PyNormaliz Tutorial

Winfried Bruns, Sebastian Gutsche, Justin Shenk, Richard Sieg

This small tutorial introduces the main syntax of the Python interface PyNormaliz for Normaliz. To install PyNormaliz on your machine, download PyNormaliz and run

```
python setup.py install
```

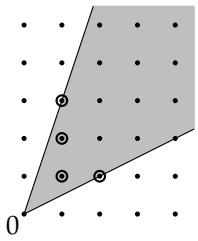
or

```
pip install .
```

If you are using python3 you might need to enter python3/pip3 instead of python/pip.

1 A cone in dimension 2

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



In the cone constructor of PyNormaliz, we specify the input type (`cone`) and an input matrix for this input type. To compute its Hilbert basis we use the point operator and call `HilbertBasis()` on our cone.

```
In [1]: from PyNormaliz import * # or import PyNormaliz

        gen = [[1,3],[2,1]]
        C = Cone(cone=gen)
        C.HilbertBasis()
```

```
[[1, 1], [1, 2], [1, 3], [2, 1]]
```

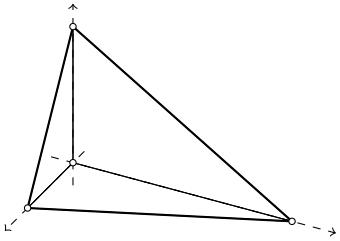
We can also use inequalities as an input. We also check that the extreme rays of the new cone are really the generators of the previous one.

```
In [2]: ineq = [[-1,2],[3,-1]]
        C2 = Cone(inequalities=ineq)
        gen2 = C2.ExtremeRays()
        print(gen==gen2)
        HB2 = C2.HilbertBasis()
        print(HB2)
```

```
True
[[1, 1], [1, 2], [1, 3], [2, 1]]
```

The general construction of objects in PyNormaliz follows the same fashion: You can generate a cone via `Cone` and then specify the usual input type(s) as parameters with the input data as `type=input data`. Once the cone is created, all its possible data is accessible using the `.` operator, like `C.HilbertBasis()`. Basically all input types and output data is available. An overview is given in the usual Normaliz manual.

2 A lattice polytope



Now we investigate a lattice simplex, which is not normal/integrally closed. The input type is "polytope". Note that we do not need a homogenizing variable but really specify the points in their actual dimension.

We compute its Hilbert series and Hilbert polynomial, which are also sometimes referred to as Ehrhart series and polynomial:

Hilbert series: $H(t) = \frac{1+14t+15t^2}{(1-t)^4}$

Hilbert polynomial: $p(k) = 1 + 4k + 8k^2 + 5k^3$

```
In [3]: vertices = [[0,0,0],[2,0,0],[0,3,0],[0,0,5]]
        poly = Cone(polytope=vertices)
        HB = poly.HilbertBasis()
        print(len(HB))
        print(poly.IsDeg1HilbertBasis())
        print(poly.HilbertSeries())
        print(poly.HilbertQuasiPolynomial())
```

```
19
False
[[1, 14, 15], [1, 1, 1, 1], 0]
[[1, 4, 8, 5], 1]
```

The output for the Hilbert series and quasipolynomial has to be read as follows: For the series, the first vector consists of the coefficients of the numerator polynomial, e.g., `[1,14,15]` is $1 + 14t + 15t^2$. The second vector collects the exponents in the denominator. For example `[1,1,1,1]` represents $(1-t)(1-t)(1-t)(1-t) = (1-t)^4$ and `[1,2]` is $(1-t)(1-t^2)$. The last entry is the possible shift of the Hilbert series. The output of the quasipolynomial usually consists of a list of vectors which show the coefficients of the respective polynomial associated to the congruence class modulo the period of the quasipolynomial. In the above example it is a proper polynomial, so only one vector is listed. The last entry is the greatest common divisor of all denominators in these polynomials.

If you are only interested in the lattice points, you can compute them via the command `Deg1Elements()`. Note that the output now has the homogenizing variable as last coordinate.

```
In [4]: LP = poly.Deg1Elements()
        print(len(LP))
        print(LP)
```

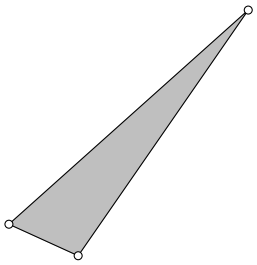
```
18
[[0, 0, 0, 1], [0, 0, 1, 1], [0, 0, 2, 1], [0, 0, 3, 1], [0, 0, 4, 1], [0, 0, 5, 1],
[0, 1, 0, 1], [0, 1, 1, 1], [0, 1, 2, 1], [0, 1, 3, 1], [0, 2, 0, 1], [0, 2, 1, 1],
[0, 3, 0, 1], [1, 0, 0, 1], [1, 0, 1, 1], [1, 0, 2, 1], [1, 1, 0, 1], [2, 0, 0, 1]]
```

If you would like to have a list of all already computed properties and data, you can use the `print_properties()` function.

```
In [12]: poly.print_properties()

Generators:
[[0, 0, 0, 1], [0, 0, 5, 1], [0, 3, 0, 1], [2, 0, 0, 1]]



ExtremeRays:
[[0, 0, 0, 1], [0, 0, 5, 1], [0, 3, 0, 1], [2, 0, 0, 1]]


SupportHyperplanes:
[[-15, -10, -6, 30], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]]

...
```

