BigInt, -B

Note that Normaliz tries to avoid overflows by intermediate results (even if LongLong is set). If such overflow should happen, the computation is repeated locally with infinite precision. (The number of such GMP transitions is shown in the terminal output.) If a final result is too large, Normaliz must restart the computation globally.

LongLong is not a cone property.

Caveat. The overflow check of Normaliz is not an absolute guarantee. The probability that it fails is microscopically small, but failure is not totally excluded. Very critical computations for which one has no other confirmation should be redone in BigInt.

4.5. Control of computations and communication with interfaces

In addition to the computation goals in Section 4.3, the following elements of ConeProperties control the work flow in libnormaliz and can be used by programs calling Normaliz to ensure the availability of the data that are controlled by them.

Generators controls the generators of the efficient cone.

OriginalMonoidGenerators controls the generators of the original monoid.

ModuleGenerators controls the module generators in inhomogeneous computation.

ExtremeRays controls the extreme rays.

VerticesOfPolyhedron controls the vertices of the polyhedron in the inhomogeneous case.

MaximalSubspace controls the maximal linear subspace of the (homogenized) cone.

EmbeddingDim controls the embedding dimension.

Rank controls the rank.

RecessionRank controls the rank of the recession monoid in inhomogeneous computations.

AffineDim controls the affine dimension of the polyhedron in inhomogeneous computations.

ModuleRank in inhomogeneous computations it controls the rank of the module of lattice points in the polyhedron as a module over the recession monoid.

ExcludedFaces controls the excluded faces.

InclusionExclusionData controls data derived from the excluded faces.

Grading controls the grading.

GradingDenom controls its denominator.

Dehomogenization controls the dehomogenization.

ReesPrimaryMultiplicity controls the multiplicity of a monomial ideal, provided it is primary to the maximal ideal generated by the indeterminates. Used only with the input type rees_algebra.

EuclideanVolume controls the Euclidean volume.

WitnessNotIntegrallyClosed controls witness against integral closedness.

GeneratorOfInterior controls the generator of the interior if the monoid is Gorenstein.

Equations controls the equations.

Congruences controls the congruences.
ExternalIndex controls the external index.
InternalIndex controls the internal index.
UnitGroupIndex controls the unit group index.
IsInhomogeneous controls the inhomogeneous case..
HilbertQuasiPolynomial controls the Hilbert quasipolynomial.
EhrhartQuasiPolynomial controls the Ehrhart quasipolynomial.
WeightedEhrhartQuasiPolynomial controls the weighted Ehrhart quasipolynomial.
IsTriangulationNested controls the indicator of this property.
IsTriangulationPartial similar.

4.6. Rational and integer solutions in the inhomogeneous case

The integer solutions of a homogeneous diophantine system generate the rational solutions as well: every rational solution has a multiple that is an integer solution. Therefore the rational solutions do not need an extra computation. If you prefer geometric language: a rational cone is generated by its lattice points.

This is no longer true in the inhomogeneous case where the computation of the rational solutions is an extra task for Normaliz. This extra step is inevitable for the primal algorithm, but not for the dual algorithm. In general, the computation of the rational solutions is much faster than the computation of the integral solutions, but this by no means always the case.

Therefore we have decoupled the two computations if the dual algorithm is applied to inhomogeneous systems or to the computation of degree 1 points in the homogeneous case. The combinations

DualMode HilbertBasis, -dN
DualMode Deg1Elements, -d1
DualMode ModuleGenerators

do not imply the computation goal `SupportHyperplanes` (and not even `Sublattice`) which would trigger the computation of the rational solutions (geometrically: the vertices of the polyhedron). If you want to compute them, you must add one of

SupportHyperplanes, -s
ExtremeRays
VerticesOfPolyhedron

The last choice is only possible in the inhomogeneous case. Another possibility in the inhomogeneous case is is to use `DualMode` without `-N`.

If `Projection` or `ProjectionFloat` is used for parallelotopes defined by inequalities, then Normaliz does not compute the vertices, unless asked for by one of the three computation goals just mentioned.

5. Running Normaliz

The standard form for calling Normaliz is

```
normaliz [options] <project>
```

where `<project>` is the name of the project, and the corresponding input file is `<project>.in`. Note that `normaliz` may require to be prefixed by a path name, and the same applies to `<project>`. A typical example on a Linux or Mac system:

```
./normaliz --verbose -x=5 example/big
```

that for MS Windows must be converted to

```
.\normaliz --verbose -x=5 example\big
```

Normaliz uses the standard conventions for calls from the command line:

(1) the order of the arguments on the command line is arbitrary.
(2) Single letter options are prefixed by the character `-` and can be grouped into one string.
(3) Verbatim options are prefixed by the characters `--`.

The options for computation goals and algorithmic variants have been described in Section 4. In this section the remaining options for the control of execution and output are discussed, together with some basic rules for the use of the options.

5.1. Basic rules

The options for computation goals and algorithms variants have been explained in Section 4. The options that control the execution and the amount of output will be explained in the following. Basic rules for the use of options:

1. If no `<project>` is given, the program will terminate.
2. The option `-x` differs from the other ones: `<T>` in `-x=<T>` represents a positive number assigned to `-x`; see Section 5.3.
3. Similarly the option `--OutputDir=<outdir>` sets the output directory; see 5.5.
4. Normaliz will look for `<project>.in` as input file.

   If you inadvertently typed `rafa2416.in` as the project name, then Normaliz will first look for `rafa2416.in.in` as the input file. If this file doesn't exist, `rafa2416.in` will be loaded.
5. The options can be given in arbitrary order. All options, including those in the input file, are accumulated, and syntactically there is no mutual exclusion. However, some options may block others during the computation. For example, `KeepOrder` blocks `BottomDecomposition`.
6. If Normaliz cannot perform a computation explicitly asked for by the user, it will terminate. Typically this happens if no grading is given although it is necessary.
7. In the options include `DefaultMode`, Normaliz does not complain about missing data

(anymore). It will simply omit those computations that are impossible.

8. If a certain type of computation is not asked for explicitly, but can painlessly be produced as a side effect, Normaliz will compute it. For example, as soon as a grading is present and the Hilbert basis is computed, the degree 1 elements of the Hilbert basis are selected from it.

5.2. Info about Normaliz

`--help, -?` displays a help screen listing the Normaliz options.
`--version` displays information about the Normaliz executable.

5.3. Control of execution

The options that control the execution are:

`--verbose, -c` activates the verbose ("console") behavior of Normaliz in which Normaliz writes additional information about its current activities to the standard output.

`-x=<T>` Here `<T>` stands for a positive integer limiting the number of threads that Normaliz is allowed access on your system. The default value is 8. (Your operating system may set a lower limit).

`-x=0` switches off the limit set by Normaliz.

If you want to run Normaliz in a strictly serial mode, choose `-x=1`.

The number of threads can also be controlled by the environment variable `OMP_NUM_THREADS`. See Section 8.1 for further discussion.

5.4. Interruption

During a computation `normaliz` can be interrupted by pressing Ctrl-C on the keyboard. If this happens, Normaliz will stop the current computation and write the already computed data to the output file(s).

At present, the Normaliz interrupt control has no effect during SCIP computations.

If Crtl-C is pressed during the output phase, Normaliz is stopped immediately.

5.5. Control of output files

In the default setting Normaliz writes only the output file `<project>.out` (and the files produced by `Triangulation` and `StanleyDec`). The amount of output files can be increased as follows:

`--files, -f` Normaliz writes the additional output files with suffixes `gen`, `cst`, and `inv`, provided the data of these files have been computed.

--all-files, -a includes `Files`, Normaliz writes all available output files (except `typ`, the triangulation or the Stanley decomposition, unless these have been requested).

--<suffix> chooses the output file with suffix `<suffix>`.

For the list of potential output files, their suffixes and their interpretation see Section 7. There may be several options `--<suffix>`.

If the computation goal `IntegerHull` is set, Normaliz computes a second cone and lattice. The output is contained in `<project>.IntHull.out`. The options for the output of `<project>` are applied to `<project>.IntHull` as well. There is no way to control the output of the two computations individually.

Similarly, if symmetrization has been used, Normaliz writes the file `<project>.symm.out`. It contains the data of the symmetrized cone.

Sometimes one wants the output to be written to another directory. The output directory can be set by

--OutputDir=<outdir> . The path `<outdir<` is an absolute path or a path relative to the current directory (which is not necessarily the directory of `<project>.in`.)

Note that all output files will be written to the chosen directory. It must be created before Normaliz is started.

Extreme rays and vertices may have very long integer coordinates. One can suppress their output by

NoExtRaysOutput

For similar reasons one may want to suppress the output of support hyperplanes, namely by

NoSuppHypsOutput

`NoExtRaysOutput` and `NoSuppHypsOutput` are not cone properties.

5.6. Ignoring the options in the input file

Since Normaliz accumulates options, one cannot get rid of settings in the input file by command line options unless one uses

--ignore, -i This option disables all settings in the input file.

6. Advanced topics

6.1. Computations with a polytope

In many cases the starting point of a computation is a polytope, i.e., a bounded polyhedron – and not a cone or monoid. Normaliz offers two types of input for polytopes that allow almost the same computations, namely

(1) *homogeneous* input type for which the polytope is the intersection of a cone with a

hyperplane defined by the grading (automatically bounded): $P = \{x \in C : \deg x = 1\}$.

(2) *inhomogeneous* input defining a polytope (and not an unbounded polyhedron).

A problem that can arise with homogeneous input is the appearance of a grading enumerator $g > 1$. In this case the polytope P defined by the input gading is replaced by gP. This may be undesirable and can therefore be blocked by `NoGradingDenom`. Note: a grading denominator $g > 1$ can only appear if the affine space spanned by the polytope does not contain a lattice point. This is a rare situation, but nevertheless you may want to safeguard against it.

Computation goals whose names have a "polytopal touch" (as opposed to "algebraic touch") set `NoGradingDenom` automatically. These computation goals are also to be used with inhomogeneous input; see the following table. The homogeneous input type `polytope` sets `NoGradingDenom` as well.

In the following table L is the lattice of reference defined by the input data.

desired data	inhom input or hom input blocking grading denominator	hom input allowing grading denominator
lattice points	`LatticePoints`	`Deg1Elements`
generating function of $k \mapsto \#(kP \cap L)$	`EhrhartSeries`	`HilbertSeries`
volume or multiplicity	`Volume`	`Multiplicity`
integral	`Integral`	—

Note that `HilbertSeries` and `Multiplicity` make also sense with inhomogeneous input, but they refer to a different counting function, namely

$$k \mapsto \#(x \in P \cap L, \deg x = k).$$

Even if P is a polytope, this function has applications; see Section 6.10.2.

With homogeneous input `IntegerHull` does not imply `NoGradingDenom`. If necessary you must set in explicitly.

6.1.1. Lattice normalized and Euclidean volume

As just mentioned, for polytopes defined by homogeneous input Normaliz has two computation goals, `Multipicity`, `-v`, and `Volume, -V`, that are almost identical: `Voulme = Multiplicity + NoGradingDenom`. Both compute the lattice normalized volume; moreover, `Volume` additionally computes the Euclidean volume and can also be used with inhomogeneous input, for which `Multiplicity` has a different meaning. (For the algebraic origin of `Multiplicity` See Appendix A.7.)

In the following we want to clarify the notion of *lattice normalized volume*.

(1) Let $P \subset \mathbb{R}^d$ be a polytope of dimension r and let A be the affine subsapce spanned by P. Then the Euclidean volume $\mathrm{vol}_{\mathrm{eucl}}(P)$ of P is computed with respect to the r-dimensional Lebesgue measure in which an r-dimensional cube in A of edge length 1 has measure 1.

(2) For the lattice normalized volume we need a lattice L of reference. We assume that $\mathrm{aff}(P) \subset \mathrm{aff}(L)$. (It would be enough to have this inclusion after a parallel translation of $\mathrm{aff}(P)$.) Choosing the origin in L, one can assume that $\mathrm{aff}(L)$ is a vector subspace of $\mathbb{R}^d$ so that we can identify it with $\mathbb{R}^d$ after changing d if necessary. After a coordinate transformation we can further assume that $L = \mathbb{Z}^d$ (in general this is not an orthogonal change of coordinates!). To continue we need that $\mathrm{aff}(P)$ is a rational subspace. There exists $k \in \mathbb{N}$ such that $k\,\mathrm{aff}(P)$ contains a lattice simplex. The lattice normalized volume vol_L of kP is then given by the Lebesgue measure on $k\,\mathrm{aff}(P)$ in which the smallest possible lattice simplex in $k\,\mathrm{aff}(P)$ has volume 1. Finally we set $\mathrm{vol}_L(P) = \mathrm{vol}_L(kP)/k^r$ where $r = \dim(P)$.

If P is a full-dimensional polytope in $\mathbb{R}^d$ and $L = \mathbb{Z}^d$, then $\mathrm{vol}_L(P) = d!\,\mathrm{vol}_{\mathrm{eucl}}(P)$, but in general the correction factor is $cr!$ with c depending on $\mathrm{aff}(P)$: the line segment in $\mathbb{R}^2$ connecting $(1,0)$ and $(0,1)$ has euclidean length $\sqrt{2}$, but lattice normalized volume 1. As this simple example shows, c can be irrational.

6.1.2. Developer's choice: homogeneous input

Our recommendation: if you have the choice between homogeneous and inhomogeneous input, go homogeneous. You do not lose any computation goal and can only gain efficiency.

6.2. Lattice points in polytopes once more

Normaliz has three main algorithms for the computation of lattice points of which two have two variants each:

(1) the project-and-lift algorithm (`Projection, -j`),
(2) its variant using floating point arithmetic (`ProjectionFloat, -J`),
(3) the triangulation based Normaliz primal algorithm specialized to lattice points (`PrimalMode, -P`),
(4) its variant using approximation of rational polytopes (`Approximate, -r`),
(5) the dual algorithm specialized to lattice points (`DualMode, -d`).

The options `Projection`, `ProjectionFloat` and `Approximate` do not implya computation goal. Since `PrimalMode` can also be used for the computation of Hilbert series and Hilbert bases, one must add the computation goal to it. In the homogeneous case one must add the computation goal also to `DualMode`.

Remark. The triangulation based primal algorithm and the dual algorithm do not depend on the embedding of the computed objects into the ambient space since they use only data that are invariant under coordinate transformations. This is not true for project-and-lift and the approximation discussed below.`Projection` and `ProjectionFloat` (and in certain cases also `PrimalMode`) profit significantly from LLL reduced coordinates (since version 3.4.1). We

discuss this feature in Section 6.2.3.

We recommend the reader to experiment with the following input files:

- `5x5.in`
- `6x6.in`
- `max_polytope_cand.in`
- `hickerson-18.in`
- `knapsack_11_60.in`
- `ChF_2_64.in`
- `ChF_8_1024.in`
- `VdM_16_1048576.in` (may take some time)
- `pedro2.in`

In certain cases you must use `-i` on the command line to override the options in the input file.

`max_polytope_cand.in` came up in connection with the paper "Quantum jumps of normal polytopes" by W. Bruns, J. Gubeladze and M. Michałek, Discrete Comput. Geom. 56 (2016), no. 1, 181–215. `hickerson-18.in` is taken from the LattE distribution [4]. `pedro2.in` was suggested by P. Garcia-Sanchez.

The files `ChF*.in` and `VdM*.in` are taken from the paper "On the orthogonality of the Chebyshev-Frolov lattice and applications" by Ch. Kacwin, J. Oettershagen and T. Ullrich (lattice and applications, Monatsh. Math. 184 (2017), 425–441). The file `VdM_16_1048576.in` is based on the linear map given directly by the Vandermonde matrix. A major point of the paper is a coordinate transformation that simplifies computations significantly, and the files `ChF*.in` are based on it.

6.2.1. Project-and-lift

We have explained the project-and-lift algorithm in Section 2.13. This algorithm is very robust arithmetically since it needs not compute determinants or solve systems of linear equations. Moreover, the project-and-lift algorithm itself does not use the vertices of the polytope explicitly and only computes lattice points in P and its successive projections. Therefore it is rather insensitive against rational vertices with large denominators. (To get started it must usually compute the vertices of the input polytope; an exception are parallelotopes, as mentioned in Section 2.13 and discussed below.)

The option for project-and-lift is

`Projection, -j`

There are two complications that may slow it down unexpectedly: (i) the projections may have large numbers of support hyperplanes, as seen in the example `VdM_16_1048576.in` (it uses floating point arithmetic in the lifting part):

```
Projection
embdim 17 inequalities 32
embdim 16 inequalities 240
...
```

```
embdim 11 inequalities 22880
embdim 10 inequalities 25740
embdim 9 inequalities 22880
...
embdim 3 inequalities 32
embdim 2 inequalities 2
```

(ii) The projections may have many lattice points that cannot be lifted to the top. As an example we look at the terminal output of `pedro2.in`:

```
Lifting
Lifting
embdim 2 Deg1Elements 40
embdim 3 Deg1Elements 575
embdim 4 Deg1Elements 6698
embdim 5 Deg1Elements 6698
embdim 6 Deg1Elements 2
```

Despite of these potential problems, `Projection` is the default choice of Normaliz for the computation of lattice points (if not combined with Hilbert series or Hilbert basis). If you do not want to use it, you must either choose another method explicitly or switch it off by `NoProjection`. Especially for lattice polytopes with few extreme rays, but many support hyperplanes the triangulation base algorithm is often the better choice.

Parallelotopes. Lattice points in parallelotopes that are defined by inequalities, like those in the input files `VdM*.in`, can be computed without any knowledge of the vertices. In fact, for them it is favorable to present a face F by the list of facets whose intersection F is (and not by the list of the $2^{\dim F}$ vertices of F!). Parallelotopes are not only simple polytopes. It is important that two faces do not intersect if and only if they are contained in parallel facets, and this is easy to control. Normaliz recognizes parallelotopes by itself, and suppresses the computation of the vertices unless asked to compute them.

6.2.2. Project-and-lift with floating point arithmetic

Especially the input of floating point numbers often forces Normaliz into GMP arithmetic. Since GMP arithmetic is slow (compared to arithmetic with machine integers or floating point numbers), Normaliz has a floating point variant of the project-and-lift algorithm. (Such an algorithm makes no sense for Hilbert bases or Hilbert series.) It behaves very well, even in computations for lower dimensional polytopes. We have not found a single deviation from the results with GMP arithmetic in our examples. Nevertheless, the projection phase is done in the in integer arithmetic, and only the lifting uses floating point.

The option for the floating point variant of project-and-lift is

`ProjectionFloat, -J`

If you want a clear demonstration of the difference between `Projection` and `ProjectionFloat`, try `VdM_16_1048576.in`.

The use of `ProjectionFloat` or any other algorithmic variant is independent of the input type. The coordinates of the lattice points computed by `ProjectionFloat` are assumed to be at most 64 bits wide, independently of the surrounding integer type. If this condition should not be satisfied in your application, you must use `Projection` instead.

6.2.3. LLL reduced coordinates and relaxation

The project-and-lift algorithm depends very much on the embedding of the polytope in the ambient space. We use LLL reduction to find coordinate transformations of the ambient space in which the vertices of the polytope have small coordinates so that the successive projections have few lattice points. Roughly speaking, LLL reduced coordinates are computed as follows. We form a matrix A whose *rows* are the vertices or the support hypeplanes of the polytope, depending on the situation. Suppose A has d columns; A need not have integral entries, but it must have rank d. Then we apply LLL reduction to the lattice generated by the *columns* of A. This amounts to finding a matrix $U \in \mathrm{GL}(d,\mathbb{Z})$ such that the columns of AU are short vectors (in the Euclidean norm). The matrix U and its inverse then define the coordinate transformations forth and back.

Often LLL reduction has a stunning effect. We have a look at the terminal output of `pedro2.in` run with `-P`. The left column shows the present version, the right one is produced by Normaliz 3.4.0:

```
Lifting                              Lifting
embdim 2 Deg1Elements 2              embdim 2 Deg1Elements 40
embdim 3 Deg1Elements 2              embdim 3 Deg1Elements 672
embdim 4 Deg1Elements 2              embdim 4 Deg1Elements 6698
embdim 5 Deg1Elements 3              embdim 5 Deg1Elements 82616047
embdim 6 Deg1Elements 2              embdim 6 Deg1Elements 2
Project-and-lift complete            Project-and-lift complete
```

We have no example for which LLL increases the computation time. Though its application not seem to be a real disadvantage, it can be switched off for `Projection` and `ProjectionFloat` by

`NoLLL`

Without LLL certain computations are simply impossible – just try `VdM_16_1048576` with `NoLLL`. (This option is used internally to avoid a repetition of LLL computations.)

We use the original LLL original algorithm with the factor 0.9.

Another aspect of the implementation that must be mentioned is the relaxation of inequalities: for the intermediate lifting of lattice points Normaliz uses at most 1000 (carefully chosen) inequalities. Some additional intermediate lattice points are acceptable if the evaluation of inequalities is reduced by a substantial factor. On the left we see `VdM_16_1048576` with relaxation, on the right without:

```
Lifting                                  Lifting
...                                      ...
```

```
embdim 6 Deg1Elements 2653                embdim 6 Deg1Elements 2297
...                                       ...
embdim 10 Deg1Elements 431039             embdim 10 Deg1Elements 128385
embdim 11 Deg1Elements 1031859            embdim 11 Deg1Elements 277859
embdim 12 Deg1Elements 2016708            embdim 12 Deg1Elements 511507
embdim 13 Deg1Elements 2307669            embdim 13 Deg1Elements 806301
...                                       ...
```

No surprise that relaxation increases the number of intermediate lattice points, but it reduces the computation time by about a factor 2.

It is of course not impossible that relaxation exhausts RAM or extends the computation time. Therefore one can switch it off by

NoRelax

6.2.4. The triangulation based primal algorithm

With this algorithm, Normaliz computes a partial triangulation as it does for the computation of Hilbert bases (in primal mode) for the cone over the polytope. Then it computes the lattice points in each of the subpolytopes defined by the simplicial subcones in the triangulation. The difference to the Hilbert basis calculation is that all points that do not lie in our polytope P can be discarded right away and that no reduction is necessary.

The complications that can arise are (i) a large triangulation or (ii) large determinants of the simplicial cones. Normaliz tries to keep the triangulations small by restricting itself to a partial triangulation, but often there is nothing one can do. Normaliz deals with large determinants by applying project-and-lift to the simplicial subcones with large determinants. We can see this by looking at the terminal output of `max_polytope_cand.in`, computed with `-cP -x=1`:

```
...
evaluating 49 simplices
|||||||||||||||||||||||||||||||||||||||||||||||||||
49 simplices, 819 deg1 vectors accumulated.
47 large simplices stored
Evaluating 47 large simplices
Large simplex 1 / 47
************************************************************
starting primal algorithm (only support hyperplanes) ...
Generators sorted lexicographically
Start simplex 1 2 3 4 5
Pointed since graded
Select extreme rays via comparison ... done.
------------------------------------------------------------
transforming data... done.
Computing lattice points by project-and-lift
Projection
embdim 6 inequalities 7
```

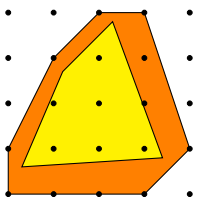
```
...
embdim 2 inequalities 2
Lifting
embdim 2 Deg1Elements 9
...
embdim 6 Deg1Elements 5286
Project-and-lift complete
...
```

After finishing the 49 "small" simplicial cones, Normaliz takes on the 47 "large" simplicial cones, and does them by project-and-lift (including LLL). Therefore one can say that Normaliz takes a hybrid approach to lattice points in primal mode.

An inherent weakness of the triangulation based algorithm is that its efficiency drops with $d!$ where d is the dimension because the proportion of lattice points in P of all points generated by the algorithm must be expected to be $1/d!$ (as long as small simplicial cones are evaluated). To some extent this is compensated by the extremely dast generation of the candidates.

6.2.5. Lattice points by approximation

Large determinants come up easily for rational polytopes P whose vertices have large denominators. In previous versions, Normaliz fought against large determinants coming from rational vertices by finding an integral polytope Q containing P, computing the lattice points in Q and then sieving out those that are in $Q \setminus P$:



This approach is still possible. It is requested by the option

`Approximate, -r`

This is often a good choice, especially in low dimension.

