Contents

1 Installing Polyhedra 3

2 Contents 5
   2.1 Installation ................................................................. 5
   2.2 Representation ............................................................... 6
   2.3 Polyhedron ................................................................. 7

Python Module Index 11
Polyhedra is a package for polyhedra manipulations in Julia. It provides an unified interface for Polyhedra Manipulation Libraries such as CDDLlib.jl and LRSLib.jl.

Polyhedra can either be represented by a set of linear inequalities or by vertices and rays. In the first case, the points of the polyhedron are the points which satisfies all the inequalities and in the second case they are the points that can be expressed as a convex combination of the vertices plus a conic combination of the rays. The manipulations that Polyhedra can perform include

- Projection: Projection of a polyhedron on a lower dimensional space, e.g. Fourier-Motzkin elimination.
- Changing the Representation
  - Vertex enumeration problem: Computing the extremal vertices and rays from an inequality representation
  - Convex hull problem: Computing a set of linear inequalities describing the polyhedron from a vertex/ray representation
- Removal of redundant inequalities or redundant vertices/rays.
- Decomposition of 3D/2D polyhedra into a points and triangular faces, enabling easy visualization of 3D/2D polyhedra using DrakeVisualizer or GLVisualize.

Depending on the library, those manipulation can either be in floating point or exact rational arithmetic.

Polyhedra remains under active development, and we welcome your feedback, suggestions, and bug reports.
If you are familiar with Julia you can get started quickly by using the package manager to install Polyhedra:

```julia
julia> Pkg.add("Polyhedra")
```

And a Polyhedra Manipulation Library, e.g.:

```julia
julia> Pkg.add("CDDLib")
```
CHAPTER 2

Contents

Installation

This section shows how to install Julia, Polyhedra and a Polyhedra Manipulation Library of your choice.

Getting Julia

The first step is to install Julia. At the time of writing, the latest release of Julia is version 0.4 and the version 0.5 is in development. Polyhedra can be used with either Julia 0.4 and 0.5. Download links and more detailed instructions are available on the Julia website.

Getting Polyhedra

Julia has a package manager that makes the installation of new packages ridiculously simple. Open a Julia console (e.g. enter julia at the command line) and write:

```
julia> Pkg.add("Polyhedra")
```

To start using Polyhedra, you can now just write:

```
julia> using Polyhedra
```

Of course without installing a library, you won’t be able to do much. See the next section on installing a library.

Getting Libraries

Many C libraries are available for manipulating Polyhedra. Some of them works with floating point arithmetic and some of them can do the computation exactly using rational arithmetic and multiple precision libraries such as GMP. Julia also natively support Rational arithmetic using multiple precision libraries and of course floating point arithmetic. That makes the use of both arithmetic very easy and transparent.
The following table provides a list of Polyhedra Manipulation Libraries. When they have a Julia library implementing the interface of `Polyhedra.jl` then the “Library” column shows the name of the library.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Julia Package</th>
<th>Library</th>
<th>License</th>
<th>Exact Rational</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdd</td>
<td>CDDLib.jl</td>
<td>CDDLibrary()</td>
<td>GPL</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ConvexHull</td>
<td>ConvexHull.jl</td>
<td>ConvexHullLib()</td>
<td>MIT</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Lrs</td>
<td>LRSLib.jl</td>
<td>LRSLibrary()</td>
<td>GPL</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>qhull</td>
<td>QHull.jl</td>
<td>QHullLib()</td>
<td>MIT</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CHull2d</td>
<td>CHull2d.jl</td>
<td></td>
<td>MIT</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NewPolka</td>
<td>None</td>
<td></td>
<td>GPL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Parma Polyhedra Library</td>
<td>None</td>
<td></td>
<td>GPL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>pd</td>
<td>None</td>
<td></td>
<td>GPL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>porta</td>
<td>None</td>
<td></td>
<td>GPL</td>
<td>X (overflow !)</td>
<td></td>
</tr>
</tbody>
</table>

Please let me know if you plan to write a new wrapper (or an implementation in pure Julia). Since libraries use different algorithms, no library is better for every problem; here and here are comparisons.

**Representation**

Polyhedra can be described in 2 different ways.

1. **H-representation:** As the intersection of finitely many halfspaces given by its facets.
2. **V-representation:** As the convex hull of its vertices + the conic hull of its rays where ‘+’ is the Minkowski sum.

In `Polyhedra.jl`, those representations are given the respective abstract types `HRepresentation` and `VRepresentation` which are themselves subtypes of `Representation`.