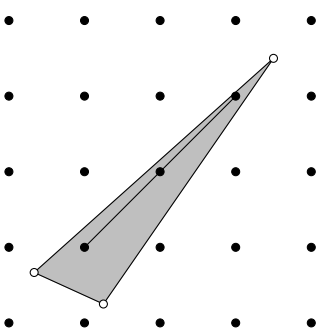
3 A rational polytope

We construct a polytope with vertices $(5/2, 3/2), (-2/3, -4/3), (1/4, -7/4)$. This is the first time we enter two different input types into the constructor. With the grading we declare the last coordinate to be the denominator of the vertices.



```
In [5]: rat_vert = [[5,3,2],[-2,-4,3],[1,-7,4]]
        g = [0,0,1]
        rat_poly = Cone(cone=rat_vert,grading=g)
        print(rat_poly.HilbertSeries())
        print(rat_poly.HilbertQuasiPolynomial())

[[1, 1, 4, 4, 5, 2, 4, 6, 3, 2, 6, 4, 2, 3], [1, 1, 12], 0]
[[24, 6, 47], [19, 6, 47], [16, 6, 47], [15, 6, 47], [40, 6, 47],
[19, 6, 47], [0, 6, 47], [31, 6, 47], [40, 6, 47], [3, 6, 47],
[16, 6, 47], [31, 6, 47], 24]
```

We compute the integer hull of this polytope which is again a cone object and thus we can access its vertices or support hyperplanes.

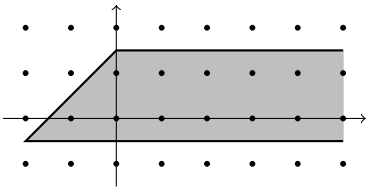


```
In [13]: int_hull = rat_poly.IntegerHull()
         print(int_hull.VerticesOfPolyhedron())
         print(int_hull.SupportHyperplanes())
         print(int_hull.Equations())

    [[0, -1, 1], [2, 1, 1]]
    [[1, -2, 0], [1, 0, 0]]
    [[1, -1, -1]]
```

4 A polyhedron

We define a polyhedron by inequalties:

$$\begin{aligned} 2x_2 &\geq -1 \\ 2x_2 &\leq 3 \\ -2x_1 + 2x_2 &\leq 3 \end{aligned}$$



The input type is `inhom_inequalities` where a row $(a_1, \ldots, a_n, b)$ in the input matrix defines the inequality

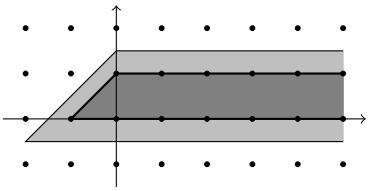
$$a_1 x_1 + \cdots + a_n x_n + b \geq 0.$$

In this case, `HilbertBasis()` returns the Hilbert basis of the recession cone associated to the polyhedron.

```
In [7]: ineq2 = [[0,2,1],[0,-2,3],[2,-2,3]]
        polyhedron = Cone(inhom_inequalities=ineq2)
        HB_rec = polyhedron.HilbertBasis()
        print(HB_rec)

        module_gen = polyhedron.ModuleGenerators()
        print(module_gen)

        vert_poly = polyhedron.VerticesOfPolyhedron()
        print(vert_poly)
```

```
[[1, 0, 0]]
[[-1, 0, 1], [0, 1, 1]]
[[-4, -1, 2], [0, 3, 2]]
```

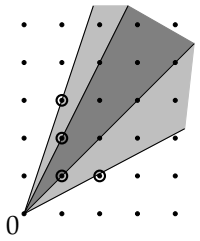
We can also define the polyhedron via its generators, i.e. its vertices and generators of its recession cone.



```
In [8]: poly2 = Cone(vertices=vert_poly,cone=[[1,0]])

        int_hull2 = poly2.IntegerHull()
        print(int_hull2.VerticesOfPolyhedron())
        print(int_hull2.ExtremeRays())
```

```
[[-1, 0, 1], [0, 1, 1]]
[[1, 0, 0]]
```

5 Creating new functions



As an example for the usefulness of PyNormaliz we write a small function that creates the intersection of two cones - something which would be quite messy if we work with input files.

```
In [9]: def intersection(cone1, cone2):
            intersection_ineq = cone1.SupportHyperplanes()+cone2.SupportHyperplanes()
            C = Cone(inequalities = intersection_ineq)
            return C

        C1 = Cone(cone=[[1,2],[2,1]])
        C2 = Cone(cone=[[1,1],[1,3]])
        print(intersection(C1,C2).ExtremeRays())
```

```
[[1, 1], [1, 2]]
```

A magic square is an $n \times n$ table in which each row and column and the two diagonals sum up to the same magic constant $\mathcal{M}$. The function

```
magic_square(n)
```

generates all defining equations of the $n \times n$ magic squares.

So only a few lines of code are necessary to create the respective cone and compute its Hilbert basis (in the dual mode) and Hilbert series.

```
In [11]: n=4
         grading = [1 if x<n else 0 for x in range(n**2)]
         M = Cone(equations=magic_square(n),grading=grading)
         print(len(M.HilbertBasis(DualMode=True)))
         print(M.HilbertSeries())
```

```
20
[[1, 4, 18, 36, 50, 36, 18, 4, 1], [1, 1, 1, 1, 2, 2, 2, 2], 0]
```