It is not advisable to use approximation for polytopes with a large number of vertices since it must be expected that the approximation multiplies the number of vertices by $\dim P + 1$ so that it may become difficult to compute the triangulation.

6.2.6. Lattice points by the dual algorithm

Often the dual algorithm is extremely fast. But it can also degenerate terribly. It is very fast for `6x6.in` run with `-d1`. The primal algorithm or approximation fail miserably. (`-1`, the default choice project-and-lift, is also quite good. The difference is that `-d1` does not compute the vertices that in this case are necessary for the preparation of project-and-lift.)

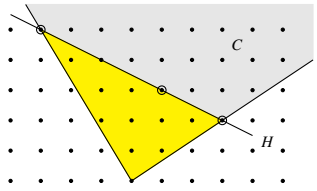
On the other hand, the dual algorithm is hopeless already for the 2-dimensional parallelotope `ChF_2_64.in`. Try it. It is clear that its complicated arithmetic is forbidding for the dual algorithm. (The dual algorithm successively computes the lattice points correctly for all intermediate polyhedra, defined as intersections of the half spaces that have been processed so far. The intermediate polyhedra can be much more difficult than the final polytope, as in this case.)

In certain cases (see Section 6.5) Normaliz will try the dual algorithm if you forbid project-and-lift by `NoProjection`.

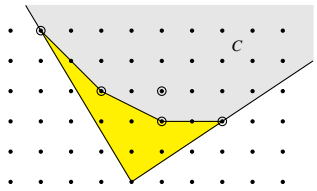
6.3. The bottom decomposition

The triangulation size and the determinant sum of the triangulation are critical size parameters in Normaliz computations. Normaliz always tries to order the generators in such a way that the determinant sum is close to the minimum, and on the whole this works out well. The use of the bottom decomposition by `BottomDecomposition, -b` enables Normaliz to compute a triangulation with the optimal determinant sum for the given set of generators, as we will explain in the following.

The determinant sum is independent of the order of the generators of the cone C if they lie in a hyperplane H. Then the determinant sum is exactly the normalized volume of the polytope spanned by 0 and $C \cap H$. The triangulation itself depends on the order, but the determinant sum is constant.



This observation helps to find a triangulation with minimal determinant sum in the general case. We look at the *bottom* (the union of the compact faces) of the polyhedron generated by $x_1, \ldots, x_n$ as vertices and C as recession cone, and take the volume underneath the bottom:



With the option `BottomDecomposition, -b`, Normaliz computes a triangulation that respects the bottom facets. This yields the optimal determinant sum for the given generators. If one can compute the Hilbert basis by the dual algorithm, it can be used as input, and then one obtains the absolute bottom of the cone, namely the compact facets of the convex hull of all nonzero lattice points.

Normaliz does not always use the bottom decomposition by default since its computation requires some time and administrative overhead. However, as soon as the input "profile" is considered to be "rough" it is invoked. The measure of roughness is the ratio between the maximum degree (or L_1 norm without a grading) and the minimum. A ratio ≥ 10 activates the bottom decomposition.

If you have the impression that the bottom decomposition slows down your computation, you can suppress it by `NoBottomDec`, `-o`.

The bottom decomposition is part of the subdivision of large simplicial cones discussed in the next section.

The example `strictBorda.in` belongs to social choice theory like `Condorcet.in` (see Section 2.10), `PluralityVsCutoff.in` and `CondEffPlur.in`. The last two profit enormously from symmetrization (see Section 6.8), but `strictBorda.in` does not. Therefore we must compute the Hilbert series for a monoid in dimension 24 whose cone has 6363 extreme rays. It demonstrates the substantial gain that can be reached by bottom decomposition. Since the roughness is large enough, Normaliz chooses bottom decomposition automatically, unless we block it.

algorithm	triangulation size	determinant sum
bottom decomposition	30,399,162,846	75,933,588,203
standard order of extreme rays, `-o`	119,787,935,829	401,249,361,966

6.4. Subdivision of large simplicial cones

Especially in computations with rational polytopes one encounters very large determinants that can keep the Normaliz primal algorithm from terminating in reasonable time. As an example we take `hickerson-18.in` from the LattE distribution [4]. It is simplicial and the complexity is totally determined by the large determinant $\approx 4.17 \times 10^{14}$ (computed with `-v`).

If we are just interested in the degree 1 points, Normaliz uses the project-and-lift method of Section 6.2.1 and finds 44 degree 1 points in the blink of an eye. If we use these points together with the extreme rays of the simplicial cone, then the determinant sum decreases to $\approx 1.3 \times 10^{12}$, and the computation of the Hilbert basis and the Hilbert series is in reach. But it is better to pursue the idea of subdividing large simplicial cones systematically. Normaliz employs two methods:

(1) computation of subdivision points by the IP solver SCIP,
(2) its own algorithm for finding optimal subdivision points, based on project-and-lift(and LLL reduced coordinates)

Normaliz tries to subdivide a simplicial cone if it has determinant $\geq 10^8$ or 10^7 if the Hilbert basis is computed. Both methods are used recursively via stellar subdivision until simplicial cones with determinant $< 10^6$ have been reached or no further improvement is possible. All subdivision points are then collected, and the start simplicial cone is subdivided with bottom decomposition, which in general leads to substantial further improvement.

The use of SCIP requires a Normaliz executable built with SCIP see Section 10). Moreover,

the option `SCIP` must be set since in many cases the Normaliz method is faster and always finds a subdivision point if such exists.

The following table contains some performance data for subdivisions based on the Normaliz method (default mode, parallelization with 8 threads).

	`hickerson-16`	`hickerson-18`	`knapsack_11_60`
simplex volume	9.83×10^7	4.17×10^{14}	2.8×10^{14}
stellar determinant sum	3.93×10^6	9.07×10^8	1.15×10^8
volume under bottom	8.10×10^5	3.86×10^7	2.02×10^7
volume used	3.93×10^6	6.56×10^7	2.61×10^7
runtime without subdivision	2.8 s	>12 d	>8 d
runtime with subdivision	0.4 s	24 s	5.1 s

A good nonsimplicial example showing the subdivision at work is `hickerson-18plus1.in` with option `-q`.

Note: After subdivision the decomposition of the cone may no longer be a triangulation in the strict sense, but a decomposition that we call a *nested triangulation*; see 6.15.1. If the creation of a nested triangulation must be blocked, one uses the option `NoSubdivision`. Inevitably it blocks the subdivision of large simplicial cones.

Remark The bounds mentioned above work well up to dimension ≈ 10. For a fixed determinant, the probability for finding a subdivision point decreases rapidly.

6.5. Primal vs. dual – division of labor

As already mentioned several times, Normaliz has two main algorithms for the computation of Hilbert bases and degree 1 points, the primal algorithm and the dual algorithm. It is in general very hard to decide beforehand which of the two is better for a specific example. Nevertheless Normaliz ties to guess, unless `PrimalMode`, `-P` or `DualMode`, `-d` is explicitly chosen by the user. In first approximation one can day that the dual algorithm is chosen if the computation is based on constraints and the number of inequalities is neither too small nor too large. Normaliz chooses the dual algorithm if at the start of the Hilbert basis computation the cone is defined by s inequalities such that

$$r + \frac{50}{r} \leq s \leq 2e$$

where r is the rank of the monoid to be computed and e is the dimension of the space in which the data are embedded. These conditions ate typically fulfilled for diophantine systems of equations whose nonnegative solutions are asked for. In the case of very few or many hyperplanes Normaliz prefers the primal algorithm. While this combinatorial condition is the only criterion for Normaliz, it depends also on the arithmetic of the example what algorithm is better. At present Normaliz makes no attempt to measure it in some way.

When both Hilbert basis and Hilbert series are to be computed, the best solution can be the combination of both algorithms. We recommend `2equations.in` as a demonstration example which combines the algorithmic variant `DualMode` and the computation goal `HilbertSeries`:

```
amb_space 9
equations 2
1 6 -7 -18 25 -36 6 8 -9
7 -13 15 6 -9 -8 11 12 -2
total_degree
DualMode
HilbertSeries
```

As you will see, the subdivision of large simplicial cones is very useful for such computations.

Compare `2equations.in` and `2equations_default.in` for an impression on the relation between the algorithms.

6.6. Volume by descent in the face lattice

As discussed above, for Hilbert basis computations it is often better to avoid the triangulation based primal algorithm if the input is in terms of inequalities (and other constraints). The same is true for volume or multiplicity computations. Under the same condition as for the automatic choice of the dual algorithm (see Section 6.5), Normaliz chooses a method m for volume computation that is based on descent in the face lattice.

While the descent algorithm is not chosen automatically if the number of inequalities exceeds $2 * \dim$, it is often very fast also for larger numbers of inequalities, as long as the number of inequalities is moderate. Typical examples are `lo6.in` and `bo5.in`. In both cases, descent is much faster than triangulation which has size $\approx 5 * 10^9$ and $\approx 20 * 10^9$, respectively.

Let us come back to `strictBorda.in`. If we run it with `-c`, the terminal output contains

```
Multiplicity by descent in the face lattice
Polytope is not simple
Descent from dim 24, size 1
Descent from dim 23, size 3
...
Descent from dim 12, size 109351
...
Descent from dim 6, size 752392
...
Descent from dim 3, size 106379
................................................
Mult 1281727528386311499990911876166511/259402550584412815249731747840000000000
Mult (float) 4.94107527277e-05
Full tree size 20378050380
Number of descent steps 9374850
Determinants computed 901955
```

```
Number of faces in descent system 4407824
Multiplicity by descent done
```

The computation is done in minutes whereas the triangulation based algorithm takes several hours. The data below the multiplicity give some information on the complexity of the computation. The full tree size indicates the size of a triangulation that could be constructed by the same recursion. (We use a 64 bit integer for this magnitude. It is not difficult to make examples that exceed $2^{64}-1$ so that a wrong number is not excluded.)

The idea is to exploit the formula

$$\mathrm{mult}(P) = \sum_i \mathrm{ht}_{F_i}(v)\,\mathrm{mult}(F_i)/\deg(v).$$

recursively where v is a vertex of the polytope P with as few opposite facets F_i as possible, and $\mathrm{ht}_{F_i}(v)$ is the lattice height of v over F_i. This requires building a subset $\mathscr{F}$ of the face lattice so that for each face F to which the formula is applied all the facets of F are contained in $\mathscr{F}$. However, if a face is simplicial, its multiplicity is computed by the standard determinantal formula. The algorithm is implemented in such a way that all data are collected in the descent and no backtracking is necessary. The RAM usage is essentially determined by the two largest layers.

As said above, this algorithm is often chosen by default. You can force it by

`Descent, -F`

and block it by

`NoDescent`

Note that `Descent` does *not* imply `Multiplicity` or `Volume`. (We cannot exclude that in the future descent is used also for other computations.)

6.7. Checking the Gorenstein property

If the Hilbert series has been computed, one can immediately see whether the monoid computed by Normaliz is Gorenstein: this is the case if and only if the numerator is a symmetric polynomial, and Normaliz indicates that (see Section 2.8). However, there is a much more efficient way to check the Gorenstein property, which does not even require the existence of a grading: we must test whether the *dual* cone has degree 1 extreme rays. This amounts to checking the existence of an implicit grading on the dual cone.

This very efficient Gorenstein test is activated by the option `IsGorenstein`, equivalently `-G` on the command line. We take `5x5Gorenstein.in`:

```
amb_space 25
equations 11
1 1 1 1 1 -1 -1 -1 -1 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
...
1 1 1 1 0  0  0  0 -1  0  0  0 -1  0  0  0 -1  0  0  0 -1  0  0  0  0
IsGorenstein
```

In the output we see

```
Monoid is Gorenstein
Generator of interior
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

In fact, the Gorenstein property is (also) equivalent to the fact that the interior of our monoid is generated by a single element as an ideal, and this generator is computed if the monoid is Gorenstein. (It defines the grading under which the extreme rays of the dual cone have degree 1.)

If the monoid is not Gorenstein, Normaliz will print the corresponding message.

6.8. Symmetrization

Under certain conditions one can count lattice points in a cone C by mapping C to a cone C' of lower dimension and then counting each lattice point y in C' with the number of its lattice preimages. This approach works well if the number of preimages is given by a polynomial in the coordinates of y. Since C' has lower dimension, one can hope that its combinatorial structure is much simpler that that of C. One must of course pay a price: instead of counting each lattice point with the weight 1, one must count it with a polynomial weight. This amounts to a computation of a weighted Ehrhart series that we will discuss in Section 6.9. Similarly multiplicity can be computed as the virtual multiplicity of a polynomial after projection.

The availability of this approach depends on symmetries in the coordinates of C, and therefore we call it *symmetrization*. Normaliz tries symmetrization under the following condition: C is given by constraints (inequalities, equations, congruences, excluded faces) and the inequalities contain the sign conditions $x_i \geq 0$ for all coordinates x_i of C. (Coordinate hyperplanes may be among the excluded faces.) Then Normaliz groups coordinates that appear in all constraints and the grading (!) with the same coefficients, and, roughly speaking, replaces them by their sum. The number of preimages that one must count for the vector y of sums is then a product of binomial coefficients – a polynomial as desired. More precisely, if y_j, $j = 1,\dots,m$, is the sum of u_j variables x_i then

$$f(y) = \binom{u_1 + y_1 - 1}{u_1 - 1} \cdots \binom{u_m + y_m - 1}{u_m - 1}.$$

is the number of preimages of $(y_1,\dots,y_m)$. This approach to Hilbert series has been suggested by A. Schï¿½rmann [18].

As an example we look again at the input for the Condorcet paradox:

```
amb_space 24
inequalities 3
1 1 1 1 1 1 -1 -1 -1 -1 -1 -1   1  1 -1 -1  1 -1   1  1 -1 -1  1 -1
1 1 1 1 1 1  1  1 -1 -1  1 -1  -1 -1 -1 -1 -1 -1   1  1  1 -1 -1 -1
1 1 1 1 1 1  1  1  1 -1 -1 -1   1  1  1 -1 -1 -1  -1 -1 -1 -1 -1 -1
nonnegative
```

```
total_degree
Multiplicity
```

The grading is completely symmetric, and it is immediately clear that the input is symmetric in the first 6 coordinates. But also the column of three entries -1 appears 6 times, and there are 6 more groups of 2 coordinates each (one group for each ± 1 pattern). With the suitable labeling, the number of preimages of$(y_1,\dots,y_8)$ is given by

$$f(y) = \binom{y_1+5}{5}(y_2+1)(y_3+1)(y_4+1)(y_5+1)(y_6+1)(y_7+1)\binom{y_8+5}{5}.$$

Normaliz finds the groups of variables that appear with the same sign pattern, creates the data for the weighted Ehrhart series, and interprets it as the Hilbert series of the monoid defined by the input data.

However, there is a restriction. Since the polynomial arithmetic has its own complexity and Normaliz must do it in GMP integers, it makes no sense to apply symmetrization if the dimension does not drop by a reasonable amount. Therefore we require that

$$\dim C' \le \frac{2}{3}\dim C).$$

If called with the option `-q`, Normaliz will try symmetrization, and also with `-v`, provoded the multiplicity has not already been computed by the descent algorithm (see Section 6.6). If the inequality for $\dim C'$ is not satisfied, it will simply compute the Hilbert series or the multiplicity without symmetrization. (In default mode it of course tries symmetrization for the Hilbert series.)

Whenever Normaliz has used symmetrization, it writes the file `<project>.symm.out` that contains the data of the symmetrized object. In it you find the multiplicity of `<project>.out` as virtual multiplicity and the Hilbert series as weighted Ehrhart series.

If you use the option `Symmetrize`, then the behavior depends on the other options:

(1) If neither the `HilbertSeries` nor `Multiplicity` is to be computed, Normaliz writes only the output file `<project>.symm.out` computed with `SupportHyperplanes`.
(2) If one of these goals is to be computed, Normaliz will do the symmetrization, regardless of the dimension inequality above (and often this makes sense).

By doing step (1) first, the user gets useful information of what to expect by symmetrization. In a second run, one can add `HilbertSeries` or `Multiplicity` if (1) was satisfactory.

The Condorcet example is too small in order to demonstrate the power of symmetrization. A suitable example is `PluralityVsCutoff.in`:

```
winfried@ubuntu:~/Dropbox/git_normaliz/source$ time ./normaliz -c ../example/PluralityVsCutoff
                                                \.....|
                    Normaliz 3.3.0               \....|
                                                  \...|
    (C) The Normaliz Team, University of Osnabrueck   \..|
                     March  2017                       \.|
```

```
                                                            \|
************************************************************
Command line: -c ../example/PluralityVsCutoff
Compute: DefaultMode
Embedding dimension of symmetrized cone = 6
...
------------------------------------------------------------
transforming data... done.

real        0m2.655s
user        0m5.328s
sys        0m0.080s
```

The Hilbert series is computable without symmetrization, but you better make sure that there is no power failure for the next week if you try that. (The time above includes the Hilbert basis computed automatically in dual mode).

Another good example included in the distribution is `CondEffPlur.in`, but it takes some hours with symmetrization (instead of days without). For it, the dimension drops only from 24 to 13.

Symmetrization is a special type of computations with a polynomial weight, and therefore requires Normaliz to be built with CoCoALib.

6.9. Computations with a polynomial weight

For a graded monoid M, which arises as the intersection $M = C \cap L$ of a rational cone$C$ and a lattice L, Normaliz computes the volume of the rational polytope

$$P = \{x \in \mathbb{R}_+M : \deg x = 1\},$$

called the multiplicity of M (for the given grading), the Hilbert series of M, and the quasipolynomial representing the Hilbert function. This Hilbert series of M is also called the Ehrhart series of P (with respect to L), and for the generalization introduced in this section we speak of Ehrhart series and functions.

The computations of these data can be understood as integrals of the constant polynomial $f = 1$, namely with respect to the counting measure defined by L for the Ehrhart function, and with respect to the (suitably normed) Lebesgue measure for the volume. Normaliz generalizes these computations to arbitrary polynomials f in n variables with rational coefficients. (Mathematically, there is no need to restrict oneself to rational coefficients for f.)

More precisely, set

$$E(f,k) = \sum_{x \in M, \deg x = k} f(x),$$

and call $E(f,\_)$ the *weighted Ehrhart function* for f. (With $f = 1$ we simply count lattice

points.) The *weighted Ehrhart series* is the ordinary generating function

$$E_f(t) = \sum_{k=0}^{\infty} E(f,k)t^k.$$

It turns out that $E_f(t)$ is the power series expansion of a rational function at the origin, and can always be written in the form

$$E_f(t) = \frac{Q(t)}{(1-t^\ell)^{\operatorname{totdeg} f+\operatorname{rank} M}}, \qquad Q(t) \in \mathbb{Q}[t],\ \deg Q < \operatorname{totdeg} f + \operatorname{rank} M.$$

Here $\operatorname{totdeg} f$ is the total degree of the polynomial f, and ℓ is the least common multiple of the degrees of the extreme integral generators of M. See [11] for an elementary account, references and the algorithm used by Normaliz.

At present, weighted Ehrhart series can only be computed with homogeneous data. Note that `excluded_faces` is a homogeneous input type. For them the monoid M is replaced by the set

$$M' = C' \cap L$$

where $C' = C \setminus \mathscr{F}$ and $\mathscr{F}$ is the union of a set of faces (not necessarily facets) of C. What has been said above about the structure of the weighted Ehrhart series remains true. We discuss an example below.

It follows from the general theory of rational generating functions that there exists a quasipolynomial $q(k)$ with rational coefficients and of degree $\leq \operatorname{totdeg} f + \operatorname{rank} M - 1$ that evaluates to $E(f,k)$ for all $k \geq 0$.

Let $m = \operatorname{totdeg} f$ (we use this notation to distinguish the degree of the polynomial from the degree of lattice points) and f_m be the degree m homogeneous component of f. By letting k go to infinity and approximating f_m by a step function that is constant on the meshes of $\frac{1}{k}L$ (with respect to a fixed basis), one sees

$$q^{(j)}_{\operatorname{totdeg} f+\operatorname{rank} M-1} = \int_P f_m \, d\lambda$$

where $d\lambda$ is the Lebesgue measure that takes value 1 on a basic mesh of $L \cap \mathbb{R}M$ in the hyperplane of degree 1 elements in $\mathbb{R}M$. In particular, the *virtual leading coefficient* $q^{(j)}_{\operatorname{totdeg} f+\operatorname{rank} M-1}$ is constant and depends only on f_m. If the integral vanishes, the quasipolynomial q has smaller degree, and the true leading coefficient need not be constant. Following the terminology of commutative algebra and algebraic geometry, we call

$$(\operatorname{totdeg} f + \operatorname{rank} M - 1)! \cdot q_{\operatorname{totdeg} f+\operatorname{rank} M-1}$$

the *virtual multiplicity* of M and f. It is an integer if f_m has integral coefficients and P is a lattice polytope.

The input format of polynomials has been discussed in Section 3.1.8.

The terminal output contains a factorization of the polynomial as well as some computation results. From the terminal output you may also recognize that Normaliz first computes the

triangulation and the Stanley decomposition and then applies the algorithms for integrals and weighted Ehrhart series.

Remarks (1) Large computations with many parallel threads may require much memory due to the fact that very long polynomials must be stored. Another reason for large memory usage can be the precomputed triangulation or Stanley decomposition.

(2) You should think about the option `BottomDecomposition`. It will be applied to the symmetrized input. (Under suitable conditions it is applied automatically.)

(3) A priori it is not impossible that Normaliz replaces a given grading deg by deg $/g$ where g is the grading denominator. If you want to exclude this possibility, set `NoGradingDenom`.

6.9.1. A weighted Ehrhart series

We discuss the Condorcet paradox again (and the last time), now starting from the symmetrized form. The file `Condorcet.symm.in` from the directory `example` contains the following:

```
amb_space 8
inequalities 3
1 -1 1 1 1 -1 -1 -1
1 1 -1 1 -1 1 -1 -1
1 1 1 -1 -1 -1 1 -1
nonnegative
total_degree
polynomial
1/120*1/120*(x[1]+5)*(x[1]+4)*(x[1]+3)*(x[1]+2)*(x[1]+1)*(x[2]+1)*
(x[3]+1)*(x[4]+1)*(x[5]+1)*(x[6]+1)*(x[7]+1)*(x[8]+5)*(x[8]+4)*
(x[8]+3)*(x[8]+2)*(x[8]+1);
```

We have seen this polynomial in Section 6.8 above.

From the Normaliz directory we start the computation by

```
./normaliz -cE example/Condorcet.symm
```

We could have used `--WeightedEhrhartSeries` instead of `-E` or put `WeightedEhrhartSeries` into the input file.

The file `Condorcet.symm.out` we find the information on the weighted Ehrhart series:

```
Weighted Ehrhart series:
1 5 133 363 ... 481 15 6
Common denominator of coefficients: 1
Series denominator with 24 factors:
1: 1  2: 14  4: 9

degree of weighted Ehrhart series as rational function = -25
```

```
Weighted Ehrhart series with cyclotomic denominator:
...
```

The only piece of data that we haven't seen already is the common denominator of coefficients. But since the polynomial has rational coefficients, we cannot any longer expect that the polynomial in the numerator of the series has integral coefficients. We list them as integers, but must then divide them by the denominator (which is1 in thus case since the weighted Ehrhart series is a Hilbert series in disguise). As usual, the representation with a denominator of cyclotomic polynomials follows.

And we have the quasipolynomial as usual:

```
Weighted Ehrhart quasi-polynomial of period 4:
 0:   693959790182222163590774784000 20899225...000000 ... 56262656
 1:   203475031022335179700809216000 0  7092764...648000 ... 56262656
 2:   693308184929915219977568256000 20892455...168000 ... 56262656
 3:   203475031022335179700809216000 0  7092764...648000 ... 56262656
with common denominator: 693959790182222163590774784000
```

The left most column indicates the residue class modulo the period, and the numbers in line k are the coefficients of the k-th polynomial after division by the common denominator. The list starts with $q_0^{(k)}$ and ends with (the constant) $q_{23}^{(k)}$. The interpretation of the remaining data is obvious:

```
Degree of (quasi)polynomial: 23

Expected degree: 23

Virtual multiplicity: 1717/8192
Virtual multiplicity (float) = 0.209594726562
```

6.9.2. Virtual multiplicity

Instead of the option `-E` (or (`--WeightedEhrhartSeries`) we use `-L` or (`--VirtualMultiplicity`). Then we can extract the virtual multiplicity from the output file.

6.9.3. An integral

In their paper *Multiplicities of classical varieties* (Proc. Lond. Math. Soc. (3) 110 (2015), 1033–105) J. Jeffries, J. Montaï¿½o and M. Varbaro ask for the computation of the integral

$$\int\limits_{\substack{[0,1]^m \\ \sum x=t}} (x_1 \cdots x_m)^{n-m} \prod_{1\le i<j\le m} (x_j - x_i)^2 \mathrm{d}\mu$$

taken over the intersection of the unit cube in $\mathbb{R}^m$ and the hyperplane of constant coordinate sum t. It is supposed that $t \leq m \leq n$. We compute the integral for $t = 2$, $m = 4$ and $n = 6$.

The polytope is specified in the input file `j462.in` (partially typeset in 2 columns):

```
amb_space 5          -1 0 0 0 1
inequalities 8       0 -1 0 0 1
1 0 0 0 0            0 0 -1 0 1
0 1 0 0 0            0 0 0 -1 1
0 0 1 0 0            equations 1
0 0 0 1 0            -1 -1 -1 -1 2
grading
unit_vector 5
polynomial
(x[1]*x[2]*x[3]*x[4])^2*(x[1]-x[2])^2*(x[1]-x[3])^2*
(x[1]-x[4])^2*(x[2]-x[3])^2*(x[2]-x[4])^2*(x[3]-x[4])^2;
```

The 8 inequalities describe the unit cube in $\mathbb{R}^4$ by the inequalities $0 \leq z_i \leq 1$ and the equation gives the hyperplane $z_1 + \cdots + z_4 = 2$ (we must use homogenized coordinates!). (Normaliz would find the grading itself.)

From the Normaliz directory the computation is called by

```
./normaliz -cI example/j462
```

where `-I` could be replaced by `--Integral`.

It produces the output in `j462.out` containing

```
integral  = 27773/29515186701000
integral  (float) = 9.40973210888e-10
```

As pointed out above, Normaliz integrates with respect to the measure in which the basic lattice mesh has volume 1. (this is $r!$ times the lattice normalized measure, $r = \dim P$.) In the full.dimensional case that is just the standard Lebesgue measure. But in lower dimensional cases this often not the case, and therefore Normaliz also computes the integral with respect to this *Euclidean* measure:

```
integral (euclidean) = 1.88194642178e-09
```

Note that `Integral` automatically sets `NoGradingDenom` since the polytope must be fixed for integrals.

6.9.4. Restrictions in MS Windows

We have not succeeded in compiling Normaliz with CoCoALib under MS Windows. in previous versions of Normaliz, the computations with polynomial weights were done by the separate program NmzIntegrate, and NmzIntegrate can still be used (in all operating systems). One

must start the computation from NmzIntegrate (and not from Normaliz, as was also possible in previous versions).

Unfortunately 1.3 is the last version of NmzIntegrate that we could compile under MS Windows. This causes some restrictions in the use of NmzIntegrate:

(1) Due to a bug it is possible that a segmentation fault occurs if excluded faces are used.
(2) The option `OutputDir` is not available.

An excellent way out is to run Normaliz (and NmzIntegrate) in the Linux subsystem of Windows 10 or in the Docker container.

6.10. Expansion of the Hilbert or weighted Ehrhart series

Normaliz can compute the expansion of the Hilbert function or the weighted Ehrhart function up to a given degree. To this end its expands the series. For the Hilbert function there is a second possibility by lattice point computation.

6.10.1. Series expansion

This is best explained by `CondorcetExpansion.in`:

```
amb_space 24
inequalities 3
1 1 1 1 1 1      -1 -1 -1 -1 -1 -1   1 1 -1 -1 1 -1     1 1 -1 -1 1 -1
1 1 1 1 1 1       1 1 -1 -1 1 -1    -1 -1 -1 -1 -1 -1   1 1 1 -1 -1 -1
1 1 1 1 1 1       1 1 1 -1 -1 -1     1 1 1 -1 -1 -1    -1 -1 -1 -1 -1 -1
nonnegative
total_degree
HilbertSeries
expansion_degree 50
```

By `expansion_degree 50` we tell Nprmaliz to compute the coefficients from degree 0 to degree 50 in theexpansion of the Hilbertt series. So the output contains

```
Expansion of Hilbert series
0: 1
1: 6
2: 153
3: 586
4: 7143
5: 21450
...
49: 817397314032054600
50: 1357391110355875044
```

If the shift is nonzero, it is automatically added to the degree so that the expansion always starts at the shift.

The expansion degree applies to the weighted Ehrhart series as well if it is computed.

There is nothing more to say, except that (in principle) there is another method, as discussed in the next section.

6.10.2. Counting lattice points by degree

As an example we look at `CondorcetRange.in`:

```
amb_space 24
inequalities 3
1 1 1 1 1 1      -1 -1 -1 -1 -1 -1   1 1 -1 -1 1 -1     1 1 -1 -1 1 -1
1 1 1 1 1 1       1 1 -1 -1 1 -1    -1 -1 -1 -1 -1 -1   1 1 1 -1 -1 -1
1 1 1 1 1 1       1 1 1 -1 -1 -1     1 1 1 -1 -1 -1    -1 -1 -1 -1 -1 -1
nonnegative
total_degree
constraints 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 <= 5
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 >= 3
Projection
LatticePoints
HilbertSeries
expansion_degree 5
```

This input defines the polytope that is cit out from the cone (defined by the 3 inequalities) by the two inequalities that are defined as constraints (for clarity). These two inequalities mean that we want to compute the polytope of all points x in the cone satisfying the condition $3 \leq \deg x \leq 5$. We add `Projection` in conjunction with `LatticePoints` to keep Normaliz from choosing the primal algorithm, which would do the job as well, but much more slowly.

In the output we find

```
Hilbert series:
586 7143 21450
denominator with 0 factors:

shift = 3
```

Taking the shift into account, we see that there are 586 lattice points in degree 3, 7413 in degree 4 and 21450 in degree 5. But this becomes even more obvious by (the unnecessary) `expansion_degree 5`:

```
Expansion of Hilbert series
3: 586
4: 7143
5: 21450
```

The limitation of this method is the necessity to store the lattice points, and even if we had the possibility of counting them without storage, it would be difficult to get as far as with the

series expansion. So this method should be considered as a way out in cases where the Hilbert series is not computable (or takes too much time) and the expected numbers are not too large.

6.11. Significant coefficients of the quasipolynomial

If the degree and simultaneously the period of the Hilbert or weighted Ehrhart quasipolynomial are large, the space needed to store it (usually with large coefficients) may exceed the available memory. Depending on the application, only a certain number of the coefficients may be significant. Therefore one can limit the number of highest coefficients that are stored and printed. We look at the input file `CondorcetN.in`:

```
amb_space 24
inequalities 3
1 1 1 1 1 1     -1 -1 -1 -1 -1 -1     1  1 -1 -1  1 -1     1  1 -1 -1  1 -1
1 1 1 1 1 1      1  1 -1 -1  1 -1    -1 -1 -1 -1 -1 -1     1  1  1 -1 -1 -1
1 1 1 1 1 1      1  1  1 -1 -1 -1     1  1  1 -1 -1 -1    -1 -1 -1 -1 -1 -1
nonnegative
total_degree
nr_coeff_quasipol 2
```

The output file shows the following information on the quasipolynomial:

```
Hilbert quasi-polynomial of period 4:
only 2 highest coefficients computed
their common period is 2
0:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 15982652919 56262656
1:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 15528493056 56262656
with common denominator = 693959790182222163590774784000
```

Normaliz computes and prints only as many components of the quasipolynomial as required by the common period of the printed coefficients. Coefficients outside the requested range are printed as 0.

The bound on the significant coefficients applies simultaneously to the Hilbert polynomial and the weighted Ehrhart quasipolynomial—usually one is interested in only one of them.

By default Normaliz computes the quasipolynomial only if the period does not exceed a preset bound, presently 10^6. If this bound is too small for your computation, you can remove it by the option

```
NoPeriodBound
```

6.12. Explicit dehomogenization

Inhomogeneous input for data in $\mathbb{R}^d$ is homogenized by an extra $(d+1)$th coordinate. The dehomogenization sets the last coordinate equal to 1. Other systems may prefer the first coordinate. By choosing an explicit dehomogenization Normaliz can be adapted to such input.

The file dehomogenization.in

```
amb_space 3
inequalities 2
-1 1 0
-1 0 1
dehomogenization
unit_vector 1
```

indicates that in this case the first variable is the homogenizing one. The output file

```
1 module generators
2 Hilbert basis elements of recession monoid
1 vertices of polyhedron
2 extreme rays of recession cone
3 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 2

size of triangulation   = 0
resulting sum of |det|s = 0

dehomogenization:
1 0 0


module rank = 1

************************************************************************

1 module generators:
1 1 1

2 Hilbert basis elements of recession monoid:
0 0 1
0 1 0

1 vertices of polyhedron:              3 support hyperplanes of polyhedron (homogenized)
1 1 1                                  -1 0 1
                                       -1 1 0
2 extreme rays of recession cone:       1 0 0
0 0 1
0 1 0
```

shows that Normaliz does the computation in the same way as with implicit dehomogenization, except that now the first coordinate decides what is in the polyhedron and what belongs

to the recession cone, roughly speaking.

Note that the dehomogenization need not be a coordinate. It can be any linear form that is nonnegative on the cone generators.

6.13. Projection of cones and polyhedra

Normaliz can not only compute projections (as has become visible in the discussion of project-and-float), but also export them if asked for by the computation goal

`ProjectCone`

As the computation goal says, only the cone is projected. Lattice data are not taken care of. The image of the projection is computed with the goals `SupportHyperplanes` and `ExtremeRays`, and the result is contained in an extra output file `<project>.ProjectCone.out`, similarly to the result of the integer hull computation. (All other computation goals are applied to the input cone.)

The image and the kernel a of the projection are complementary vector subspaces generated by unit vectors. Those spanning the image are selected by the entries 1 in the 0-1 vector `projection_coordinates` of the input file. As an example we take `small_proj.in`:

```
amb_space 6
cone 190
6 0 7 0 10 1
...
0 0 0 16 7 1
projection_coordinates
1 1 0 1 0  1
ProjectCone
```

As you can see from `small_proj.out`, almost nothing is computed dor the input cone itself. (However, any further computation goal would change this.) The result of the projection is contained in `small_proj.ProjectCone.out`:

```
14 extreme rays
9 support hyperplanes

embedding dimension = 4
...
14 extreme rays:
0  0  1 1
0  0 17 1
 ...
 11  0  5 1
 11  0  6 1

 9 support hyperplanes:
 -1 -1 -1 20
```

```
...
 1  0  1 -1
```

In this example, the input is of type "generators". Normaliz simply projects them and uses the images as input of type cone for the projection. If the input is of type "constraints", as in small_sh_proj.in,

```
amb_space 6
inequalities 32
 0 -2 -2  0 -1 24
-1 -1 -1 -1 -1 23
...
 -2 -1  0  1 -1 28
  0  1  1 -1  0 16
 projection_coordinates
 1 1 0 1 0  1
 ProjectCone
```

then Normaliz uses the projection part of the project-and-lift algorithm to find the result, as you can see from the terminal output. Equivalent inhomogeneous input files are small_proj_inhom.in and small_sh_proj_inhom.in.

Polyhedra and polytopes are treated by Normaliz as intersections of cones and hyperplanes. The hyperplane is given by the grading in the homogeneous case and by the dehomogenization in the inhomogeneous case. For the projection of the polyhedron, the kernel of the projection must be parallel to this hyperplane. Normaliz checks this condition (automatically satisfied for inhomogeneous input) and transfers the grading or the dehomogenization, respectively, to the image. Therefore the image of the input polyhedron is indeed the polyhedron defined by the projection.

6.14. Nonpointed cones

Nonpointed cones and nonpositive monoids contain nontrivial invertible elements. The main effect is that certain data are no longer unique, or may even require a new definition. An important point to note is that cones always split off their unit groups as direct summands and the same holds for normal affine monoids. Since Normaliz computes only normal affine monoids, we can always pass to the quotient by the unit groups. Roughly speaking, all data are computed for the pointed quotient and then lifted back to the original cone and monoid. It is inevitable that some data are no longer uniquely determined, but are unique only modulo the unit group, for example the Hilbert basis and the extreme rays. Also the multiplicity and the Hilbert series are computed for the pointed quotient. From the algebraic viewpoint this means to replace the field K of coefficients by the group ring L of the unit group, which is a Laurent polynomial ring over K: instead of K-vector space dimensions one considers ranks over L.

6.14.1. A nonpointed cone

As a very simple example we consider the right halfplane (`halfspace2.in`):

```
amb_space 2
inequalities 1
1 0
```

When run in default mode, it yields the following output:

```
1 Hilbert basis elements
1lattice points in polytope (Hilbert basis elements of degree 1)
1 extreme rays
1 support hyperplanes

embedding dimension = 2
rank = 2 (maximal)
external index = 1
dimension of maximal subspace = 1

size of triangulation   = 1
resulting sum of |det|s = 1

grading:
1 0

degrees of extreme rays:
1: 1

Hilbert basis elements are of degree 1

multiplicity = 1

Hilbert series:
1
denominator with 1 factors:
1: 1

degree of Hilbert Series as rational function = -1

Hilbert polynomial:
1
with common denominator = 1

rank of class group = 0
class group is free

***********************************************************************
```

```
1 lattice points in polytope (Hilbert basis elements of degree 1):
1 0

0 further Hilbert basis elements of higher degree:

1 extreme rays:
1 0

1 basis elements of maximal subspace:
0 1

1 support hyperplanes:
1 0
```

In the preamble we learn that the cone contains a nontrivial subspace. In this case it is the vertical axis, and close to the end we see a basis of this subspace, namely $(0,1)$. This basis is always simultaneously a $\mathbb{Z}$-basis of the unit group of the monoid. The rest of the output is what we have gotten for the positive horizontal axis which in this case is a natural representative of the quotient modulo the maximal subspace, The quotient can always be embedded in the cone or monoid respectively, but there is no canonical choice. We could have gotten $(1,5)$ as the Hilbert basis as well.

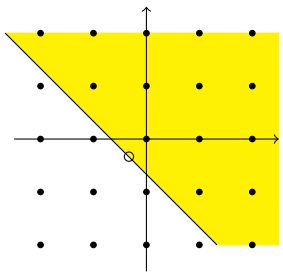
Normaliz has found a grading. Of course it vanishes on the unit group, but is positive on the quotient monoid modulo the unit group.

Note that the data of type "dimension" (embedding dimension, rank, rank of recession monoid in the inhomogeneous case, affine dimension of the polyhedron)) are measured before the passage to the quotient modulo the maximal subspace. The same is true for equations and congruences (which are trivial for the example above).

6.14.2. A polyhedron without vertices

We define the affine halfspace of the figure by `gen_inhom_nonpointed.in`:

```
amb_space 2
cone 3
1 -1
-1 1
0 1
vertices 1
-1 -1 3
```



It is clear that the “vertex” is not a vertex in the strict sense, bt only gives a displacement of the cone. The output when run in default mode:

```
1 module generators
1 Hilbert basis elements of recession monoid
1 vertices of polyhedron
1 extreme rays of recession cone
2 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 2
internal index = 3
dimension of maximal subspace = 1

size of triangulation   = 1
resulting sum of |det|s = 3

dehomogenization:
0 0 1


module rank = 1

************************************************************************

1 module generators:
0 0 1

1 Hilbert basis elements of recession monoid:
0 1 0

1 vertices of polyhedron:
0 -2 3

1 extreme rays of recession cone:
0 1 0
```

```
1 basis elements of maximal subspace:
1 -1 0

2 support hyperplanes of polyhedron (homogenized):
0 0 1
3 3 2
```

The "vertex" of the polyhedron shown is of course the lifted version of the vertex modulo the maximal subspace. It is not the input "vertex", but agrees with it up to a unit.

6.14.3. Checking pointedness first

Nonpointed cones will be an exception in Normaliz computations, and therefore Normaliz assumes that the (recession) cone it must compute is pointed. Only in rare circumstances it could be advisable to have this property checked first. There is no need to do so when the dual algorithm is used since it does not require the cone to be pointed. Moreover, if an explicit grading is given or a grading dependent computation is asked for, one cannot save time by checking the pointedness first.

The exceptional case is a computation, say of a Hilbert basis, by the primal algorithm in which the computation of the support hyperplanes needs very long time to be completed. If you are afraid this may happen, you can force Normaliz to compute the support hyperplanes right away by adding `IsPointed` to the computation goals. This is a disadvantage only if the cone is unexpectedly pointed.

6.14.4. Input of a subspace

If a linear subspace contained in the cone is known a priori, it can be given to Normaliz via the input type `subspace`. If Normaliz detects a `subspace`, it appends the rows of the matrix to the generators of the cone, and additionally the negative of the sum of the rows (since we must add the subspace as a cone). If `subspace` is combined with `cone_and_lattice`, then the rows of `subspace` are also appended to the generators of the lattice. It is not assumed that the vectors in `subspace` are linearly independent or generate the maximal linear subspace of the cone. A simple example (`subspace4.in`):

```
amb_space 4
cone 4
1 0 2 0
0 1 -2 1
0 0 0 1
0 0 0 -1
subspace 1
0 0 1 0
```

From the output:

```
2 lattice points in polytope (Hilbert basis elements of degree 1):
 0 1 0 0
 1 0 0 0

0 further Hilbert basis elements of higher degree:

2 extreme rays:
 0 1 0 0
 1 0 0 0

2 basis elements of maximal subspace:
 0 0 1 0
 0 0 0 1

2 support hyperplanes:
 0 1 0 0
 1 0 0 0
```

One should note that the maximal subspace is generated by the smallest face that contains all invertible elements. Therefore, in order to make all vectors in a face invertible, it is enough to put a single vector from the interior of the face into `subspace`.

6.14.5. Data relative to the original monoid

If original monoid generators are defined, there are two data related to them that must be read with care.

First of all, we consider the original monoid generators as being built from the vectors in `cone` or `cone_and_lattice` plus the vectors in `subspace` and additionally the negative of the sum of the latter (as pointed out above).

The test for "Original monoid is integrally closed' is correct – it returns `true` if and only if the original monoid as just defined indeed equals the computed integral closure. (There was a mistake in version 3.0.)

The "module generators over the original monoid" only refer to the *image* of the original monoid and the image of the integral closure *modulo the maximal subspace*. They do not take into account that the unit group of the integral closure may not be generated by the original generators. An example in which the lack of integral closedness is located in the unit group (`normface.in`):

```
amb_space 5
cone 4
0 0 0 1 1
1 0 0 1 1
0 1 0 1 1
0 0 1 1 1
```

```
subspace 4
0 0 0 0 1
1 0 0 0 1
0 1 0 0 1
1 1 2 0 1
```

From the output file:

```
...
dimension of maximal subspace = 4
original monoid is not integrally closed
unit group index = 2
...

1 lattice points in polytope (Hilbert basis elements of degree 1):
0 0 0 1 0
...
1 module generators over original monoid:
0 0 0 0 0
```

The original monoid is not integrally closed since the unit group of the integral closure is strictly larger than that of the original monoid: the extension has index 2, as indicated. The quotients modulo the unit groups are equal, as can be seen from the generator over the original monoid or the Hilbert basis (of the integral closure) that is contained in the original monoid.

6.15. Exporting the triangulation

The option `-T` asks Normaliz to export the triangulation by writing the files `<project>.tgn` and `<project>.tri`:

`tgn` The file `tgn` contains a matrix of vectors (in the coordinates of $\mathbb{A}$) spanning the simplicial cones in the triangulation.

`tri` The file `tri` lists the simplicial subcones. There are two variants, depending on whether `ConeDecomposition` had been set. Here we assume that `ConeDecomposition` is not computed. See Section 6.15.2 for the variant with `ConeDecomposition`.
The first line contains the number of simplicial cones in the triangulation, and the next line contains the number $m+1$ where $m = \operatorname{rank}\mathbb{E}$. Each of the following lines specifies a simplicial cone Δ: the first m numbers are the indices (with respect to the order in the file `tgn`) of those generators that span Δ, and the last entry is the multiplicity of Δ in $\mathbb{E}$, i. e. the absolute value of the determinant of the matrix of the spanning vectors (as elements of $\mathbb{E}$).

The following example is the 2-dimensional cross polytope with one excluded face (`cross2.in`). The excluded face is irrelevant for the triangulation.

```
amb_space 3
polytope 4
```

```
1  0
0  1
-1  0
0 -1
excluded_faces 1
1  1 -1
```

Its `tgn` and `tri` files are

```
tgn             tri
4               2
3               4
1 0 1           1 2 3    2
0 1 1           1 3 4    2
-1 0 1          plain
0 -1 1
```

We see the 4 vertices $v_1, \ldots, v_4$ in homogenized coordinates in `tgn` and the 2 simplices (or the simplicial cones over them) in `tri`: both have multiplicity 2. The last word `plain` indicates that Normaliz has computed a triangulation in the strict sense, namely a simplicial subdivision in which neighboring simplicial cones match along common faces. The alternative is `nested` that we discuss below.

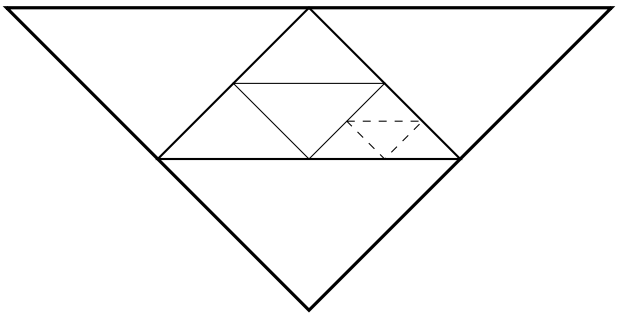
In addition to the files `<project>.tgn` and `<project>.tri`, also the file `<object>.inv` is written. It contains the data of the file `<project>.out` above the line of stars in a human and machine readable format.

6.15.1. Nested triangulations

If Normaliz has subdivided a simplicial cone of a triangulation of the cone C, the resulting decomposition of C may no longer be a triangulation in the strict sense. It is rather a *nested triangulation*, namely a map from a rooted tree to the set of full-dimensional subcones of C with the following properties:

(1) the root is mapped to C,
(2) every other node is mapped to a full dimensional simplicial subcone,
(3) the simplicial subcones corresponding to the branches at a node x form a triangulation of the simplicial cone corresponding to x.

The following figure shows a nested triangulation:



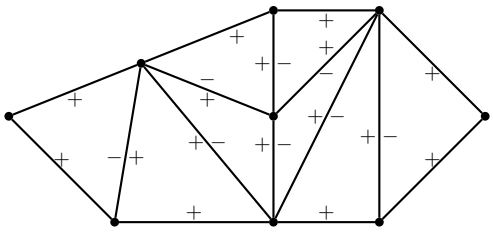
For the Normaliz computations, nested triangulations are as good as ordinary triangulations, but in other applications the difference may matter. With the option `-T`, Normaliz prints the leaves of the nested triangulation to the `tri` file. They constitute the simplicial cones that are finally evaluated by Normaliz.

The triangulation is always plain if `-T` is the only computation goal or if it is just combined with `-v`. Otherwise it can only fail to be plain if it contains determinants $\geq 10^8$.

The subdivision can be blocked by `NoSubdivision`, independently of the computation goals.

6.15.2. Disjoint decomposition

Normaliz can export the disjoint decomposition of the cone that it has computed. This decomposition is always computed together with a full triangulation, unless only the multiplicity is asked for. It represents the cone as the disjoint union of semiopen simplicial subcones. The corresponding closed cones constitute the triangulation, and from each of them some facets are removed so that one obtains a disjoint decomposition. See [8] for more information. In the following figure, the facets separating the triangles are omitted in the triangle on the $-$ side.



If you want to access the disjoint decomposition, you must activate the computation goal `ConeDecomposition` or use the command line option is `-D`. As an example we compute `cross2.in` with the computation goal `ConeDecomposition`. The file `cross2.tri` now looks as follows:

```
2
7
1 2 3    2    0 0 0
2 3 4    2    0 0 1
plain
```

As before the first line contains the size of the triangulation and the second is the number of entries of each row. The first 3 entries in each line are the indices of the extreme rays with respect to the `tgn` file and the fourth entry is the determinant. They are followed by a $0/1$ vector indicating the open facets in the order in which they are opposite to the extreme rays. If the corresponding entry is 1, the facet must be removed.

In our example all facets of the first simplicial cone are kept, and from the second simplicial cone the facet opposite to the third extreme ray (with index 4 relative to `tgn`) must be removed.

The disjoint decomposition which is the basis of all Hilbert series computations uses the algorithm suggested by Kï¿½ppe and Verdoolaege [16].

6.16. Exporting the Stanley decomposition

The option `-y` makes Normaliz write the files `<project>.tgn`, `<project>.dec` and `<project>.inv`. Stanley decomposition is contained in the file with the suffix `dec`. But this file also contains the inclusion/exclusion data if there are excluded faces:

(a) If there are any excluded faces, the file starts with the word `in_ex_data`. The next line contains the number of such data that follow. Each of these lines contains the data of a face and the coefficient with which the face is to be counted: the first number lists the number of generators that are contained in the face, followed by the indices of the generators relative to the `tgn` file and the last number is the coefficient.

(b) The second block (the first if there are no excluded faces) starts with the word `Stanley_dec`, followed by the number of simplicial cones in the triangulation.

For each simplicial cone Δ in the triangulation this file contains a block of data:

(i) a line listing the indices $i_1, \dots, i_m$ of the generators $v_{i_1}, \dots, v_{i_m}$ relative to the order in `tgn` (as in `tri`, $m = \operatorname{rank} \mathbb{E}$);

(ii) a $\mu \times m$ matrix where μ is the multiplicity of Δ (see above).

In the notation of [8], each line lists an "offset" $x + \varepsilon(x)$ by its coordinates with respect to $v_{i_1}, \dots, v_{i_m}$ as follows: if $(a_1, \dots, a_m)$ is the line of the matrix, then

$$x + \varepsilon(x) = \frac{1}{\mu}(a_1 v_{i_1} + \cdots + a_m v_{i_m}).$$

The `dec` file of the example above is

```
in_ex_data
1
2 1 2 -1
Stanley_dec
2
1 3 4              1 2 3
2                  2
3                  3
```

```
0 0 2                0 0 0
1 1 2                1 0 1
```

There is 1 face in `in_ex_data` (namely the excluded one), it contains the 2 generators v_1 and v_2 and appears with multiplicity -1. The Stanley decomposition consists of 4 components of which each of the simplicial cone contains 2. The second offset in the second simplicial cone is

$$\frac{1}{2}(1v_1+0v_2+1v_3)=(0,0,1).$$

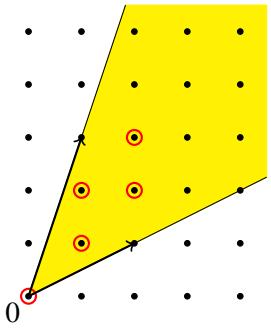
We recommend you to process the file `3x3magiceven.in` with the option `-ahTy` activated. Then inspect all the output files in the subdirectory `example` of the distribution.

6.17. Module generators over the original monoid

Suppose that the original generators are well defined in the input. This is always the case when these consists just of a `cone` or a `cone_and_lattice`. Let M be the monoid generated by them. Then Normaliz computes the integral closure N of M in the effective lattice $\mathbb{E}$. It is often interesting to understand the difference set $N \setminus M$. After the introduction of a field K of coefficients, this amounts to understanding $K[N]$ as a $K[M]$-module. With the option `ModuleGeneratorsOverOriginalMonoid, -M` Normaliz computes a minimal generating set T of this module. Combinatorially this means that we find an irreducible cover

$$N=\bigcup_{x\in T} x+M.$$

Note that $0 \in T$ since $M \subset N$.



As an example, we can run `2cone.in` with the option `-M` on the command line. This yields the output

```
...
4 Hilbert basis elements:
 1 1
 1 2                  5 module generators over original monoid:
 1 3                   0 0
 2 1                   1 1
```

```
                        1 2
2 extreme rays:         2 2
 1 3                    2 3
 2 1
```

In the nonpointed case Normaliz can only compute the module generators of N/N_0 over $M/(M \cap N_0)$ where N_0 is the unit group of N. If $M_0 \neq M_0$, this is not a system of generators of M over N.

6.17.1. An inhomogeneous example

Let us have a look at a very simple input file (`genmod_inhom2.in`):

```
amb_space 2
cone 2
0 3
2 0
vertices 1
0 0 1
ModuleGeneratorsOverOriginalMonoid
```

The cone is the positive orthant that we have turned into a polyhedron by adding the vertex $(0,0)$. The original monoid is generated by $(2,0)$ and $(0,3)$.

In addition to the original monoid M and its integral closure N we have a third object, namely the module P of lattice points in the polyhedron.We compute

1. the system of generators of P over N (the `module generators`) and
2. the system of generators of P over N (the `module generators over original monoid`).

We do not compute the system of generators of N over M (that we get in the homogeneous case).

The output:

```
1 module generators
2 Hilbert basis elements of recession monoid
1 vertices of polyhedron
2 extreme rays of recession cone
6 module generators over original monoid
3 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 2
internal index = 6

size of triangulation   = 1
resulting sum of |det|s = 6
```

```
dehomogenization:
0 0 1


module rank = 1

**************************************************************************

1 module generators:
 0 0 1

2 Hilbert basis elements of recession monoid:
 0 1 0
 1 0 0

1 vertices of polyhedron:
 0 0 1

2 extreme rays of recession cone:
 0 1 0
 1 0 0

6 module generators over original monoid:
 0 0 1
 0 1 1
 0 2 1
 1 0 1
 1 1 1
 1 2 1

3 support hyperplanes of polyhedron (homogenized):
 0 0 1
 0 1 0
 1 0 0
```

6.18. Lattice points in the fundamental parallelepiped

Let $u_1,\dots,u_n$ be linearly independent vectors in $\mathbb{Z}^d \subset \mathbb{R}^d$. They span a simplicial cone C. Moreover let U be the subgroup of $(\mathbb{R}^d,+)$ generated by $u_1,\dots,u_n$ and let $v \in \mathbb{R}^d$. We are interested in the shifted cone $C' = v+C$. We assume that C' contains a lattice point. This need not be true if $n < s$, but with our assumption we can also assume that $n = d$ after the restriction to the affine space spanned by C'. The *fundamental* parallelepiped of C (with respect to U) is

$$F = \mathrm{par}(u_1,\dots,u_d) = \{q_q u_1 + \cdots + q_d u_d : 0 \le q_i < 1\}.$$

Set $F' = v + F$. Then the translates $u + F'$, $u \in U$, tile $\mathbb{R}^d$; so F' is a fundamental domain for the action of U on $\mathbb{R}^d$ by translation, and we call it F' the *fundamental* parallelepiped of C' (with respect to U). Every point in $\mathbb{R}^d$ differs from exactly one point in F' by an element of U. This holds in particular for the lattice points.

One of the main basic tasks if Normaliz is the computation of the lattice points in F', especially in the case $v = 0$ (but not only). Looking back at the examples in Section 6.17, we see that we can in fact compute and export these lattice points via the computation goal `ModuleGeneratorsOverOriginalMonoid`.

Often however, an additional complication comes up: we must shift F' by an infinitesimally small vector in order to exclude certain facets of C'. This would be difficult in Normaliz without the input type `open_facets` (see Section 3.12). Recall that this is a 0-1-vector whose entries 1 indicate which facets must be avoided: if its i-th entry is 1, then the facet opposite to $v + u_i$ must be made 'open".

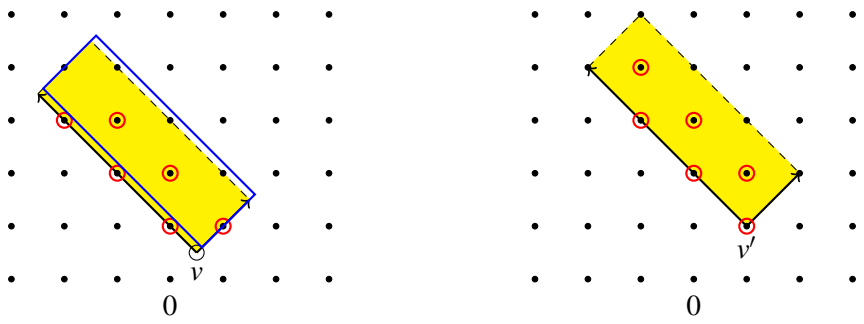
The input file `no_open_facets.in` is

```
amb_space 2
cone 2
1 1
-3 3
vertices 1
1/2 1/2 1
ModuleGeneratorsOverOriginalMonoid
```

Then `no_open_facets.out` contains

```
6 module generators over original monoid:
-2 3 1
-1 2 1
-1 3 1
 0 1 1
 0 2 1
 1 1 1
```

These are 6 encircled points in the left figure.



Now we add

```
open_facets
1 0
```

to the input (to get `open_facets.in`). We have tried to indicate the infinitesimal shift by the blue rectangle in the left figure. The computation yields

```
6 module generators over original monoid:
-1 3 1
-1 4 1
0 2 1
0 3 1
1 1 1
1 2 1
```

which are the encircled lattice points in the right figure. It is explained in Section 3.12 how the new vector v' is computed.

Note that the lattice points are listed with the homogenizing coordinate 1. In fact, both `vertices` and `open_facets` make the computation inhomogeneous. If both are missing, then the lattice points are listed without the homogenizing coordinate. If you want a uniform format for the output, you can use the zero vector for `open_facets` or the origin as the vertex. Both options change the result only to the extent that the homogenizing coordinate is added.

6.19. Precomputed data

The input of precomputed data can be useful if their computation takes long and they can be used again in subsequent computations. Normaliz takes their correctness for granted since there is no way of checking it without recomputation.

6.19.1. Support hyperplanes

They are formally introduced in Section 3.3.1. The file `2cone_supp.in` is just a toy example:

```
amb_space 2
cone 2
2 1
1 3
support_hyperplanes 2
-1  2
 3 -1
```

Note: (i) the input type `support_hyperplanes` is only allowed together with homogeneous input.

(ii) They are not used in the definition of the cone. However, once the cone is defined (by generators or constraints), they override any other inequalities that are implicitly or explicitly

given in the input. `excluded_faces` are an exception; they remain excluded.

6.19.2. Extreme rays

They are formally introduced in Section 3.2.1. Again we give only a toy example in `2cone_ext.in`:

```
amb_space 2
inequalities 2
-1  2
3 -1
extreme_rays 2
2 1
1 3
```

Note: (i) the input type `extreme_rays` is only allowed together with homogeneous input.

(ii) They are not used in the definition of the cone. However, once the cone is defined (by generators or constraints), they override any other cone generators. They are not considered to be generators of the original monoid.

6.19.3. Hilbert basis of the recession cone

In applications one may want to compute several polyheda with the same recession cone. In these cases it is useful to add the Hilbert basis of the recession cone to the input. An example is `small_inhom_hbrc.in`:

```
amb_space 6
cone 190
6 0 7 0 10 1
...
vertices 4
0 0 0 0 0 0 1
1 2 3 4 5 6 2
-1 3 9 8 7 1 3
0 2 4 5 8 10 7
hilbert_basis_rec_cone 34591
0  0  0  1  6 1 0
0  0  0  1  7 1 0
...
```

As in the other cases with precomputed data, Normaliz must believe you and the precomputed Hilbert basis of the recession cone does not define the latter.

It requires inhomogeneous input. Note that it can only be used in the primal algorithm. In the dual algorithm it is useless and therefore ignored.

6.20. Shift, denominator, quasipolynomial and multiplicity

In this section we discuss the interplay of shift, denominator of the grading and the quasipolynomial. As long as the denominator is 1, the situation is very simple and no ambiguity arises. See Section 2.9. We modify the example from that section as follows (`InhomIneq_7.in`):

```
amb_space 2
inhom_inequalities 3
0  2  1
0 -2  3
2 -2  3
grading
7 0
```

The output related to the grading is

```
grading:
7 0 0
with denominator = 7


module rank = 2
multiplicity = 2

Hilbert series:
1 1
denominator with 1 factors:
1: 1

shift = -1

degree of Hilbert Series as rational function = -1

Hilbert polynomial:
2
with common denominator = 1
```

The Hilbert series computed by hand is

$$\frac{t^{-7}+1}{1-t^7}.$$

We obtain it from the output as follows. The printed series is

$$\frac{1+t}{1-t}.$$

Now the shift is applied and yields

$$\frac{t^{-1}+1}{1-t}.$$

Finally we make the substitution $t \mapsto t^7$, and obtain the desired result.

Now we add the complication $x_1 + x_2 \equiv -1 \mod 8$ (`InhomIneq_7_8.in`):

```
amb_space 2
inhom_inequalities 3
0  2  1
0 -2  3
2 -2  3
grading
7 0
inhom_congruences 1
1 1 1 8
```

The result:

```
grading:
7 0 0
with denominator = 7


module rank = 2
multiplicity = 2

Hilbert series:
1 0 0 0 0 0 0 1
denominator with 1 factors:
8: 1

shift = -1

degree of Hilbert Series as rational function = -2

The numerator of the Hilbert series is symmetric.

Hilbert series with cyclotomic denominator:
-1 1 -1 1 -1 1 -1
cyclotomic denominator:
1: 1  4: 1  8: 1

Hilbert quasi-polynomial of period 8:
0:  0           4:  0
1:  0           5:  0
2:  0           6:  1
3:  0           7:  1
with common denominator = 1
```

The printed Hilbert series is

$$\frac{1+t^7}{1-t^8}.$$

The application of the shift yields

$$\frac{t^{-1}+t^6}{1-t^8}.$$

the correct result for the divided grading. *The Hilbert quasipolynomial is computed for the divided grading*, as already explained in Section 2.6.2. As a last step, we can apply the substitution $t \mapsto t^7$ in order obtain the Hilbert series

$$\frac{t^{-7}+t^{42}}{1-t^{56}}$$

for the original grading.

Like the quasipolynomial, *the multiplicity is computed for the divided grading*. The definition of multiplicity is discussed in Section A.7, and we leave it to the reader to interpret the values above in its light.

7. Optional output files

Note that the options `-T, Triangulation` and `-y, StanleyDec` also write files in addition to `<project>.out`. Also symmetrization and `IntegerHull` produce extra output files for the derived cones. But these are *not* optional.

When one of the options `Files,-f` or `AllFiles, -a` is activated, Normaliz writes additional optional output files whose names are of type `<project>.<type>`. Moreover one can select the optional output files individually on the command line. Most of these files contain matrices in a simple format:

```
<m>
<n>
<x_1>
...
<x_m>
```

where each row has `<n>` entries. Exceptions are the files with suffixes `cst`, `inv`, `esp`.

Note that the files are only written if they would contain at least one row.

As pointed out in Section 5.5, the optional output files for the integer hull are the same as for the original computation, as far as their content has been computed.

7.1. The homogeneous case

The option `-f` makes Normaliz write the following files:

gen contains the Hilbert basis. If you want to use this file as an input file and reproduce the computation results, then you must make it a matrix of type `cone_and_lattice` (and add the dehomogenization in the inhomogeneous case).

cst contains the constraints defining the cone and the lattice in the same format as they would appear in the input: matrices of types *constraints* following each other. Each matrix is concluded by the type of the constraints. Empty matrices are indicated by 0 as the number of rows. Therefore there will always be at least 3 matrices.

If a grading is defined, it will be appended. Therefore this file (with suffix `in`) as input for Normaliz will reproduce the Hilbert basis and all the other data computed, at least in principle.

inv contains all the information from the file `out` that is not contained in any of the other files.

If `-a` is activated, then the following files are written *additionally:*

ext contains the extreme rays of the cone.

ht1 contains the degree 1 elements of the Hilbert basis if a grading is defined.

egn,esp These contain the Hilbert basis and support hyperplanes in the coordinates with respect to a basis of $\mathbb{E}$. `esp` contains the grading and the dehomogenization in the coordinates of $\mathbb{E}$. Note that no equations for $\mathbb{C} \cap \mathbb{E}$ or congruences for $\mathbb{E}$ are necessary.

lat contains the basis of the lattice $\mathbb{E}$.

mod contains the module generators of the integral closure modulo the original monoid.

msp contains the basis of the maximal subspace.

In order to select one or more of these files individually, add an option of type `--<suffix>` to the command line where `<suffix>` can take the values

```
gen, cst, inv, ext, ht1, egn, esp, lat, mod, msp, typ
```

The type `typ` is not contained in `Files` or `AllFiles` since it can be extremely large. It is of the matrix format described above. It is the product of the matrices corresponding to `egn` and the transpose of `esp`. In other words, the linear forms representing the support hyperplanes of the cone C are evaluated on the Hilbert basis. The resulting matrix, with the generators corresponding to the rows and the support hyperplanes corresponding to the columns, is written to this file.

The suffix `typ` is motivated by the fact that the matrix in this file depends only on the isomorphism type of monoid generated by the Hilbert basis (up to row and column permutations). In the language of [5] it contains the *standard embedding*.

Note: the explicit choice of an optional output file does *not* imply a computation goal. Output files that would contain unknown data are simply not written without a warning or error message.

7.2. Modifications in the inhomogeneous case

The optional output files are a subset of those that can be produced in the homogeneous case. The main difference is that the generators of the solution module and the Hilbert basis of the recession monoid appear together in the file `gen`. They can be distinguished by evaluating the dehomogenization on them (simply the last component with inhomogeneous input), and the same applies to the vertices of the polyhedron and extreme rays of the recession cone. The file `cst` contains the constraints defining the polyhedron and the recession module in conjunction with the dehomogenization, which is also contained in the `cst` file, following the constraints.

With `-a` the files `egn` and `esp` are produced. These files contain `gen` and the support hyperplanes of the homogenized cone in the coordinates of $\mathbb{E}$, as well as the dehomogenization.

8. Performance

8.1. Parallelization

The executables of Normaliz have been compiled for parallelization on shared memory systems with OpenMP. Parallelization reduces the "real" time of the computations considerably, even on relatively small systems. However, one should not underestimate the administrational overhead involved.

- It is not a good idea to use parallelization for very small problems.
- On multi-user systems with many processors it may be wise to limit the number of threads for Normaliz somewhat below the maximum number of cores.

By default, Normaliz limits the number of threads to 8. One can override this limit by the Normaliz option `-x` (see Section 5.3).

Another way to set an upper limit to the number of threads is via the environment variable `OMP_NUM_THREADS`:

```
export OMP_NUM_THREADS=<T>      (Linux/Mac)
```

or

```
set OMP_NUM_THREADS=<T>      (Windows)
```

where `<T>` stands for the maximum number of threads accessible to Normaliz. For example, we often use

```
export OMP_NUM_THREADS=20
```

on a multi-user system system with 24 cores.

Limiting the number of threads to 1 forces a strictly serial execution of Normaliz.

The paper [8] contains extensive data on the effect of parallelization. On the whole Normaliz scales very well. However, the dual algorithm often performs best with mild parallelization, say with 4 or 6 threads.

8.2. Running large computations

Normaliz can cope with very large examples, but it is usually difficult to decide a priori whether an example is very large, but nevertheless doable, or simply impossible. Therefore some exploration makes sense. The following applies to the primal algorithm.

See [8] for some very large computations. The following hints reflect the authors' experience with them.

(1) Run Normaliz with the option `-cs` and pay attention to the terminal output. The number of extreme rays, but also the numbers of support hyperplanes of the intermediate cones are useful data.

(2) In many cases the most critical size parameter for the primal algorithm is the number of simplicial cones in the triangulation. It makes sense to determine it as the next step. Even with the fastest potential evaluation (option `-v` or `TriangulationDetSum`), finding the triangulation takes less time, say by a factor between 3 and 10. Thus it makes sense to run the example with `-t` in order to explore the size.

As you can see from [8], Normaliz has successfully evaluated triangulations of size $\approx 5 \cdot 10^{11}$ in dimension 24.

(3) Another critical parameter are the determinants of the generator matrices of the simplicial cones. To get some feeling for their sizes, one can restrict the input to a subset (of the extreme rays computed in (1)) and use the option `-v` or the computation goal `TriangulationDetSum` if there is no grading.

The output file contains the number of simplicial cones as well as the sum of the absolute values of the determinants. The latter is the number of vectors to be processed by Normaliz in triangulation based calculations.

The number includes the zero vector for every simplicial cone in the triangulation. The zero vector does not enter the Hilbert basis calculation, but cannot be neglected for the Hilbert series.

Normaliz has mastered calculations with $> 10^{15}$ vectors.

(4) If the triangulation is small, we can add the option `-T` in order to actually see the triangulation in a file. Then the individual determinants become visible.

(5) If a cone is defined by inequalities and/or equations consider the dual mode for Hilbert basis calculation, even if you also want the Hilbert series.

(6) The size of the triangulation and the size of the determinants are *not* dangerous for memory by themselves (unless `-T` or `-y` are set). Critical magnitudes can be the number of support hyperplanes, Hilbert basis candidates, or degree 1 elements.

9. Distribution and installation

9.1. Docker image

The easiest and absolutely hassle free access to Normaliz is via its Docker image. To run it, you must first install Docker on your system. This is easy on up-to-date versions of the three major platforms. After installation you can issue the command

```
docker run -ti normaliz/normaliz
```

This will download the Docker image if it is not yet present and open a Docker container. As a result you will get a Linux terminal. Normaliz is installed in the standard location `/usr/local`. Moreover, the source is contained in the subdirectory `Normaliz` of the home directory. (Your username is `norm`.) In the Docker container `Normaliz` is the Normaliz directory (independently of the version number).

Try

```
normaliz -c Normaliz/exampe/small
```

as a first test.

Of course, you want to make your data available to Normaliz in the container. The Docker documentation tells you how to do it.

The command above downloads the image labeled "latest". It is produced from the branch "release" on GitHub, which always contains the latest official release. There is a second image, produced from the branch "master". It is updated with every commit to this branch. You can get it by appending `:master` to the command above.

The Docker image contains a full installation including PyNormaliz and PyQNormaliz.

9.2. Basic package

In order to install Normaliz you should first download the basic package. Follow the instructions in

`http://normaliz.uos.de/download/`.

They guide you to our GitHub repository

`https://github.com/Normaliz/Normaliz/releases`.

The full version `normaliz-3.6.1`-full.zip of the basic package contains the documentation, examples, source code, jNormaliz, and the packages for PyNormaliz, Singular and Macaulay2.

The minimal version `normaliz-3.6.1.zip` (also available as `.tar.gz`) contains the source files, examples, documentation and PyNormaliz.

Then unzip the downloaded file in a directory of your choice. (Any other downloaded zip file for Normaliz should be unzipped in this directory, too.)

This process will create a directory `normaliz-3.6.1` (called Normaliz directory) and several subdirectories in it. The names of the subdirectories created are self-explanatory. Nevertheless we give an overview:

- In the Normaliz directory you should find `jNormaliz.jar`, several files, for example installation scripts mentioned below, and subdirectories.
- The subdirectory `source` contains the source files.
- The subdirectory `doc` contains the file you are reading and further documentation.
- In the subdirectory `example` are the input files for some examples. It contains all named input files of examples of this manual.
- Automated tests which run Normaliz on different inputs and options are contained in the subdirectory `test`.
- The subdirectory `Singular` contains the SINGULAR library `normaliz.lib` and a PDF file with documentation.
- The subdirectory `Macaulay2` contains the MACAULAY2 package `Normaliz.m2`.
- The subdirectory `PyNormaliz` contains the source PYTHON interface.
- The subdirectory `lib` contains libraries for jNormaliz.
- Moreover, there are subdirectories whose name starts with or contains `Q`. They contain the source code, examples and tests for QNormaliz (see Appendix F)

If you build Normaliz yourself, ubdirectories `nmz_opt_lib` and `local` will be created (with rthe fefault settings).

9.3. Executables

We provide executables for Windows, Linux and Mac. Download the archive file corresponding to your system `normaliz-3.6.1<systemname>.zip` and unzip it. This process will store the Normaliz executable in the Normaliz directory. In case you want to run Normaliz from the command line or use it from other systems, you may have to copy the executables to a directory in the search path for executables or update your search path.

The MS Windows executable is compiled without any optional package; in particular there is no `Qnormaliz`. However, the Linux 64 binary runs in the Linux subsystem of Windows 10. It ias extremely easy to install the Linux subsystem.

Note:

1. The linux binaries `normaliz` and `Qnormaluiz` are truly static executables.
2. The Mac OS binaries cannot be statically linked in the absolute sense, but they depend only on Mac OS system libraries and `libomp.dylib` (contained in the zip file) which makes parallelization possible. This dynamic library must be kept in the same directoory as the binaries.
3. For MS Windows we provide `libiomp5md.dll` for parallelization. Keep it in the same

directory as `normaliz.exe`.

9.4. Cloning the GitHub repository

Another way to download the Normaliz source is cloning the repository from GitHub by

```
git clone https://github.com/Normaliz/Normaliz.git
```

The Normaliz directory is then called `Normaliz`. You may need

```
sudo apt-get install autoconf libtool
```

if you want to build Normaliz by autotools. To this end change to the Normaliz directory and run

```
./bootstrap.sh
```

After this step you can follow the instruction in the next section.

Note that he GitHub repository Normaliz/Normaliz does not contain Py(Q)Normaliz. You can clone them from the repositories Normaliz/PyNormaliz and Normaliz/PyQNormaliz.

10. Building Normaliz yourself

We recommend building Normaliz through the install scripts described below. They use the `autotools` scripts have been written by Matthias Kï¿½ppe. The Normaliz team thanks him cordially for his generous help.

In previous versions it was also possible to build Normaliz through cmake. The cmake files do still exist, but theyare not up-to-date.

10.1. Prerequisites

We require some C++11 features (e.g. `std::exception_ptr`), supported by:

- GNU g++ 4.4,
- clang++ 2.9,
- Intel icpc 12.0

See `https://github.com/Normaliz/Normaliz/issues/26` for a more detailed discussion.

The mentioned compilers are also able to handle OpenMP 3.0, with the exception of clang++, there the first OpenMP support was introduced in 3.7.

For compiling Normaliz the following libraries are needed:

- GMP including the C++ wrapper (libgmpxx and libgmp)
- Boost (headers only)

We will only discuss how to build Normaliz with the install scrips in the distribution. See the file `INSTALL` for additional information.

Any optional package that you want to use, must be installed before the compilation of Normaliz, independently of the method used for building Normaliz. The installation scripts mentioned below make and use directories within the Normaliz directory.

10.1.1. Linux

The standard compiler choice on Linux is `g++`. We do no not recommend `clang++` since its support for OpenMP is not as comprehensive as that of `g++`.

On Ubunto we suggest

```
sudo apt-get install tar g++ libgmp-dev libboost-dev wget make
```

for the basic installation of the required libraries (plus compiler).

10.1.2. Mac OS X

Currently Apple does not supply a compiler which supports OpenMP. The install scripts discussed below *require LLVM 3.9 or newer from Homebrew.* See `http://brew.sh/` from where you can also download GMP and Boost:

```
brew install gmp boost llvm wget
```

You also need to download and install the Xcode Command Line Tools from the AppStore:

```
xcode-select --install
```

10.2. Normaliz at a stroke

Navigate to the Normaliz directory. The command

```
./install_normaliz_with_opt.sh
```

downloads the optional packages and compiles Normaliz (and a basic version of QNormaliz). Alternatively, you can call

```
./install_normaliz_with_qnormaliz_eantic.sh
```

It installs the optional packages that are needed for the computation of algebraic polytopes by QNormaliz and does the full compilation.

The sources of the optional packages are dowloaded to the subdirectory `nmz_opt_lib` of the Normaliz directory. They are installed in the subdirectory `local` (imitating `/usr/local`) where they exist in static and dynamic versions (except CoCoALib that can only be built statically).

The libraries `libnormaliz` and `libQnormaliz` are compiled statically and shared. They are installed in `local` as well.

The binaries `normaliz` and `Qnormaliz` are stored in `local/bin`, but they are lalso coiped to the Normaliz directory. They are not linked statically, but they depend only on shared objects provided by the operating system.

Remarks:

(1) For e-antic one needs a special version of Flint. If you should have installed Normaliz without e-antic, you must delete the Flint source in `nmz_opt_lib` before installing Normaliz with e-antic.

(2) By

```
export NMZ_SHARED=yes
./install_normaliz_with_...
```

one can build Normaliz completely with shared libraries.

(3) It is possible to compile statically linked Linux versions of `normaliz` and `Qnormaliz` through the `Makefile.classic` in the directories `source` and `Qsource` after the installation of the optional libaries.

(4) If you want a global installation (and have the rights to do it), you can ask for

```
sudo cp -r local /usr
```

at the end.

(5) Another way to a global installation (or to an installation in a place of your choice) is to use

```
export NMZ_PREFIX=<your choice>
./install_normaliz_with_...
```

For the typical choice `/usr/local` you need superuser rights (as in (4)). Note that `NMZ_PREFIX` must be an absolute path name.

(6) If you don't want the optional packages or if you have them already (in the right location, as mentioned above),

```
./install_normaliz.sh
```

compiles Normaliz and Qnormaliz , using the optional packages that it can find.

(7) The install scripts can be further customized. Have a look at them or at `INSTALL`.

10.3. Optional packages

Here we only discuss the optional packages used by Normaliz itself. See Appendix F for the optional packages of QNormaliz.

10.3.1. CoCoALib

Normaliz can be built without CoCoALib, which is however necessary for the computation of integrals and weighted Ehrhart series and, hence, for symmetrization. If you want to compile Normaliz with CoCoALib, install CoCoALib first by navigating to the Normaliz directory and typeing the command

```
./install_nmz_cocoa.sh
```

CoCoALib is downloaded and compiled as described above.

10.3.2. Flint

Flint does not extend the functionality of Normaliz, and is therefore truly optional. However, the ultrafast polynomial arithmetic of Flint is very useful if quasipolynomials with large periods come up in the computation of Hilbert series or weighted Ehrhart series. Install Flint (and MPFR) navigating to the Normaliz directory and typing the command

```
./install_nmz_flint.sh
```

10.3.3. SCIP

As discussed in the manual, Normaliz can use SCIP, but Normaliz' own method seems to be superior. See the file `INSTALL` for instructions on the use of SCIP.

10.4. Windows

One can compile Windows executables with the Cygwin port of GCC. Unfortunately it is not compatible to OpenMP.

Using Visual Studio is a bit tricky. Microsoft's C++ compiler does not support OpenMP 3.0. Creating a Normaliz Visual Studio project via cmake is currently not fully supported. The executables that are offered in the Normaliz distribution have been compiled with Intel icpc and a manually created project. Please contact us if you want to build Normaliz on Windows.

Note that the statically linked Linux binaries run in the Linux subsystem of Windows 10. We have not yet tried to build Normaliz in it. And the Docker image is of course an excellent alternative.

11. Copyright and how to cite

Normaliz 3.1 is free software licensed under the GNU General Public License, version 3. You can redistribute it and/or modify it under the terms of the GNU General Public License as

published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

It is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with the program. If not, see `http://www.gnu.org/licenses/`.

Please refer to Normaliz in any publication for which it has been used:

W. Bruns, B. Ichim, T. Rï¿½mer, R. Sieg and C. Sï¿½ger: Normaliz. Algorithms for rational cones and affine monoids. Available at `http://normaliz.uos.de`

The corresponding `\bibitem`:

```
\bibitem{Normaliz} W. Bruns, B. Ichim, T. R\"omer, R. Sieg and C. S\"oger:
Normaliz. Algorithms for rational cones and affine monoids.
Available at \url{http://normaliz.uos.de}.
```

A BibTeX entry:

```
@Misc{Normaliz,
author = {W. Bruns and B. Ichim and T. R\"omer and R. Sieg and C. S\"oger},
title = Normaliz. Algorithms for rational cones and affine monoids,
howpublished ={Available at \url{http://normaliz.uos.de}}
```

It is now customary to evaluate mathematicians by such data as numbers of publications, citations and impact factors. The data bases on which such dubious evaluations are based do not list mathematical software. Therefore we ask you to cite the article [8] in addition. This is very helpful for the younger members of the team.

A. Mathematical background and terminology

For a coherent and thorough treatment of the mathematical background we refer the reader to [5].

A.1. Polyhedra, polytopes and cones

An *affine halfspace* of $\mathbb{R}^d$ is a subset given as

$$H_\lambda^+ = \{x : \lambda(x) \geq 0\},$$

where λ is an affine form, i.e., a non-constant map $\lambda : \mathbb{R}^d \to \mathbb{R}$, $\lambda(x) = \alpha_1 x_1 + \cdots + \alpha_d x_d + \beta$ with $\alpha_1, \ldots, \alpha_d, \beta \in \mathbb{R}$. If $\beta = 0$ and λ is therefore linear, then the halfspace is called *linear*. The halfspace is *rational* if λ is *rational*, i.e., has rational coordinates. If λ is rational, we can assume that it is even *integral*, i.e., has integral coordinates, and, moreover, that these are coprime. Then λ is uniquely determined by H_λ^+. Such integral forms are called *primitive*, and the same terminology applies to vectors.

Definition 1. A (rational) *polyhedron* P is the intersection of finitely many (rational) halfspaces. If it is bounded, then it is called a *polytope*. If all the halfspaces are linear, then P is a *cone*.

The *dimension* of P is the dimension of the smallest affine subspace $\mathrm{aff}(P)$ containing P.

A support hyperplane of P is an affine hyperplane H that intersects P, but only in such a way that H is contained in one of the two halfspaces determined by H. The intersection $H \cap P$ is called a *face* of P. It is a polyhedron (polytope, cone) itself. Faces of dimension 0 are called *vertices*, those of dimension 1 are called *edges* (in the case of cones *extreme rays*), and those of dimension $\dim(P) - 1$ are *facets*.

When we speak of *the* support hyperplanes of P, then we mean those intersecting P in a facet. Their halfspaces containing P cut out P from $\mathrm{aff}(P)$. If $\dim(P) = d$, then they are uniquely determined (up to a positive scalar).

The constraints by which Normaliz describes polyhedra are

(1) linear equations for $\mathrm{aff}(P)$ and
(2) linear inequalities (simply called support hyperplanes) cutting out P from $\mathrm{aff}(P)$.

In other words, the constraints are given by a linear system of equations and inequalities, and a polyhedron is nothing else than the solution set of a linear system of inequalities and equations. It can always be represented in the form

$$Ax \geq b, \qquad A \in \mathbb{R}^{m \times d}, b \in \mathbb{R}^m,$$

if we replace an equation by two inequalities.

A.2. Cones

The definition describes a cone by constraints. One can equivalently describe it by generators:

Theorem 2 (Minkowski-Weyl)**.** *The following are equivalent for $C \subset \mathbb{R}^d$;*

1. *C is a (rational) cone;*
2. *there exist finitely many (rational) vectors $x_1, \ldots, x_n$ such that*

$$C = \{a_1x_1 + \cdots + a_nx_n : a_1, \ldots, a_n \in \mathbb{R}_+\}.$$

By $\mathbb{R}_+$ we denote the set of nonnegative real numbers; $\mathbb{Q}_+$ and $\mathbb{Z}_+$ are defined in the same way.

The conversion between the description by constraints and that by generators is one of the basic tasks of Normaliz. It uses the *Fourier-Motzkin elimination*.

Let C_0 be the set of those $x \in C$ for which $-x \in C$ as well. It is the largest vector subspace contained in C. A cone is *pointed* if $C_0 = 0$. If a rational cone is pointed, then it has uniquely determined *extreme integral generators*. These are the primitive integral vectors spanning the extreme rays. These can also be defined with respect to a sublattice L of $\mathbb{Z}^d$, provided C is contained in $\mathbb{R}L$. If a cone is not pointed, then Normaliz computes the extreme rays of the pointed C/C_0 and lifts them to C. (Therefore they are only unique modulo C_0.)

The *dual cone* C^* is given by

$$C^* = \{\lambda \in (\mathbb{R}^d)^* : \lambda(x) \geq 0 \text{ for all } x \in C\}.$$

Under the identification $\mathbb{R}^d = (\mathbb{R}^d)^{**}$ one has $C^{**} = C$. Then one has

$$\dim C_0 + \dim C^* = d.$$

In particular, C is pointed if and only if C^* is full dimensional, and this is the criterion for pointedness used by Normaliz. Linear forms $\lambda_1, \ldots, \lambda_n$ generate C^* if and only if C is the intersection of the halfspaces $H^+_{\lambda_i}$. Therefore the conversion from constraints to generators and its converse are the same task, except for the exchange of $\mathbb{R}^d$ and its dual space.

A.3. Polyhedra

In order to transfer the Minkowski-Weyl theorem to polyhedra it is useful to homogenize coordinates by embedding $\mathbb{R}^d$ as a hyperplane in $\mathbb{R}^{d+1}$, namely via

$$\kappa : \mathbb{R}^d \to \mathbb{R}^{d+1}, \qquad \kappa(x) = (x, 1).$$

If P is a (rational) polyhedron, then the closure of the union of the rays from 0 through the points of $\kappa(P)$ is a (rational) cone $C(P)$, called the *cone over P*. The intersection $C(P) \cap (\mathbb{R}^d \times \{0\})$ can be identified with the *recession* (or tail) *cone*

$$\text{rec}(P) = \{x \in \mathbb{R}^d : y + x \in P \text{ for all } y \in P\}.$$

It is the cone of unbounded directions in P. The recession cone is pointed if and only if P has at least one bounded face, and this is the case if and only if it has a vertex.

The theorem of Minkowski-Weyl can then be generalized as follows:

Theorem 3 (Motzkin)**.** *The following are equivalent for a subset $P \neq \emptyset$ of $\mathbb{R}^d$:*

1. *P is a (rational) polyhedron;*
2. *$P = Q + C$ where Q is a (rational) polytope and C is a (rational) cone.*

If P has a vertex, then the smallest choice for Q is the convex hull of its vertices, and $C = \mathrm{rec}(P)$ is uniquely determined.

The *convex hull* of a subset $X \in \mathbb{R}^d$ is

$$\mathrm{conv}(X) = \{a_1x_1 + \cdots + a_nx_n : n \geq 1, x_1, \ldots, x_n \in X, a_1, \ldots, a_n \in \mathbb{R}_+, a_1 + \cdots + a_n = 1\}.$$

Clearly, P is a polytope if and only if $\mathrm{rec}(P) = \{0\}$, and the specialization to this case one obtains Minkowski's theorem: a subset P of $\mathbb{R}^d$ is a polytope if and only if it is the convex hull of a finite set. A *lattice polytope* is distinguished by having integral points as vertices.

Normaliz computes the recession cone and the polytope Q if P is defined by constraints. Conversely it finds the constraints if the vertices of Q and the generators of C are specified.

Suppose that P is given by a system

$$Ax \geq b, \qquad A \in \mathbb{R}^{m\times d},\ b \in \mathbb{R}^m,$$

of linear inequalities (equations are replaced by two inequalities). Then $C(P)$ is defined by the *homogenized system*

$$Ax - x_{d+1}b \geq 0$$

whereas the $\mathrm{rec}(P)$ is given by the *associated homogeneous system*

$$Ax \geq 0.$$

It is of course possible that P is empty if it is given by constraints since inhomogeneous systems of linear equations and inequalities may be unsolvable. By abuse of language we call the solution set of the associated homogeneous system the recession cone of the system.

Via the concept of dehomogenization, Normaliz allows for a more general approach. The *dehomogenization* is a linear form δ on $\mathbb{R}^{d+1}$. For a cone $\widetilde{C}$ in $\mathbb{R}^{d+1}$ and a dehomogenization δ, Normaliz computes the polyhedron $P = \{x \in \widetilde{C} : \delta(x) = 1\}$ and the recession cone $C = \{x \in \widetilde{C} : \delta(x) = 0\}$. In particular, this allows other choices of the homogenizing coordinate. (Often one chooses x_0, the first coordinate then.)

In the language of projective geometry, $\delta(x) = 0$ defines the hyperplane at infinity.

A.4. Affine monoids

An *affine monoid* M is a finitely generated submonoid of $\mathbb{Z}^d$ for some $d \geq 0$. This means: $0 \in M$, $M+M \subset M$, and there exist $x_1, \ldots, x_n$ such that

$$M = \{a_1x_1 + \cdots + a_nx_n : a_1, \ldots, a_n \in \mathbb{Z}_+\}.$$

We say that $x_1, \ldots, x_n$ is a *system of generators* of M. A monoid M is positive if $x \in M$ and $-x \in M$ implies $x = 0$. An element x in a positive monoid M is called *irreducible* if it has no decomposition $x = y + z$ with $y, z \in M$, $y, z \neq 0$. The *rank* of M is the rank of the subgroup $\mathrm{gp}(M)$ of $\mathbb{Z}^d$ generated by M. (Subgroups of $\mathbb{Z}^d$ are also called sublattices.) For certain aspects of monoid theory it is very useful (or even necessary) to introduce coefficients from a field K (or a more general commutative ring) and consider the monoid algebra $K[M]$.

Theorem 4 (van der Corput)**.** *Every positive affine monoid M has a unique minimal system of generators, given by its irreducible elements.*

We call the minimal system of generators the *Hilbert basis* of M. Normaliz computes Hilbert bases of a special type of affine monoid:

Theorem 5 (Gordan's lemma)**.** *Let $C \subset \mathbb{R}^d$ be a (pointed) rational cone and let $L \subset \mathbb{Z}^d$ be a sublattice. Then $C \cap L$ is a (positive) affine monoid.*

The monoids $M = C \cap L$ of the theorem have the pleasant property that the group of units M_0 (i.e., elements whose inverse also belongs to M) splits off as a direct summand. Therefore M/M_0 is a well-defied affine monoid. If M is not positive, then Normaliz computes a Hilbert basis of M/M_0 and lifts it to M.

Let $M \subset \mathbb{Z}^d$ be an affine monoid, and let $N \supset M$ be an overmonoid (not necessarily affine), for example a sublattice L of $\mathbb{Z}^d$ containing M.

Definition 6. The *integral closure* (or *saturation*) of M in N is the set

$$\widehat{M}_N = \{x \in N : kx \in M \text{ for some } k \in \mathbb{Z}, k > 0\}.$$

If $\widehat{M}_N = M$, one calls M *integrally closed* in N.

The integral closure $\overline{M}$ of M in $\mathrm{gp}(M)$ is its *normalization*. M is *normal* if $\overline{M} = M$.

The integral closure has a geometric description:

Theorem 7.

$$\widehat{M}_N = \mathrm{cone}(M) \cap N.$$

Combining the theorems, we can say that Normaliz computes integral closures of affine monoids in lattices, and the integral closures are themselves affine monoids as well. (More generally, $\widehat{M}_N$ is affine if M and N are affine.)

In order to specify the intersection $C \cap L$ by constraints we need a system of homogeneous inequalities for C. Every sublattice of $\mathbb{Z}^d$ can be written as the solution set of a combined system of homogeneous linear diophantine equations and a homogeneous system of congruences (this follows from the elementary divisor theorem). Thus $C \cap L$ is the solution set of a homogeneous linear diophantine system of inequalities, equations and congruences. Conversely, the solution set of every such system is a monoid of type $C \cap L$.

In the situation of Theorem 7, if $\mathrm{gp}(N)$ has finite rank as a $\mathrm{gp}(M)$-module, $\widehat{M}_N$ is even a finitely generated module over M. I.e., there exist finitely many elements $y_1, \ldots, y_m \in \widehat{M}_N$ such that $\widehat{M}_N = \bigcup_{i=1}^m M + y_i$. Normaliz computes a minimal system $y_1, \ldots, y_m$ and lists the nonzero y_i as a system of module generators of $\widehat{M}_N$ modulo M. We must introduce coefficients to make this precise: Normaliz computes a minimal system of generators of the $K[M]$-module $K[\widehat{M}_N]/K[M]$.

A.5. Affine monoids from binomial ideals

Let U be a subgroup of $\mathbb{Z}^n$. Then the natural image M of $\mathbb{Z}^n_+ \subset \mathbb{Z}^n$ in the abelian group $G = \mathbb{Z}^n/U$ is a submonoid of G. In general, G is not torsionfree, and therefore M may not be an affine monoid. However, the image N of M in the lattice $L = G/\mathrm{torsion}(G)$ is an affine monoid. Given U, Normaliz chooses an embedding $L \hookrightarrow \mathbb{Z}^r$, $r = n - \mathrm{rank}\, U$, such that N becomes a submonoid of $\mathbb{Z}^r_+$. In general there is no canonical choice for such an embedding, but one can always find one, provided N has no invertible element except 0.

The typical starting point is an ideal $J \subset K[X_1, \ldots, X_n]$ generated by binomials

$$X_1^{a_1} \cdots X_n^{a_n} - X_1^{b_1} \cdots X_n^{b_n}.$$

The image of $K[X_1, \ldots, X_n]$ in the residue class ring of the Laurent polynomial ring $S = K[X_1^{\pm 1}, \ldots, X_n^{\pm 1}]$ modulo the ideal JS is exactly the monoid algebra $K[M]$ of the monoid M above if we let U be the subgroup of $\mathbb{Z}^n$ generated by the differences

$$(a_1, \ldots, a_n) - (b_1, \ldots, b_n).$$

Ideals of type JS are called lattice ideals if they are prime. Since Normaliz automatically passes to $G/\mathrm{torsion}(G)$, it replaces JS by the smallest lattice ideal containing it.

A.6. Lattice points in polyhedra

Let $P \subset \mathbb{R}^d$ be a rational polyhedron and $L \subset \mathbb{Z}^d$ be an *affine sublattice*, i.e., a subset $w + L_0$ where $w \in \mathbb{Z}^d$ and $L_0 \subset \mathbb{Z}^d$ is a sublattice. In order to investigate (and compute) $P \cap L$ one again uses homogenization: P is extended to $C(P)$ and L is extended to $\mathscr{L} = L_0 + \mathbb{Z}(w, 1)$. Then one computes $C(P) \cap \mathscr{L}$. Via this "bridge" one obtains the following inhomogeneous version of Gordan's lemma:

Theorem 8. *Let P be a rational polyhedron with vertices and $L = w + L_0$ an affine lattice as above. Set* $\mathrm{rec}_L(P) = \mathrm{rec}(P) \cap L_0$. *Then there exist* $x_1, \ldots, x_m \in P \cap L$ *such that*

$$P \cap L = \{(x_1 + \mathrm{rec}_L(P)) \cap \cdots \cap (x_m + \mathrm{rec}_L(P))\}.$$

If the union is irredundant, then $x_1, \ldots, x_m$ are uniquely determined.

The Hilbert basis of $\mathrm{rec}_L(P)$ is given by $\{x : (x,0) \in \mathrm{Hilb}(C(P) \cap \mathscr{L})\}$ and the minimal system of generators can also be read off the Hilbert basis of $C(P) \cap \mathscr{L}$: it is given by those x for which $(x,1)$ belongs to $\mathrm{Hilb}(C(P) \cap \mathscr{L})$. (Normaliz computes the Hilbert basis of $C(P) \cap L$ only at "levels" 0 and 1.)

We call $\mathrm{rec}_L(P)$ the *recession monoid* of P with respect to L (or L_0). It is justified to call $P \cap L$ a *module* over $\mathrm{rec}_L(P)$. In the light of the theorem, it is a finitely generated module, and it has a unique minimal system of generators.

After the introduction of coefficients from a field K, $\mathrm{rec}_L(P)$ is turned into an affine monoid algebra, and $N = P \cap L$ into a finitely generated torsionfree module over it. As such it has a well-defined *module rank* $\mathrm{mrank}(N)$, which is computed by Normaliz via the following combinatorial description: Let $x_1, \ldots, x_m$ be a system of generators of N as above; then $\mathrm{mrank}(N)$ is the cardinality of the set of residue classes of $x_1, \ldots, x_m$ modulo $\mathrm{rec}_L(P)$.

Clearly, to model $P \cap L$ we need linear diophantine systems of inequalities, equations and congruences which now will be inhomogeneous in general. Conversely, the set of solutions of such a system is of type $P \cap L$.

A.7. Hilbert series and multiplicity

Normaliz can compute the Hilbert series and the Hilbert (quasi)polynomial of a graded monoid. A *grading* of a monoid M is simply a homomorphism $\deg : M \to \mathbb{Z}^g$ where $\mathbb{Z}^g$ contains the degrees. The *Hilbert series* of M with respect to the grading is the formal Laurent series

$$H(t) = \sum_{u \in \mathbb{Z}^g} \#\{x \in M : \deg x = u\} t_1^{u_1} \cdots t_g^{u_g} = \sum_{x \in M} t^{\deg x},$$

provided all sets $\{x \in M : \deg x = u\}$ are finite. At the moment, Normaliz can only handle the case $g = 1$, and therefore we restrict ourselves to this case. We assume in the following that $\deg x > 0$ for all nonzero $x \in M$ and that there exists an $x \in \mathrm{gp}(M)$ such that $\deg x = 1$. (Normaliz always rescales the grading accordingly – as long as no module N is involved.) In the case of a nonpositive monoid, these conditions must hold for M/M_0, and its Hilbert series is considered as the Hilbert series of M.

The basic fact about $H(t)$ in the $\mathbb{Z}$-graded case is that it is the Laurent expansion of a rational function at the origin:

Theorem 9 (Hilbert, Serre; Ehrhart)**.** *Suppose that M is a normal positive affine monoid. Then*

$$H(t) = \frac{R(t)}{(1-t^e)^r}, \qquad R(t) \in \mathbb{Z}[t],$$

where r is the rank of M and e is the least common multiple of the degrees of the extreme integral generators of $\mathrm{cone}(M)$. *As a rational function, $H(t)$ has negative degree.*

The statement about the rationality of $H(t)$ holds under much more general hypotheses.

Usually one can find denominators for $H(t)$ of much lower degree than that in the theorem, and Normaliz tries to give a more economical presentation of $H(t)$ as a quotient of two polynomials. One should note that it is not clear what the most natural presentation of $H(t)$ is in general (when $e > 1$). We discuss this problem in [8, Section 4]. The examples 2.5 and 2.6.2, may serve as an illustration.

A rational cone C and a grading together define the rational polytope $Q = C \cap A_1$ where $A_1 = \{x : \deg x = 1\}$. In this sense the Hilbert series is nothing but the Ehrhart series of Q. The following description of the Hilbert function $H(M,k) = \#\{x \in M : \deg x = k\}$ is equivalent to the previous theorem:

Theorem 10. *There exists a quasipolynomial q with rational coefficients, degree* $\mathrm{rank}\, M - 1$ *and period π dividing e such that $H(M,k) = q(k)$ for all $q \geq 0$.*

The statement about the quasipolynomial means that there exist polynomials $q^{(j)}$, $j = 0, \ldots, \pi - 1$, of degree $\mathrm{rank}\, M - 1$ such that

$$q(k) = q^{(j)}(k), \qquad j \equiv k \quad (\pi),$$

and

$$q^{(j)}(k) = q_0^{(j)} + q_1^{(j)} k + \cdots + q_{r-1}^{(j)} k^{r-1}, \qquad r = \mathrm{rank}\, M,$$

with coefficients $q_i^{(j)} \in \mathbb{Q}$. It is not hard to show that in the case of affine monoids all components have the same degree $r-1$ and the same leading coefficient:

$$q_{r-1} = \frac{\mathrm{vol}(Q)}{(r-1)!},$$

where vol is the lattice normalized volume of Q (a lattice simplex of smallest possible volume has volume 1). The *multiplicity* of M, denoted by $e(M)$ is $(r-1)!q_{r-1} = \mathrm{vol}(Q)$.

Suppose now that P is a rational polyhedron in $\mathbb{R}^d$, $L \subset \mathbb{Z}^d$ is an affine lattice, and we consider $N = P \cap L$ as a module over $M = \mathrm{rec}_L(P)$. Then we must give up the condition that deg takes the value 1 on $\mathrm{gp}(M)$ (see Section 6.20 for an example). But the Hilbert series

$$H_N(t) = \sum_{x \in N} t^{\deg x}$$

is well-defined, and the qualitative statement above about rationality remain valid. However, in general the quasipolynomial gives the correct value of the Hilbert function only for $k > r$ where r is the degree of the Hilbert series as a rational function. The multiplicity of N is given by

$$e(N) = \mathrm{mrank}(N) e(M).$$

where mrank(M) is the module rank of M.

Since N may have generators in negative degrees, Normaliz shifts the degrees into $\mathbb{Z}_+$ by subtracting a constant, called the *shift*. (The shift may also be positive.)

Above the multiplicity of M was defined under the assumption that gp(M) contains an element of degree 1. In the homogeneous situation where no module N comes into play, Normaliz achieves this extra condition by dividing the grading by the *grading denominator* so that we are effectively in the situation considered above, except in two situations: (i) the use of the grading denominator is blocked; (ii) when a module N is considered, it can easily happen that the grading restricted to the recession monoid M has a denominator $g > 1$, but there occur degrees in N that are not divisible by g. Let $\deg' = \deg/g$ and let $e'(M)$ be the multiplicity of M with respect to $\deg'$. Then

$$e(M) = \frac{e'(M)}{g^{r-1}}.$$

With this definition, $e(M)$ has the expected property as a dimension normed leading coefficient of the Hulbert quesipolynomial: if $q^{(j)}$ is a *nonzero* component of the quasipolynomial of M, then its leading coefficient satisfies

$$q^{(j)}_{r-1} = \frac{e(M)}{(r-1)!}.$$

This follows immadiately from the substitution $k \mapsto k/g$ in the Hilbert function when we pass from $\deg'$ to deg: $H(M,k) = H'(M,k/g)$ if g divides k and $H(M.k) = 0$ otherwise. Also the interpretation as a volume is consistent: $e(M)$ is the lattice normalized volume of the polytope $C \cap \{x : \deg x = 1\}$ (whereas $e'(M)$ is the lattice normalized volume of $C \cap \{x : \deg x = g\}$).

For the interpretation of the multiplicity $e(N) = \text{mrank}(N)e(M)$ one must first split the module N into a direct sum where each summand bundles the elements whose degrees belong to a fixed residue class modulo g. Let $N^0, \dots, N^{g-1}$ be these summands. Then $e(N^k)$ is the dimension normed constant leading coefficient of the Hilbert quasipolynomial of N^k for each k, and $e(N) = \sum_k e(N^k)$.

A.8. The class group

A normal affine monoid M has a well-defined divisor class group. It is naturally isomorphic to the divisor class group of $K[M]$ where K is a field (or any unique factorization domain); see [5, 4.F], and especially [5, 4.56]. The class group classifies the divisorial ideals up to isomorphism. It can be computed from the standard embedding that sends an element x of gp(M) to the vector $\sigma(x)$ where σ is the collection of support forms $\sigma_1, \dots, \sigma_s$ of M: $\text{Cl}(M) = \mathbb{Z}^s/\sigma(\text{gp}(M))$. Finding this quotient amounts to an application of the Smith normal form to the matrix of σ.

B. Annotated console output

B.1. Primal mode

With

```
./normaliz -ch example/A443
```

we get the following terminal output.

```
                                                    \.....|
                  Normaliz 3.2.0                     \....|
                                                      \...|
    (C) The Normaliz Team, University of Osnabrueck    \..|
                  January  2017                         \.|
                                                         \|
************************************************************
Command line: -ch example/A443
Compute: HilbertBasis HilbertSeries
************************************************************
starting primal algorithm with full triangulation ...
Roughness 1
Generators sorted by degree and lexicographically
Generators per degree:
1: 48
```

Self explanatory so far (see Section 6.3 for the definition of roughness). Now the generators are inserted.

```
Start simplex 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 22 25 26 27 28 31 34
37 38 39 40 43 46
```

Normaliz starts by searching linearly independent generators with indices as small as possible. They span the start simplex in the triangulation. The remaining generators are inserted successively. (If a generator does not increase the cone spanned by the previous ones, it is not listed, but this does not happen for `A443`.)

```
gen=17, 39 hyp, 4 simpl
```

We have now reached a cone with 39 support hyperplanes and the triangulation has 4 simplices so far. We omit some generators until something interesting happens:

```
gen=35, 667 hyp, 85 pyr, 13977 simpl
```

In view of the number of simplices in the triangulation and the number of support hyperplanes, Normaliz has decided to build pyramids and to store them for later triangulation.

```
gen=36, 723 hyp, 234 pyr, 14025 simpl
...
```

```
gen=48, 4948 hyp, 3541 pyr, 14856 simpl
```

All generators have been processed now. Fortunately our cone is pointed:

```
Pointed since graded
Select extreme rays via comparison ... done.
```

Normaliz knows two methods for finding the extreme rays. Instead of "comparison" you may see "rank". Now the stored pyramids must be triangulated. They may produce not only simplices, but also pyramids of higher level, and indeed they do so:

```
**************************************************
level 0 pyramids remaining: 3541
**************************************************
**************************************************
all pyramids on level 0 done!
**************************************************
level 1 pyramids remaining: 5935
**************************************************
**************************************************
all pyramids on level 1 done!
**************************************************
level 2 pyramids remaining: 1567
**************************************************
1180 pyramids remaining on level 2, evaluating 2503294 simplices
```

At this point the preset size of the evaluation buffer for simplices has been exceeded. Normaliz stops the processing of pyramids, and empties the buffer by evaluating the simplices.

```
||||||||||||||||||||||||||||||||||||||||||||||||||
2503294 simplices, 0 HB candidates accumulated.
**************************************************
all pyramids on level 2 done!
**************************************************
level 3 pyramids remaining: 100
**************************************************
**************************************************
all pyramids on level 3 done!
```

This is a small computation, and the computation of pyramids goes level by level without the necessity to return to a lower level. But in larger examples the buffer for level $n+1$ may be filled before level n is finished. Then it becomes necessary to go back. Some simplices remaining in the buffer are now evaluated:

```
evaluating 150978 simplices
||||||||||||||||||||||||||||||||||||||||||||||||||
2654272 simplices, 0 HB candidates accumulated.
Adding 1 denominator classes... done.
```

Since our generators form the Hilbert basis, we do not collect any further candidates. If all generators are in degree 1, we have only one denominator class in the Hilbert series, but otherwise there may be many. The collection of the Hilbert series in denominator classes reduces the computations of common denominators to a minimum.

```
Total number of pyramids = 14137, among them simplicial 2994
```

Some statistics of the pyramid decomposition.

```
------------------------------------------------------------
transforming data... done.
```

Our computation is finished.

A typical pair of lines that you will see for other examples is

```
auto-reduce 539511 candidates, degrees <= 1 3 7
reducing 30 candidates by 73521 reducers
```

It tells you that Normaliz has found a list of 539511 new candidates for the Hilbert basis, and this list is reduced against itself (auto-reduce). Then the 30 old candidates are reduced against the 73521 survivors of the auto-reduction.

B.2. Dual mode

Now we give an example of a computation in dual mode. It is started by the command

```
./normaliz -cid example/5x5
```

The option `i` is used to suppress the `HSOP` in the input file. The console output:

```
                                                    \.....|
                  Normaliz 3.2.0                     \....|
                                                      \...|
    (C) The Normaliz Team, University of Osnabrueck    \..|
                  January  2017                         \.|
                                                         \|
************************************************************
Command line: -cid example/5x5
Compute: DualMode
No inequalities specified in constraint mode, using non-negative orthant.
************************************************************
```

Indeed, we have used only equations as the input.

```
************************************************************
computing Hilbert basis ...
================================================
cut with halfspace 1 ...
```

```
Final sizes: Pos 1 Neg 1 Neutral 0
```

The cone is cut out from the space of solutions of the system of equations (in this case) by successive intersections with halfspaces defined by the inequalities. After such an intersection we have the positive half space, the "neutral" hyperplane and the negative half space. The final sizes given are the numbers of Hilbert basis elements strictly in the positive half space, strictly in the negative half space, and in the hyperplane. This pattern is repeated until all hyperplanes have been used.

```
================================================
cut with halfspace 2 ...
Final sizes: Pos 1 Neg 1 Neutral 1
```

We leave out some hyperplanes ...

```
================================================
cut with halfspace 20 ...
auto-reduce 1159 candidates, degrees <= 13 27
Final sizes: Pos 138 Neg 239 Neutral 1592
================================================
cut with halfspace 21 ...
Positive: 1027  Negative: 367
................................................
Final sizes: Pos 1094 Neg 369 Neutral 1019
```

Sometimes reduction takes some time, and then Normaliz may issue a message on "auto-reduction" organized by degree (chosen for the algorithm, not defined by the given grading). The line of dots is printed is the computation of new Hilbert basis candidates takes time, and Normaliz wants to show you that it is not sleeping. Normaliz shows you the number of positive and negative partners that must be pared produce offspring.

```
================================================
cut with halfspace 25 ...
Positive: 1856  Negative: 653
................................................
auto-reduce 1899 candidates, degrees <= 19 39
Final sizes: Pos 1976 Neg 688 Neutral 2852
```

All hyperplanes have been taken care of.

```
Find extreme rays
Find relevant support hyperplanes
```

Well, in connection with the equations, some hyperplanes become superfluous. In the output file Normaliz will list a minimal set of support hyperplanes that together with the equations define the cone.

```
Hilbert basis 4828
```

The number of Hilbert basis elements computed is the sum of the last positive and neutral numbers.

```
Find degree 1 elements
```

The input file contains a grading.

```
transforming data... done.
```

Our example is finished.

The computation of the new Hilbert basis after the intersection with the new hyperplane proceeds in rounds, and there can be many rounds . . . (not in the above example). then you can see terminal output like

```
Round 100
Round 200
Round 300
Round 400
Round 500
```

C. Normaliz 2 input syntax

A Normaliz 2 input file contains a sequence of matrices. Comments or options are not allowed in it. A matrix has the format

```
<m>
<n>
<x_1>
...
<x_m>
<type>
```

where <m> denotes the number of rows, <n> is the number of columns and <x_1>...<x_n> are the rows with <m> entries each. All matrix types of Normaliz 3 are allowed (with Normaliz 3), also `grading` and `dehomogenization`. These vectors must be encoded as matrices with 1 row.

The optional output files of with suffix `cst` are still in this format. Just create one and inspect it.

D. libnormaliz

The kernel of Normaliz is the C++ class library `libnormaliz`. It implements all the classes that are necessary for the computations. The central class is `Cone`. It realizes the communication with the calling program and starts the computations most of which are implemented in other classes. In the following we describe the class `Cone`; other classes of `libnormaliz` may follow in the future.

Of course, Normaliz itself is the prime example for the use of libnormaliz, but it is rather complicated because of the input and output it must handle. Therefore we have a added a simple example program at the end of this introduction.

libnormaliz defines its own name space. In the following we assume that

```
using namespace std;
using namespace libnormaliz;
```

have been declared. It is clear that opening these name spaces is dangerous. In this documentation we only do it to avoid constant repetition of std:: and libnormaliz::

D.1. Integer type as a template parameter

A cone can be constructed for two integer types, long long and mpz_class. (Also long is possible, but we disregard it in the following, since one should make sure that the integer type has at least 64 bits.) It is reasonable to choose mpz_class since the main computations will then be tried with long long and restarted with mpz_class if long long cannot store the results. This internal change of integer type is not possible if the cone is constructed for long long. (Nevertheless, the linear algebra routines can use mpz_class locally if intermediate results exceed long long; have a look into matrix.cpp.)

Internally the template parameter is called Integer. In the following we assume that the integer type has been fixed as follows:

```
typedef mpz_class Integer;
```

The internal passage from mpz_class to long long can be suppressed by

```
MyCone.deactivateChangeOfPrecision();
```

where we assume that MyCone has been constructed as described in the next section.

D.1.1. Alternative integer types

It is possible to use libnormaliz with other integer types than mpz_class or long long, but we have tested only these types.

If you want to use other types, you probably have to implement some conversion functions which you can find in integer.h and integer.cpp. Namely the functions

```
bool libnormaliz::try_convert(TypeA, TypeB);
            // converts TypeB to TypeA, returns false if not possible
```

where one type is your type and the other is long, long long, mpz_class and nmmz_float. Additionally, if your type uses infinite precision (for example, it is some wrapper for GMP), you must also implement

```
template<> inline bool libnormaliz::using_GMP<YourType>() { return true; }
```

If you want to compile libnormaliz in one compile unit: libnormaliz-all.cpp calls all source files..

D.1.2. Decimal fractions and floating point numbers

libnormaliz has a type `nmz_float` (presently set to `double`) that allows the construction of cones from floating point data point numbers. These are are first converted into `mpq_class` by using the GMP constructor of `mpq_class`, and then denomiantors are cleared. (The input routine of Normaliz goeas another way by reading the floating point input as decimal fractions.)

D.2. Construction of a cone

The construction requires the specification of input data consisting of one or more matrices and the input types they represent.

The term "matrix" stands for

```
vector<vector<number> >
```

where predefined choices of number are `long long`, `mpz_class`, `mpq_class` and `nmz_flot` (the latter representing `double`).

The available input types (from `libnormaliz.h`) are defined as follows:

```
namespace Type {
enum InputType {
//
// homogeneous generators
//
polytope,
rees_algebra,
subspace,
cone,
cone_and_lattice,
lattice,
saturation,
//
// inhomogeneous generators
//
vertices,
offset,
//
// homogeneous constraints
//
inequalities,
signs,
equations,
congruences,
//
// inhomogeneous constraints
//
inhom_equations,
```

```
inhom_inequalities,
strict_inequalities,
strict_signs,
inhom_congruences,
//
// linearforms
//
grading,
dehomogenization,
//
// special
open_facets,
projection_coordinates,
excluded_faces,
lattice_ideal,
//
// prwecomputed data
//
support_hyperplanes,
extreme_rays,
hilbert_basis_rec_cone,
//
// deprecated
//
integral_closure,
normalization,
polyhedron
};
} //end namespace Type
```

The input types are explained in Section 3. In certain environments it is not possible to use the enumeration. Therefore we provide a function that converts a string into the corresponding input type:

```
Type::InputType to_type(const string& type_string)
```

The types `grading`, `dehomogenization`, `offset`and `open_facets` must be encoded as matrices with a single row. We come back to this point below.

The simplest constructor has the syntax

```
Cone<Integer>::Cone(InputType input_type, const vector< vector<Integer> >& Input)
```

and can be used as in the following example:

```
vector<vector <Integer> > Data = ...
Type::InputType type = cone;
Cone<Integer> MyCone = Cone<Integer>(type, Data);
```

For two and three pairs of type and matrix there are the constructors

```
Cone<Integer>::Cone(InputType type1, const vector< vector<Integer> >& Input1,
                    InputType type2, const vector< vector<Integer> >& Input2)

Cone<Integer>::Cone(InputType type1, const vector< vector<Integer> >& Input1,
                    InputType type2, const vector< vector<Integer> >& Input2,
                    InputType type3, const vector< vector<Integer> >& Input3)
```

If you have to combine more than three matrices, you can define a

```
map <InputType, vector< vector<Integer> > >
```

and use the constructor with syntax

```
Cone<Integer>::Cone(const map< InputType,
                    vector< vector<Integer> > >& multi_input_data)
```

The four constructors also exist in a variant that uses the `libnormaliz` type `Matrix<Integer>` instead of `vector< vector<Integer> >` (see `cone.h`).

For the input of rational numbers we have all constructors also in variants that use `mpq_class` for the input matrix, for example

```
Cone<Integer>::Cone(InputType input_type, const vector< vector<mpq_class> >& Input)
```

etc.

Similarly, for the input of decimal fractions and floating point numbers we have all constructors also in variants that use `nmz_float` for the input matrix, for example

```
Cone<Integer>::Cone(InputType input_type, const vector< vector<nmz_float> >& Input)
```

etc.

For convenience we provide the function

```
vector<vector<T> > to_matrix<Integer>(vector<T> v)
```

in `matrix.h`. It returns a matrix whose first row is `v`. A typical example:

```
size_t dim = ...
vector<vector <Integer> > Data = ...
Type::InputType type = cone;
vector<Integer> total_degree(dim,1);
Type::InputType grad = grading;
Cone<Integer> MyCone = Cone<Integer>(type, Data,grad,to_matrix(total_degree));
```

There is a default constructor for cones,

```
Cone<Integer>::Cone()
```

D.2.1. Setting the grading

If your computation needs a grading, you should include it into the construction of the cone. However, especially in interactive use via PyNormaliz or other interfaces, it can be useful to add the grading if it was forgotten or to change it later on. The following function allows this:

```
void Cone<Integer>::resetGrading(const vector<Integer>& grading)
```

Note that it deletes all previously computed results that depend on the grading.

D.2.2. Setting the polynomial

The polynomial needed for integrals and weighted Ehrhart series must be passed to the cone after construction:

```
void Cone<Integer>::setPolynomial(string poly)
```

Like the grading ot can be changed later on. Then the results depending on the previous polynomial will be deleted.

D.2.3. Setting the expansion degree

This is done by

```
void Cone<Integer>::setExpansionDegree(long degree)
```

The default value of `degree` is -1, signaling 'no expansion". Also the expansion degree can be changed. Since the expansion is always computed on the fly, it does not delete previously computed results.

D.2.4. Setting the number of significant coefficients of the quasipolynomial

This is done by

```
void Cone<Integer>::setNrCoeffQuasiPol(long nr_coeff)
```

The default value of `nr_coeff` is -1, signaling 'all coefficients". If the number of significant coefficients is changed later on, the previously computed quesipolynomials will be deleted.

D.3. Computations in a cone

Before starting a computation in a (previously constructed) cone, one must decide what should be computed and in which way it should be computed. The computation goals and algorithmic variants (see Section 4) are defined as follows (`cone_property.h`):

```
namespace ConeProperty {
        enum Enum {
                FIRST_MATRIX,
```

```
Generators = ConeProperty::FIRST_MATRIX,
ExtremeRays,
VerticesOfPolyhedron,
SupportHyperplanes,
HilbertBasis,
ModuleGenerators,
Deg1Elements,
LatticePoints,
ModuleGeneratorsOverOriginalMonoid,
ExcludedFaces,
OriginalMonoidGenerators,
MaximalSubspace,
Equations,
Congruences,
LAST_MATRIX = ConeProperty::Congruences,
FIRST_MATRIX_FLOAT,
SuppHypsFloat = ConeProperty::FIRST_MATRIX_FLOAT,
VerticesFloat,
LAST_MATRIX_FLOAT = ConeProperty::VerticesFloat,
// Vector values
FIRST_VECTOR,
Grading = ConeProperty::FIRST_VECTOR,
Dehomogenization,
WitnessNotIntegrallyClosed,
GeneratorOfInterior,
ClassGroup,
LAST_VECTOR = ConeProperty::ClassGroup,
// Integer valued,
FIRST_INTEGER,
TriangulationDetSum = ConeProperty::FIRST_INTEGER,
ReesPrimaryMultiplicity,
GradingDenom,
UnitGroupIndex,
InternalIndex,
LAST_INTEGER = ConeProperty::InternalIndex,
FIRST_GMP_INTEGER,
ExternalIndex = FIRST_GMP_INTEGER,
LAST_GMP_INTEGER = ConeProperty::ExternalIndex,
// rational valued
FIRST_RATIONAL,
Multiplicity = ConeProperty::FIRST_RATIONAL,
Volume,
Integral,
VirtualMultiplicity,
LAST_RATIONAL = ConeProperty::VirtualMultiplicity,
// floating point valued
```

```
FIRST_FLOAT,
EuclideanVolume = ConeProperty::FIRST_FLOAT,
EuclideanIntegral,
LAST_FLOAT = ConeProperty::EuclideanIntegral,
// dimensions
FIRST_MACHINE_INTEGER,
TriangulationSize = ConeProperty::FIRST_MACHINE_INTEGER,
RecessionRank,
AffineDim,
ModuleRank,
Rank,
EmbeddingDim,
LAST_MACHINE_INTEGER = ConeProperty::EmbeddingDim,
// boolean valued
FIRST_BOOLEAN,
IsPointed = ConeProperty::FIRST_BOOLEAN,
IsDeg1ExtremeRays,
IsDeg1HilbertBasis,
IsIntegrallyClosed,
IsReesPrimary,
IsInhomogeneous,
IsGorenstein,
LAST_BOOLEAN = ConeProperty::IsGorenstein,
// complex structures
FIRST_COMPLEX_STRUCTURE,
Triangulation = ConeProperty::FIRST_COMPLEX_STRUCTURE,
StanleyDec,
InclusionExclusionData,
IntegerHull,
ProjectCone,
ConeDecomposition,
HilbertSeries,
HilbertQuasiPolynomial,
EhrhartSeries,
EhrhartQuasiPolynomial,
WeightedEhrhartSeries,
WeightedEhrhartQuasiPolynomial,
Sublattice,
LAST_COMPLEX_STRUCTURE = ConeProperty::Sublattice,
//
// integer type for computations
//
FIRST_PROPERTY,
BigInt = ConeProperty::FIRST_PROPERTY,
//
// algorithmic variants
```

```
                //
                DefaultMode,
                Approximate,
                BottomDecomposition,
                NoBottomDec,
                DualMode,
                PrimalMode,
                Projection,
                ProjectionFloat,
                NoProjection,
                Symmetrize,
                NoSymmetrization,
                NoSubdivision,
                NoNestedTri, // synonym for NoSubdivision
                KeepOrder,
                HSOP,
                NoPeriodBound,
                SCIP,
                NoLLL,
                NoRelax,
                Descent,
                NoDescent,
                NoGradingDenom,
                GradingIsPositive,
                //
                // checking properties of already computed data
                // (cannot be used as a computation goal)
                //
                IsTriangulationNested,
                IsTriangulationPartial,
                //
                // ONLY FOR INTERNAL CONTROL
                //
                ExplicitHilbertSeries,
                NakedDual,
                EnumSize,
                LAST_PROPERTY = ConeProperty::EnumSize //  ...
        }; //...
}
```

The class `ConeProperties` is based on this enumeration. Its instantiation are essentially boolean vectors that can be accessed via the names in the enumeration. Instantiations of the class are used to set computation goals and algorithmic variants and to check whether the goals have been reached. The distinction between computation goals and algorithmic variants is not completely strict. See Section 4 for implications between some `ConeProperties`.

There exist versions of `compute` for up to 3 cone properties:

```
ConeProperties Cone<Integer>::compute(ConeProperty::Enum cp)

ConeProperties Cone<Integer>::compute(ConeProperty::Enum cp1,
                    ConeProperty::Enum cp2)

ConeProperties Cone<Integer>::compute(ConeProperty::Enum cp1,
                    ConeProperty::Enum cp2, ConeProperty::Enum cp3)
```

An example:

```
MyCone.compute(ConeProperty::HilbertBasis, ConeProperty::Multiplicity)
```

If you want to specify more than 3 cone properties, you can define an instance of `ConeProperties` yourself and call

```
ConeProperties Cone<Integer>::compute(ConeProperties ToCompute)
```

An example:

```
ConeProperties Wanted;
Wanted.set(ConeProperty::Triangulation, ConeProperty::HilbertBasis);
MyCone.compute(Wanted);
```

All `get...` functions that are listed in the next section, try to compute the data asked for if they have not yet been computed. Unless you are interested a single result, we recommend to use `compute` since the data asked for can then be computed in a single run. For example, if the Hilbert basis and the multiplicity are wanted, then it would be a bad idea to call `getHilbertBasis` and `getMultiplicity` consecutively. More importantly, however, is nno choice of an algorithmic variant if you use `get...` without `compute` beforehand.

It is possible that a computation goal is unreachable. If this can be recognized from the input, a `BadInputException` will be thrown. If it cannot be recognized from the input, and `DefaultMode` is not set, then `compute()` will throw a `NotComputableException` so that `compute()` cannot r3eturn a value. In the presence of `DefaultMode`, the returned `ConeProperties` are those that have not been computed.

Please inspect `cone_property.cpp` for the full list of methods implemented in the class `ConeProperties`. Here we only mention the constructors

```
ConeProperties::ConeProperties(ConeProperty::Enum p1)

ConeProperties::ConeProperties(ConeProperty::Enum p1, ConeProperty::Enum p2)

ConeProperties::ConeProperties(ConeProperty::Enum p1, ConeProperty::Enum p2,
                               ConeProperty::Enum p3)
```

and the functions

```
ConeProperties& ConeProperties::set(ConeProperty::Enum p1, bool value)

ConeProperties& ConeProperties::set(ConeProperty::Enum p1, ConeProperty::Enum p2)
```

```
bool ConeProperties::test(ConeProperty::Enum Property) const
```

A string can be converted to a cone property and conversely:

```
ConeProperty::Enum toConeProperty(const string&)
const string& toString(ConeProperty::Enum)
```

D.4. Retrieving results

As remarked above, all `get...` functions that are listed below, try to compute the data asked for if they have not yet been computed. As also remarked above, it is often better to use `compute` first.

The functions that return a matrix encoded as `vector<vector<number> >` have variants that return a matrix encoded in the `libnormaliz` class `Matrix<number>`. These are not listed below; see `cone.h`.

Note that there are now functions that return results by type so that interfaces need not implement all the functions in this section. See D.4.21.

D.4.1. Checking computations

In order to check whether a computation goal has been reached, one can use

```
bool Cone<Integer>::isComputed(ConeProperty::Enum prop) const
```

for example

```
bool done=MyCone.isComputed(ConeProperty::HilbertBasis)
```

D.4.2. Rank, index and dimension

```
size_t Cone<Integer>::getEmbeddingDim()
size_t Cone<Integer>::getRank()
Integer Cone<Integer>::getInternalIndex()
Integer Cone<Integer>::getUnitGroupIndex()

size_t Cone<Integer>::getRecessionRank()
long Cone<Integer>::getAffineDim()
size_t Cone<Integer>::getModuleRank()
```

The *internal* index is only defined if original generators are defined. See Section D.4.15 for the external index.

The last three functions return values that are only well-defined after inhomogeneous computations.

D.4.3. Support hyperplanes and constraints

```
const vector< vector<Integer> >& Cone<Integer>::getSupportHyperplanes()
size_t Cone<Integer>::getNrSupportHyperplanes()
```

The first function returns the support hyperplanes of the (homogenized) cone. The second function returns the number of support hyperplanes.

Support hyperplanes can be returned in floating point format:

```
const vector< vector<nmz_float> >& Cone<Integer>::getSuppHypsFloat()
size_t Cone<Integer>::getNrSuppHypsFloat()
```

Together with the equations and congruences the support hyperplanes can also be accessed by

```
map< InputType , vector< vector<Integer> > > Cone<Integer>::getConstraints ()
```

The map returned contains three pairs whose keys are

```
Type::inequalities
Type::equations
Type::congruences
```

Note that `equations` and `congruences` can also be accessed via the coordinate transformation (to which they belong internally). See Section D.4.15.

D.4.4. Extreme rays and vertices

```
const vector< vector<Integer> >& Cone<Integer>::getExtremeRays()
size_t Cone<Integer>::getNrExtremeRays()
const vector< vector<Integer> >& Cone<Integer>::getVerticesOfPolyhedron()
size_t Cone<Integer>::getNrVerticesOfPolyhedron()
```

In the inhomogeneous case the first function returns the extreme rays of the recession cone, and the second the vertices of the polyhedron. (Together they form the extreme rays of the homogenized cone.)

Vertices can be returned in floating point format:

```
const vector< vector<nmz_float> >& Cone<Integer>::getVerticesFloat()
size_t Cone<Integer>::getNrVerticesFloat()
```

D.4.5. Generators

```
const vector< vector<Integer> >& Cone<Integer>::getOriginalMonoidGenerators()
size_t Cone<Integer>::getNrOriginalMonoidGenerators()
```

Note that original generators are not always defined. The system of generators of the cone that is used in the computations and its cardinality are returned by

```
const vector< vector<Integer> >& Cone<Integer>::getGenerators()
size_t Cone<Integer>::getNrGenerators()
```

D.4.6. Lattice points in polytopes and elements of degree 1

```
const vector< vector<Integer> >& Cone<Integer>::getDeg1Elements()
const vector< vector<Integer> >& Cone<Integer>::getLatticePoints()
size_t Cone<Integer>::getNrDeg1Elements()
```

This applies only to homogeneous computations. If a polytope is defined by inhomogeneous input, its lattice points appear as `ModuleGenerators`; see below.

D.4.7. Hilbert basis

In the nonpointed case we need the maximal linear subspace of the cone:

```
const vector< vector<Integer> >& Cone<Integer>::getMaximalSubspace()
size_t Cone<Integer>::getDimMaximalSubspace()
```

One of the prime results of Normaliz and its cardinality are returned by

```
const vector< vector<Integer> >& Cone<Integer>::getHilbertBasis()
size_t Cone<Integer>::getNrHilbertBasis()
```

Inhomogeneous case the functions refer to the the Hilbert basis of the recession cone. The module generators of the lattice points in the polyhedron are accessed by

```
const vector< vector<Integer> >& Cone<Integer>::getModuleGenerators()
size_t Cone<Integer>::getNrModuleGenerators()
```

If the original monoid is not integrally closed, you can ask for a witness:

```
vector<Integer> Cone<Integer>::getWitnessNotIntegrallyClosed()
```

D.4.8. Module generators over original monoid

```
const vector< vector<Integer> >&
               Cone<Integer>::getModuleGeneratorsOverOriginalMonoid()
size_t Cone<Integer>::getNrModuleGeneratorsOverOriginalMonoid()
```

D.4.9. Generator of the interior

If the monoid is Gorenstein, Normaliz computes the generator of the interior (the canonical module):

```
const vector<Integer>& Cone<Integer>::getGeneratorOfInterior()
```

Before asking for this vector, one should test `isGorenstein()`.

D.4.10. Grading and dehomogenization

```
vector<Integer> Cone<Integer>::getGrading()
Integer Cone<Integer>::getGradingDenom()
```

The second function returns the denominator of the grading.

```
vector<Integer> Cone<Integer>::getDehomogenization()
```

D.4.11. Enumerative data

```
mpq_class Cone<Integer>::getMultiplicity()
```

Don't forget that the multiplicity is measured for a rational, not necessarily integral polytope. Therefore it need not be an integer. The same applies to

```
mpq_class Cone<Integer>::getVolume()
nmz_float Cone<Integer>::getEuclideanVolume()
```

which can be computed for polytopes defined by homogeneous or inhomogeneous input. In the homogeneous case the volume is the multiplicity.

The Hilbert series is stored in a class of its own. It is retrieved by

```
const HilbertSeries& Cone<Integer>::getHilbertSeries()
```

It contains several data fields that can be accessed as follows (see `hilbert_series.h`):

```
const vector<mpz_class>& HilbertSeries::getNum() const;
const map<long, denom_t>& HilbertSeries::getDenom() const;

const vector<mpz_class>& HilbertSeries::getCyclotomicNum() const;
const map<long, denom_t>& HilbertSeries::getCyclotomicDenom() const;

const vector<mpz_class>& HilbertSeries::getHSOPNum() const;
const map<long, denom_t>& HilbertSeries::getHSOPDenom() const;

long HilbertSeries::getDegreeAsRationalFunction() const;
long HilbertSeries::getShift() const;

bool HilbertSeries::isHilbertQuasiPolynomialComputed() const;
vector< vector<mpz_class> > HilbertSeries::getHilbertQuasiPolynomial() const;
long HilbertSeries::getPeriod() const;
mpz_class HilbertSeries::getHilbertQuasiPolynomialDenom() const;

vector<mpz_class> HilbertSeries::getExpansion() const;
```

The first six functions refer to three representations of the Hilbert series as a rational function in the variable t: the first has a denominator that is a product of polynomials $(1-t^g)^e$, the second has a denominator that is a product of cylotomic polynomials. In the third case the denominator is determined by the degrees of a homogeneous system of parameters (see Section 2.5). In all cases the numerators are given by their coefficient vectors, and the denominators are lists of pairs (g,e) where in the second case g is the order of the cyclotomic polynomial.

If you have already computed the Hilbert series without HSOP and you want it with HSOP afterwards, the Hilbert series will simply be transformed, but Normaliz must compute the degrees for the denominator, and this may be a nontrivial computation.

The degree as a rational function is of course independent of the chosen representation, but may be negative, as well as the shift that indicates with which power of t the numerator tarts. Since the denominator has a nonzero constant term in all cases, this is exactly the smallest degree in which the Hilbert function has a nonzero value.

The Hilbert quasipolynomial is represented by a vector whose length is the period and whose entries are itself vectors that represent the coefficients of the individual polynomials corresponding to the residue classes modulo the period. These integers must be divided by the common denominator that is returned by the last function.

The computation goals `EhrhartSeries` and `EhrhartQuasiPolynomial` define special versions of the Hilbert series and the Hilbert quasipolynomial. Therefore the Ehrhart series and quasipolynomial are returned via the Hilbert series and quasipolynomial. Via `isComputed(ConeProperty::EhrhartSeries)` and `isComputed(ConeProperty::EhrhartQuasipolynomial)` one can find out whether these special versions were computed.

For the input type `rees_algebra` we provide

```
Integer Cone<Integer>::getReesPrimaryMultiplicity()
```

D.4.12. Weighted Ehrhart series and integrals

The wighted Ehrhart series can be accessed by

```
const pair<HilbertSeries, mpz_class>& Cone<Integer>::getWeightedEhrhartSeries()
```

The second component of the pair is the denominator of the coefficients in the series numerator. Its introduction was necessary since we wanted to keep integral coefficients for the numerator of a Hilbert series. The numerator and the denominator of the first component of type `HilbertSeries` can be accessed as usual, but one must not forget the denominator of the numerator coefficients. There is a second way to access these data; see below.

The virtual multiplicity and the integral, respectively, are got by

```
mpq_class Cone<Integer>::getVirtualMultiplicity()
mpq_class Cone<Integer>::getIntegral()
nmz_float  Cone<Integer>::getEuclideanIntegral()
```

Actually the cone saves these data in a special container of class `IntegrationData` (defined in `Hilbert_sries.h`). It is accessed by

```
IntegrationData& Cone<Integer>::getIntData()
```

The three `get` functions above are only shortcuts for the access via `getIntData()`:

```
string IntegrationData::getPolynomial() const
long IntegrationData::getDegreeOfPolynomial() const
bool IntegrationData::isPolynomialHomogeneous() const

const vector<mpz_class>& IntegrationData::getNum_ZZ() const
mpz_class IntegrationData::getNumeratorCommonDenom() const
const map<long, denom_t>& IntegrationData::getDenom() const

const vector<mpz_class>& IntegrationData::getCyclotomicNum_ZZ() const
const map<long, denom_t>& IntegrationData::getCyclotomicDenom() const

bool IntegrationData::isWeightedEhrhartQuasiPolynomialComputed() const
void IntegrationData::computeWeightedEhrhartQuasiPolynomial()
vector< vector<mpz_class> > IntegrationData::getWeightedEhrhartQuasiPolynomial()
mpz_class IntegrationData::getWeightedEhrhartQuasiPolynomialDenom() const

vector<mpz_class> IntegrationData::getExpansion() const

mpq_class IntegrationData::getVirtualMultiplicity() const
mpq_class IntegrationData::getIntegral() const
```

The first three functions refer to the polynomial defining the integral or weighted Ehrhart series.

The computation of these data is controlled by the corresponding `ConeProperty`. The expansion is always computed on-the-fly.

D.4.13. Triangulation and disjoint decomposition

The triangulation, the size and the sum of the determinants are returned by

```
const vector< pair<vector<key_t>,Integer> >& Cone<Integer>::getTriangulation()
size_t Cone<Integer>::getTriangulationSize()
Integer Cone<Integer>::getTriangulationDetSum()
```

See Section 6.15 for the interpretation of these data. The first component of the pair is the vector of indices of the simplicial cones in the triangulation. Note that the indices are here counted from 0 (whereas they start from 1 in the `tri` file). The second component is the determinant.

The type of triangulation can be retrieved by

```
bool Cone<Integer>::isTriangulationNested()
bool Cone<Integer>::isTriangulationPartial()
```

If the disjoint decomposition has been computed, one gets the $0/1$ vectors describing the facets to be removed

```
const vector<vector<bool> >& Cone<Integer>::getOpenFacets()
```

D.4.14. Stanley decomposition

The Stanley decomposition is stored in a list whose entries correspond to the simplicial cones in the triangulation:

```
const list< STANLEYDATA<Integer> >& Cone<Integer>::getStanleyDec()
```

Each entry is a record of type `STANLEYDATA` defined as follows:

```
struct STANLEYDATA {
vector<key_t> key;
Matrix<Integer> offsets;
};
```

The key has the same interpretation as for the triangulation, namely as the vector of indices of the generators of the simplicial cone (counted from 0). The matrix contains the coordinate vectors of the offsets of the components of the decomposition that belong to the simplicial cone defined by the key. See Section 6.16 for the interpretation. The format of the matrix can be accessed by the following functions of class `Matrix<Integer>`:

```
    size_t nr_of_rows() const
    size_t nr_of_columns() const
```

The entries are accessed in the same way as those of `vector<vector<Integer> >`.

D.4.15. Coordinate transformation

The coordinate transformation from the ambient lattice to the sublattice generated by the Hilbert basis and the basis of the maximal subspace can be returned as follows:

```
const Sublattice_Representation<Integer>& Cone<Integer>::getSublattice()
```

An object of type `Sublattice_Representation` models a sequence of $\mathbb{Z}$-homomorphisms

$$\mathbb{Z}^r \xrightarrow{\varphi} \mathbb{Z}^n \xrightarrow{\pi} \mathbb{Z}^r$$

with the following property: there exists $c \in \mathbb{Z}$, $c \neq 0$, such that $\pi \circ \varphi = c \cdot \mathrm{id}_{\mathbb{Z}^r}$. In particular φ is injective. One should view the two maps as a pair of coordinate transformations: φ is determined by a choice of basis in the sublattice $U = \varphi(\mathbb{Z}^r)$, and it allows us to transfer vectors from $U \cong \mathbb{Z}^r$ to the ambient lattice $\mathbb{Z}^n$. The map π is used to realize vectors from U as linear combinations of the given basis of $U \cong \mathbb{Z}^r$: after the application of π one divides by c. (If U is a direct summand of $\mathbb{Z}^n$, one can choose $c = 1$, and conversely.) Normaliz considers vectors as rows of matrices. Therefore φ is given as an $r \times n$-matrix and π is given as an $n \times r$ matrix.

The data just described can be accessed as follows (`sublattice_representation.h`). For space reasons we omit the class specification `Sublattice_Representation<Integer>::`

```
const vector<vector<Integer> >& getEmbedding() const
const vector<vector<Integer> >& getProjection() const
Integer getAnnihilator() const
```

Here "Embedding" refers to φ and "Projection" to π (though π is not always surjective). The "Annihilator" is the number c above. (It annihilates $\mathbb{Z}^r$ modulo $\pi(\mathbb{Z}^n)$.)

The numbers n and r are accessed in this order by

```
size_t getDim() const
size_t getRank() const
```

The external index, namely the order of the torsion subgroup of $\mathbb{Z}^n/U$, is returned by

```
mpz_class getExternalIndex() const
```

Very often φ and ψ are identity maps, and this property can be tested by

```
bool IsIdentity()const
```

The constraints computed by Normaliz are "hidden" in the sublattice representation. They van be accessed by

```
const vector<vector<Integer> >& getEquations() const
const vector<vector<Integer> >& getCongruences() const
```

But see Section D.4.3 above for a more direct access.

D.4.16. Class group

```
vector<Integer> Cone<Integer>::getClassGroup()
```

The return value is to be interpreted as follows: The entry for index 0 is the rank of the class group. The remaining entries contain the orders of the summands in a direct sum decomposition of the torsion subgroup.

D.4.17. Integer hull

For the computation of the integer hull an auxiliary cone is constructed. A reference to it is returned by

```
Cone<Integer>& Cone<Integer>::getIntegerHullCone() const
```

For example, the support hyperplanes of the integer hull can be accessed by

```
MyCone.getIntegerHullCone().getSupportHyperplanes()
```

D.4.18. Projection of the cone

Like the integer hull, the image of the projection is contained in an auxiliary cone that can bea aceesed by

```
Cone<Integer>& Cone<Integer>::getProjectCone() const
```

It contains constraints and extreme rays of the projection.

D.4.19. Excluded faces

Before using the excluded faces Normaliz makes the collection irredundant by discarding those that are contained in others. The irredundant collection (given by hyperplanes that intersect the cone in the faces) and its cardinality are returned by

```
const vector< vector<Integer> >& Cone<Integer>::getExcludedFaces()
size_t Cone<Integer>::getNrExcludedFaces()
```

For the computation of the Hilbert series the all intersections of the excluded faces are computed, and for each resulting face the weight with which it must be counted is computed. These data can be accessed by

```
const vector< pair<vector<key_t>,long> >&
                            Cone<Integer>::getInclusionExclusionData()
```

The first component of each pair contains the indices of the generators (counted from 0) that lie in the face and the second component is the weight.

D.4.20. Boolean valued results

All the "questions" to the cone that can be asked by the boolean valued functions in this section start a computation if the answer is not yet known.

The first, the question

```
bool Cone<Integer>::isIntegrallyClosed()
```

does not trigger a computation of the full Hilbert basis. The computation stops as soon as the answer can be given, and this is the case when an element in the integral closure has been found that is not in the original monoid. Such an element is retrieved by

```
vector<Integer> Cone<Integer>::getWitnessNotIntegrallyClosed()
```

As discussed in Section 6.14.3 it can sometimes be useful to ask

```
bool Cone<Integer>::isPointed()
```

before a more complex computation is started.

The Gorenstein property can be tested with

```
bool Cone<Integer>::isGorenstein()
```

If the answer is positive, Normaliz computes the generator of the interior of the monoid. Also see D.4.9

The next two functions answer the question whether the Hilbert basis or at least the extreme rays live in degree 1.

```
bool Cone<Integer>::isDeg1ExtremeRays()
bool Cone<Integer>::isDeg1HilbertBasis()
```

Finally we have

```
bool Cone<Integer>::isInhomogeneous()
bool Cone<Integer>::isReesPrimary()
```

`isReesPrimary()` checks whether the ideal defining the Rees algebra is primary to the irrelevant maximal ideal.

D.4.21. Results by type

It is also possible to access (and compute if necessary) the output data of Normalz by functions that only depend on the C++ type of the data:

```
const Matrix<Integer>& getMatrixConePropertyMatrix(ConeProperty::Enum property);
const vector< vector<Integer> >& getMatrixConeProperty(ConeProperty::Enum property);
const Matrix<nmz_float>& getFloatMatrixConePropertyMatrix(ConeProperty::Enum property);
const vector< vector<nmz_float> >& getFloatMatrixConeProperty(ConeProperty::Enum property);
vector<Integer> getVectorConeProperty(ConeProperty::Enum property);
Integer getIntegerConeProperty(ConeProperty::Enum property);
mpz_class getGMPIntegerConeProperty(ConeProperty::Enum property);
mpq_class getRationalConeProperty(ConeProperty::Enum property);
nmz_float getFloatConeProperty(ConeProperty::Enum property);
size_t getMachineIntegerConeProperty(ConeProperty::Enum property);
bool getBooleanConeProperty(ConeProperty::Enum property);
```

For example, `getMatrixConeProperty(ConeProperty::HilbertBasis)` will return the Hilbert basis as a `const vector< vector<Integer> >&`.

These functions make it easier to write interfaces to Normaliz since they need not to introduce new functions for results that have one of the types listed above.

It is clear that the complex results can only be accessed via their specialized "get" functions.

D.5. Control of execution

D.5.1. Exceptions

All exceptions that are thrown in `libnormaliz` are derived from the abstract class `NormalizException` that itself is derived from `std::exception`:

```
class NormalizException: public std::exception
```

The following exceptions must be caught by the calling program:

```
class ArithmeticException: public NormalizException
class BadInputException: public NormalizException
class NotComputableException: public NormalizException
class FatalException: public NormalizException
class NmzCoCoAException: public NormalizException
class InterruptException: public NormalizException
```

The `ArithmeticException` leaves `libnormaliz` if a nonrecoverable overflow occurs (it is also used internally for the change of integer type). This should not happen for cones of integer type `mpz_class`, unless it is caused by the attempt to create a data structure of illegal size or by a bug in the program. The `BadInputException` is thrown whenever the input is inconsistent; the reasons for this are manifold. The `NotComputableException` is thrown if a computation goal cannot be reached. The `FatalException` should never appear. It covers error situations that can only be caused by a bug in the program. At many places `libnormaliz` has `assert` verifications built in that serve the same purpose.

There are two more exceptions for the communication within `libnormaliz` that should not leave it:

```
class NonpointedException: public NormalizException
class NotIntegrallyClosedException: public NormalizException
```

The `InterruptException` is discussed in the next section.

D.5.2. Interruption

In order to find out if the user wants to interrupt the program, the functions in `libnormaliz` test the value of the global variable

```
volatile sig_atomic_t nmz_interrupted
```

If it is found to be `true`, an `InterruptException` is thrown. This interrupt lraves `libnormaliz`, so that the calling program can process it. The `Cone` still exists, and the data computed in it can still be accessed. Moreover, `compute` can again be applied to it.

The calling program must take care to catch the signal caused by Ctrl-C and to set `nmz_interrupted=1`.

D.5.3. Inner parallelization

By default the cone constructor sets the maximal number of parallel threads to 8, unless the system has set a lower limit. You can change this value by

```
long set_thread_limit(long t)
```

The function returns the previous value.

`set_thread_limit(0)` raises the limit set by libnormaliz to ∞.

D.5.4. Outer parallelization

The libnormaliz functions can be called by programs that are parallelized via OpenMP themselves. The functions in libnormaliz switch off nested parallelization.

As a test program you can compile and run `outerpar` in `source/outerpar`. Compile it by `make -f Makefile.classic`.

D.5.5. Control of terminal output

By using

```
bool setVerboseDefault(bool v)
```

one can control the verbose output of `libnormaliz`. The default value is `false`. This is a global setting that effects all cones constructed afterwards. However, for every cone one can set an individual value of `verbose` by

```
bool Cone<Integer>::setVerbose(bool v)
```

Both functions return the previous value.

The default values of verbose output and error output are `std::cout` and `std::cerr`. These values can be changed by

```
void setVerboseOutput(std::ostream&)
void setErrorOutput(std::ostream&)
```

D.6. A simple program

The example program is a simplified version of the program on which the experiments for the paper "Quantum jumps of normal polytopes" by W. Bruns, J. Gubeladze and M. Michałek, Discrete Comput. Geom. 56 (2016), no. 1, 181–215, are based. Its goal is to find a maximal normal lattice polytope P in the following sense: there is no normal lattice polytope $Q \supset P$ that has exactly one more lattice point than P. 'Normal" means in this context that the Hilbert basis of the cone over P is given by the lattice points of P, considered as degree 1 elements in the cone.

The program generates normal lattice simplices and checks them for maximality. The dimension is set in the program, as well as the bound for the random coordinates of the vertices.

Let us have a look at `source/maxsimplex/maxsimplex.cpp`. First the more or less standard preamble:

```
#include <stdlib.h>
#include <vector>
#include <fstream>
#include <omp.h>
using namespace std;

#include "libnormaliz/libnormaliz.h"
#include "libnormaliz/cone.h"
```

```
#include "libnormaliz/vector_operations.h"
#include "libnormaliz/cone_property.h"
#include "libnormaliz/integer.h"
using namespace libnormaliz;
```

Since we want to perform a high speed experiment which is not expected to be arithmetically demanding, we choose 64 bit integers:

```
typedef long long Integer;
```

The first routine finds a random normal simplex of dimension `dim`. The coordinates of the vertices are integers between 0 and `bound`. We are optimistic that such a simplex can be found, and this is indeed no problem in dimension 4 or 5.

```
Cone<Integer> rand_simplex(size_t dim, long bound){

    vector<vector<Integer> > vertices(dim+1,vector<Integer> (dim));
    while(true){ // an eternal loop ...
        for(size_t i=0;i<=dim;++i){
            for(size_t j=0;j<dim;++j)
                vertices[i][j]=rand()%(bound+1);
        }

        Cone<Integer> Simplex(Type::polytope,vertices);
        // we must check the rank and normality
        if(Simplex.getRank()==dim+1 && Simplex.isDeg1HilbertBasis())
            return Simplex;
    }
    vector<vector<Integer> > dummy_gen(1,vector<Integer>(1,1));
    // to make the compiler happy
    return Cone<Integer>(Type::cone,dummy_gen);
}
```

We are looking for a normal polytope $Q \supset P$ with exactly one more lattice point. The potential extra lattice points z are contained in the matrix `jump_cands`. There are two obstructions for $Q = \mathrm{conv}(P,z)$ to be tested: (i) z is the only extra lattice point and (ii) Q is normal. It makes sense to test them in this order since most of the time condition (i) is already violated and it is much faster to test.

```
bool exists_jump_over(Cone<Integer>& Polytope,
                      const vector<vector<Integer> >& jump_cands){

    vector<vector<Integer> > test_polytope=Polytope.getExtremeRays();
    test_polytope.resize(test_polytope.size()+1);
    for(size_t i=0;i<jump_cands.size();++i){
        test_polytope[test_polytope.size()-1]=jump_cands[i];
        Cone<Integer> TestCone(Type::cone,test_polytope);
        if(TestCone.getNrDeg1Elements()!=Polytope.getNrDeg1Elements()+1)
            continue;
```

```
        if(TestCone.isDeg1HilbertBasis())
           return true;
    }
    return false;
}
```

In order to make the (final) list of candidates z as above we must compute the widths of P over its support hyperplanes.

```
vector<Integer> lattice_widths(Cone<Integer>& Polytope){

    if(!Polytope.isDeg1ExtremeRays()){
        cerr<< "Cone in lattice_widths is not defined by lattice polytope"<< endl;
        exit(1);
    }
    vector<Integer> widths(Polytope.getNrExtremeRays(),0);
    for(size_t i=0;i<Polytope.getNrSupportHyperplanes();++i){
        for(size_t j=0;j<Polytope.getNrExtremeRays();++j){
            // v_scalar_product is a useful function from vector_operations.h
            Integer test=v_scalar_product(Polytope.getSupportHyperplanes()[i],
                                          Polytope.getExtremeRays()[j]);
            if(test>widths[i])
                widths[i]=test;
        }
    }
    return widths;
}
```

```
int main(int argc, char* argv[]){

    time_t ticks;
    srand(time(&ticks));
    cout << "Seed " <<ticks << endl;  // we may want to reproduce the run

    size_t polytope_dim=4;
    size_t cone_dim=polytope_dim+1;
    long bound=6;
    vector<Integer> grading(cone_dim,0);
           // at some points we need the explicit grading
    grading[polytope_dim]=1;

    size_t nr_simplex=0; // for the progress report
```

Since the computations are rather small, we suppress parallelization (except for one step below).

```
    while(true){
```

```
#ifdef _OPENMP
        omp_set_num_threads(1);
#endi
    Cone<Integer> Candidate=rand_simplex(polytope_dim,bound);
    nr_simplex++;
    if(nr_simplex%1000 ==0)
        cout << "simplex " << nr_simplex << endl;
```

Maximality is tested in 3 steps. Most often there exists a lattice point z of height 1 over P. If so, then $\mathrm{conv}(P,z)$ contains only z as an extra lattice point and it is automatically normal. In order to find such a point we must move the support hyperplanes outward by lattice distance 1.

```
    vector<vector<Integer> > supp_hyps_moved=Candidate.getSupportHyperplanes();
    for(size_t i=0;i<supp_hyps_moved.size();++i)
        supp_hyps_moved[i][polytope_dim]+=1;
    Cone<Integer> Candidate1(Type::inequalities,supp_hyps_moved,
                                    Type::grading,to_matrix(grading));
    if(Candidate1.getNrDeg1Elements()>Candidate.getNrDeg1Elements())
        continue;                      // there exists a point of height 1
```

Among the polytopes that have survived the height 1 test, most nevertheless have suitable points z close to them, and it makes sense not to use the maximum possible height immediately. Note that we must now test normality explicitly.

```
    cout << "No ht 1 jump"<< " #latt " << Candidate.getNrDeg1Elements() << endl;
    // move the hyperplanes further outward
    for(size_t i=0;i<supp_hyps_moved.size();++i)
        supp_hyps_moved[i][polytope_dim]+=polytope_dim;
    Cone<Integer> Candidate2(Type::inequalities,supp_hyps_moved,
                             Type::grading,to_matrix(grading));
    cout << "Testing " << Candidate2.getNrDeg1Elements()
         << " jump candidates" << endl; // including the lattice points in P
    if(exists_jump_over(Candidate,Candidate2.getDeg1Elements()))
        continue;
```

Now we can be optimistic that a maximal polytope P has been found, and we test all candidates z that satisfy the maximum possible bound on their lattice distance to P.

```
    cout << "No ht <= 1+dim jump" << endl;
    vector<Integer> widths=lattice_widths(Candidate);
    for(size_t i=0;i<supp_hyps_moved.size();++i)
        supp_hyps_moved[i][polytope_dim]+=
                        -polytope_dim+(widths[i])*(polytope_dim-2);
```

The computation may become arithmetically critical at this point. Therefore we use `mpz_class` for our cone. The conversion to and from `mpz_class` is done by routines contained in `convert.h`.

```
    vector<vector<mpz_class> > mpz_supp_hyps;
```

```
    convert(mpz_supp_hyps,supp_hyps_moved);
    vector<mpz_class> mpz_grading=convertTo<vector<mpz_class> >(grading);
```

The computation may need some time now. Therefore we allow a little bit of parallelization.

```
#ifdef _OPENMP
        omp_set_num_threads(4);
#endif
```

Since P doesn't have many vertices (even if we use these routines for more general polytopes than simplices), we don't expect too many vertices for the enlarged polytope. In this situation it makes sense to set the algorithmic variant `Approximate`.

```
    Cone<mpz_class> Candidate3(Type::inequalities,mpz_supp_hyps,
                               Type::grading,to_matrix(mpz_grading));
    Candidate3.compute(ConeProperty::Deg1Elements,ConeProperty::Approximate);
    vector<vector<Integer> > jumps_cand; // for conversion from mpz_class
    convert(jumps_cand,Candidate3.getDeg1Elements());
    cout << "Testing " << jumps_cand.size() << " jump candidates" << endl;
    if(exists_jump_over(Candidate, jumps_cand))
        continue;
```

Success!

```
    cout << "Maximal simplex found" << endl;
    cout << "Vertices" << endl;
    Candidate.getExtremeRaysMatrix().pretty_print(cout); // a goody from matrix.h
    cout << "Number of lattice points = " << Candidate.getNrDeg1Elements();
    cout << " Multiplicity = " << Candidate.getMultiplicity() << endl;

    } // end while
} // end main
```

For the compilation of `maxsimplex.cpp` we have added a `Makefile`. Running the program needs a little bit of patience. However, within a few hours a maximal simplex should have emerged. From a log file:

```
simplex 143000
No ht 1 jump #latt 9
Testing 22 jump candidates
No ht 1 jump #latt 10
Testing 30 jump candidates
No ht 1 jump #latt 29
Testing 39 jump candidates
No ht <= 1+dim jump
Testing 173339 jump candidates
Maximal simplex found
Vertices
1 3 5 3 1
2 3 0 3 1
```

```
3 0 5 5 1
5 2 2 1 1
6 5 6 2 1
Number of lattice points = 29 Multiplicity = 275
```

E. PyNormaliz

The PyNormaliz install script assumes that you have executed the

`install_normaliz_with_qnormaliz_eantic.sh`

script. To install PyNormaliz navigate to the Normaliz directory and type

```
./install_pynormaliz.sh --user
```

The script detects your python version, assuming the executable is in the `PATH`. If you want to install PyNormaliz system-wide, replace `--user` by `--sudo`. Then you will be asked for your root password. The following additional options are available for `install_pynormaliz.sh`:

- `--python2 <path>`: Path to a python2 executable.
- `--python3 <path>`: Path to a python3 executable.
- `--prefix <path>`: Path to the Normaliz install path

Depending on your setup, you might be able to install PyNormaliz via pip, typing

```
pip install PyNormaliz
```

at a command prompt. Depending on your python version you might want to write `pip3` instead.

For a brief introduction please consult the PyNormaliz tutorial at `https://nbviewer.jupyter.org/github/Normaliz/PyNormaliz/blob/master/doc/PyNormaliz_Tutorial.ipynb`.

You can also open the tutorial for PyNormaliz interactively on `https://mybinder.org` following the link `https://mybinder.org/v2/gh/Normaliz/NormalizJupyter/master`.

F. QNormaliz

The variant QNormaliz of Normaliz can use coefficients from real algebraic extensions of $\mathbb{Q}$. It is clear that the computations are then restricted to those that do not depend on finite generation of monoids. At present, QNormaliz is set up for two different types of fields:

(1) real algebraic extensions of $\mathbb{Q}$ via the the type `renf_elem_class` implemented by V. Delecroix and M. Köppe,

(2) $\mathbb{Q}$ itself via the number type `mpq_class` – this is essentially superfluous, but served as a number field type in the development.

We describe QNormaliz as implemented with `renf_elem_class` since the version for $\mathbb{Q}$ can be considered as a special case of it.

QNormaliz is based on a number of packages:

(1) Flint maintained by B. Hart (`http://www.flintlib.org/`),
(2) antic by B. Hart and F. Johansson, `https://github.com/wbhart/antic`),
(3) arb by F. Johansson (`http://arblib.org/`),
(4) e-antic by V. Delecroix and M. Köppe (`https://github.com/videlec/e-antic`).

F.1. Input

The following input types are NOT allowed in QNormaliz:

`lattice`	`projection_coordinates`	`offset`	`excluded_faces`
`cone_and_lattice`	`inhom_congruences`	`lattice_ideal`	`open_facets`
`congruences`	`hilbert_basis_rec_cone`		

The only other restriction is that decimal fractions and floating point numbers are not allowed in the input file.

The input format for field coefficients is explained below when we discuss an example.

It may seem contradictory, but `saturation` is allowed. It must be interpreted as a generating set for a subspace that is intersected with all the objects defined by other input items.

QNormaliz does not look for an implicit grading, but can use an explicit grading for lattice point or volume computations in the homogeneous case.

F.2. Computations

The only computation goals allowed are:

`Generators`	`ExtremeRays`	`VerticesOfPolyhedron`	`MaximalSubspace`
`SupportHyperplanes`	`Equations`	`Triangulation`	`ConeDecomposition`
`Dehomogenization`	`Rank`	`EmbeddingDim`	`Sublattice`
`KeepOrder`	`IsPointed`	`IsInhomogeneous`	`DefaultMode`
`Volume`	`Deg1Elements`	`ModuleGenerators`	`TriangulationDetSum`
`TriangulationSize`	`IntwegerHull`		

`Deg1Elements`, `ModuleGenerators` and `IntegerHull` are restricted to polytopes since polyhedra in general lack the necessary finiteness properties. The lattice of reference is the full integral lattice.

`Volume` is restricted to full-dimensional polytopes.

It may seem paradoxical that `Sublattice` appears here. As in the true lattice case, the `Sublattice Representation` is the coordinate transformation used by QNormaliz. Over a field F there is

no need for the annihilator c, and one simply has a pair of linear maps $F^r \to F^d \to F^r$ whose composition is the identity of F^r. Of course, congruences and external index make no sense anymore.

Moreover, the original monoid and any data referring to it are not defined.

Implicit or explicit `DefaultMode` is interpreted as `SupportHyperplanes`.

QNormaliz has no algorithmic variants.

F.3. An example

The examples for QNormaliz are contained in the directory `Qexample`.

The icosahedron, one of the platonic solids, needs $\sqrt{5}$ for its coordinates. Via its vertices it can be defined as follows (`icosahedron-v.in`)

```
amb_space 3
number_field min_poly (a^2 - 5) embedding [2 +/- 1]
vertices 12
0 2 (a + 1) 4
0 -2 (a + 1) 4
2 (a + 1) 0 4
...
(-a - 1) 0 -2 4
Volume
ModuleGenerators
```

The second line specifies the extension $\mathbb{Q}[\sqrt{5}]$ of $\mathbb{Q}$ over which we want to define the icosahedron. In addition to the minimal polynomial we have to specify an interval from which the zero of the polynomial is to be picked. There must be a *single* zero in that interval. The name of the root is fixed to be `a`. The number field specification must follow `amb_space`. Otherweise QNormaliz believes that you want to work over $\mathbb{Q}$. This is allowed, but then all input numbers must be rational.

Note that the entries of the input file that contain `a` must be enclosed in round brackets. You can enter any $\mathbb{Q}$-linear combination of powers of `a`, provided the exponent is smaller than the degree of the minimal polynomial. We allow $*$ between the coefficient and the power of `a` and the sign `^` to indicate the exponent, but they need not appear. So

```
(a^3-2*a2  +   4a-1/2)
(a+a-2a-10 + 10*a^0)
```

are legal matrix entries, provided the minimal polynomial has degree ≥ 4.

The result of the computation by `Qnormaliz -c ../Qexample/icosahedron-v` starts

```
Real embedded number field:
min_poly (a^2 - 5) embedding [2.23606797749978969640917366873127623 54 +/- 5.73e-38]
```

It indicates that the precision to which the root had to be computed in order to decide all the inequalities that have come up in the computation. Then we go on as usual:

```
12 vertices of polyhedron
0 extreme rays of recession cone
20 support hyperplanes of polyhedron (homogenized)

embedding dimension = 4
affine dimension of the polyhedron = 3 (maximal)
rank of recession cone = 0

size of triangulation   = 18
resulting sum of |det|s = (5/2*a+15/2 ~ 13.090)

dehomogenization:
0 0 0 1

volume (lattice normalized) = (5/2*a+15/2 ~ 13.090)
volume (Euclidean) = 2.18167
```

From the vertices below you can compute the radius of the sphere in which the icosahedron is inscribed and check that it is < 1. So no surprise:

```
1 lattice points in polytope:
0 0 0 1

12 vertices of polyhedron:
(-1/4*a-1/4 ~ -0.80902)                      0        (-1/2 = -0.5000) 1
...
  (1/4*a+1/4 ~ 0.80902)                      0        (-1/2 = -0.5000) 1
  (1/4*a+1/4 ~ 0.80902)                      0          (1/2 = 0.5000) 1

0 extreme rays of recession cone:

20 support hyperplanes of polyhedron (homogenized):
   (-a+1 ~ -1.2361) (-2*a+4 ~ -0.47214)                  0 1
...
     (a-1 ~ 1.2361) (-2*a+4 ~ -0.47214)                  0 1
     (a-1 ~ 1.2361)   (2*a-4 ~ 0.47214)                  0 1
```

Now every nonintegral number appears in round brackets together with its approximation as a decimal fraction. (The sign ~ turns into = if a rational number has an exact representation as a decimal fraction.)

As in the "ordinary" Normaliz, the data of the integer hull cone are printed into a separate file, following the same name convention.

The matrices in the (optional) output file(s) can be used as input; see `perm7_d2_dual.in`. The input routine skips all characters from ~ or = when it reads a number.

F.4. libQnormaliz

The template for the class of numbers with which QNormaliz computes is `Number`. QNormaliz has two predefined specializations, namely `mpq_class` and `renf_elem_class`. The latter is defined in e-antic. (See `Qsource/libQnormaliz/libQnormaliz-templated.cpp`.)

libQnormaliz uses the same function names as libnormaliz, as far as the return values make sense. See `Qsource/libQnormaliz/Qcone.cpp`. Most functions return data of type `Number`.

The number field must be defined outside of libQnormaliz. The `renf_elem_class` numbers contain a pointer to an instance of `renf_class`, the real embedded numebr field. They transport it into libQnormaliz. Have a lok at `Qsource/Qnormaliz.cpp` that owns the number field for QNormaliz and to `Qsource/Qinput.in` where it is assignd to the input numbers. `NULL` indicates that the number is defined over $\mathbb{Q}$.

The integer hull cone is of type `libQnormaliz::Cone<Number>`.

F.5. Docker image

The Docker image

```
normaliz/normaliz
```

contains a full installation of QNormaliz.

F.6. Source download

There is no separate download for QNormaliz (anymore). It is part of the standard distribution.

F.7. Installation

QNormaliz should be installed using the script

```
install_normaliz_with_qnormaliz_eantic.sh
```

It installs all the packages used by Normaliz and QNormaliz, and builds both. Note that libQnormaliz wants access to libnormaliz. Also see Section 10.

F.8. PyQNormaliz

To install PyQNormaliz, use the `install_pynormaliz.sh` script from E.

Depending on your setup, you might be able to install PyQNormaliz via pip, typing

```
pip install PyQNormaliz
```

at a command prompt. Depending on your python version you might want to write `pip3` instead.

For a brief introduction please have a look at the PyQNormaliz example at `https://nbviewer.jupyter.org/github/Normaliz/PyQNormaliz/blob/master/examples/Dodecahedron.ipynb`.

You can also open the example for PyQNormaliz interactively on `https://mybinder.org` following the link `https://mybinder.org/v2/gh/Normaliz/NormalizJupyter/master`.

References

[1] J. Abbott, A.M. Bigatti and G. Lagorio, *CoCoA-5: a system for doing Computations in Commutative Algebra.* Available at `http://cocoa.dima.unige.it`.

[2] T. Achterberg. *SCIP: Solving constraint integer programs*. Mathematical Programming Computation 1 (2009), 1–41. Available from `http://mpc.zib.de/index.php/MPC/article/view/4`

[3] V. Almendra and B. Ichim. *jNormaliz 1.7*. Available at `http://normaliz.uos.de`

[4] V. Baldoni, N. Berline, J.A. De Loera, B. Dutra, M. Kï¿½ppe, S. Moreinis, G. Pinto, M. Vergne, J. Wu, *A User's Guide for LattE integrale v1.7.2, 2013.* Software package LattE is available at `http://www.math.ucdavis.edu/~latte/`

[5] W. Bruns and J. Gubeladze. *Polytopes, rings, and K-theory*. Springer 2009.

[6] W. Bruns and B. Ichim. *Normaliz: algorithms for rational cones and affine monoids.* J. Algebra 324 (2010) 1098–1113.

[7] W. Bruns, R. Hemmecke, B. Ichim, M. Kï¿½ppe and C. Sï¿½ger. *Challenging computations of Hilbert bases of cones associated with algebraic statistics* . Exp. Math.20 (2011), 25–33.

[8] W. Bruns, B. Ichim and C. Sï¿½ger. *The power of pyramid decomposition in Normaliz*. J. Symb. Comp. 74 (2016), 513–536.

[9] W .Bruns and R. Koch. *Computing the integral closure of an affine semigroup*. Univ. Iagell. Acta Math. 39 (2001), 59–70.

[10] W. Bruns, R. Sieg and C. Sï¿½ger. *Normaliz 2013–2016.* To appear in the final report of the DFG SPP 1489. Preprint `arXiv:1611.07965`.

[11] W. Bruns and C. Sï¿½ger. *The computation of weighted Ehrhart series in Normaliz.* J. Symb. Comp. 68 (2015), 75–86.

[12] S. Gutsche, M. Horn, C. Sï¿½ger, *NormalizInterface for GAP*. Available at `https://github.com/gap-packages/NormalizInterface`.

[13] S. Gutsche, R. Sieg, *PyNormaliz - an interface to Normaliz from python* Available at `https://github.com/Normaliz/PyNormaliz`.

[14] S. Gutsche, *PyQNormaliz - an interface to QNormaliz from python* Available at `https://github.com/Normaliz/PyQNormaliz`.

[15] W. Hart, F. Johansson and S. Pancratz, *FLINT: Fast Library for Number Theory.* Available at `http://flintlib.org`.

[16] M. Kï¿½ppe and S. Verdoolaege. *Computing parametric rational generating functions with a primal Barvinok algorithm.* Electron. J. Comb. 15, No. 1, Research Paper R16, 19 p. (2008).

[17] L. Pottier. *The Euclide algorithm in dimension n*. Research report, ISSAC 96, ACM Press 1996.

[18] A. Schürmann, *Exploiting polyhedral symmetries in social choice.* Social Choice and Welfare **40** (2013), 1097–1110.