For instance, consider the 2-dimensional polyhedron described by the following H-representation:

\[
\begin{align*}
    x_1 + x_2 & \leq 1 \\
    x_1 - x_2 & \leq 0 \\
    x_1 & \geq 0.
\end{align*}
\]

This set of inequalities can be written in the matrix form \(Ax \leq b\) where

\[
A = \begin{pmatrix} 1 & 1 \\ 1 & -1 \\ -1 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.
\]

Let’s create this H-representation using the concrete subtype `SimpleHRepresentation` of the abstract type `HRepresentation`:

```julia
julia> using Polyhedra
julia> A = [1 1;1 -1;-1 0]
julia> b = [1,0,0]
julia> hrep = SimpleHRepresentation(A, b)
julia> typeof(hrep)
```

This polyhedron has three vertices: (0, 0), (0, 1) and (0.5, 0.5). We can create this V-representation using the concrete subtype `SimpleVRepresentation` of the abstract type `VRepresentation`. Because 0.5 is fractional, have two choices: either use exact rational arithmetic:

```julia
```
Polyhedra

As seen in the previous section, a polyhedron can be described in 2 ways: either using the H-representation or the V-representation. A typical problem is: Given the H-(or V-)representation of one or several polyhedra, what is the H-(or V-)representation of some polyhedra obtained after some operations of those initial polyhedra. This description is similar to the description usually given to algorithms except that in that case we talk about numbers given in their binary representation and not polyhedra given in their H-(or V-)representation. This motivates the creation of a type representing polyhedra. Just like the abstract type `AbstractArray{N,T}` represents an N-dimensional array of elements of type T, the abstract type `Polyhedron{N,T}` represents an N-dimensional polyhedron of elements of type T.

There is typically one concrete subtype of `Polyhedron` by library. For instance, the CDD library defines `CDDPolyhedron` and the LRS library defines `LRSPolyhedron`. It must be said that the type T is not necessarily how the elements are stored internally by the library but the polyhedron will behave just like it is stored that way. For instance, when retrieving an H-(or V-)representation, the representation will be of type T. Therefore Int for T is may result in `InexactError`. For this reason, by default, the type T chosen is not a subtype of `Integer`.

Consider the representations `hrep`, `vrep` and `vrepf` created in the preceding section. One can use the CDD library, to create an instance of a concrete subtype of `Polyhedron`:

```julia
julia> using CDDLib
julia> polyf = polyhedron(hrep, CDDLibrary())
julia> typeof(polyf)
CDDLib.CDDPolyhedron{2, Float64}
```

We see that the library has chosen to deal with floating point arithmetic. This decision does not depend on the type of hrep but only on the instance of `CDDLibrary` given. `CDDLibrary` creates `CDDPolyhedron` of type either `Float64` or `Rational{BigInt}`. One can choose the first one using `CDDLibrary(:float)` and the second one using `CDDLibrary(:exact)`, by default it is `:float`:

```julia
julia> poly = polyhedron(hrep, CDDLibrary(:exact))
julia> typeof(poly)
CDDLib.CDDPolyhedron{2, Rational{BigInt}}
```

The first polyhedron `polyf` can also be created from its V-representation using either of the 4 following lines:

```julia
julia> polyf = polyhedron(vrep, CDDLibrary(:float))
julia> polyf = polyhedron(vrepf, CDDLibrary())
julia> polyf = polyhedron(vrep, CDDLibrary(:float))
julia> polyf = polyhedron(vrep, CDDLibrary())
```

and `poly` using either of those lines:
of course, creating a representation in floating points with exact arithmetic works here because we have 0.5 which is 0.1 in binary but in general, is not a good idea.: 

```
julia> Rational{BigInt}(1/2)
1//2
julia> Rational{BigInt}(1/3)
6004799503160661//18014398509481984
julia> Rational{BigInt}(1/5)
3602879701896397//18014398509481984
```

### Retrieving a representation

One can retrieve an H-representation (resp. V-representation) from a polyhedron using hrep (resp. vrep). The concrete subtype of HRepresentation (resp. VRepresentation) returned is not necessarily the same that the one used to create the polyhedron. As a rule of thumb, it is the representation the closest to the internal representation used by the library.: 

```
julia> hrep = hrep(poly)
julia> typeof(hrep)
Polyhedra.LiftedHRepresentation{2,Rational{BigInt}}
julia> hrep = SimpleHRepresentation(hrep)
julia> typeof(hrep)
Polyhedra.SimpleHRepresentation{2,Rational{BigInt}}
julia> hrep.A
3x2 Array{Rational{BigInt},2}:
  1//1  1//1
  1//1 -1//1
 -1//1  0//1
julia> hrep.b
3-element Array{Rational{BigInt},1}:
  1//1
  0//1
  0//1
julia> vrep = vrep(poly)
julia> typeof(vrep)
Polyhedra.LiftedVRepresentation{2,Rational{BigInt}}
julia> vrep = SimpleVRepresentation(vrep)
julia> typeof(vrep)
Polyhedra.SimpleVRepresentation{2,Rational{BigInt}}
julia> vrep.V
3x2 Array{Rational{BigInt},2}:
  1//2  1//2
  0//1  1//1
  0//1  0//1
julia> vrep.R
0x2 Array{Rational{BigInt},2}
```
Creating a polyhedron from the feasible set of a JuMP model

A typical application of polyhedral computation is the computation of the set of extreme points and rays of the feasible set of an optimization problem. This comes from the fact that given a minimization of a concave function (or maximization of a convex function) on a convex feasible set (e.g. Linear Programming), we are either in the following three situations:

- The feasible set is empty, i.e. the problem is infeasible.
- An extreme ray is optimal, i.e. the problem is unbounded (or it may also be bounded if the objective is constant along the ray).
- An extreme point is optimal.

A JuMP model is treated by `polyhedron` just like any H-representation. For example, the hypercube of dimension \( n \) can be created as follows:

```julia
m = Model()
@variable(m, 0 x[1:n] 1)
poly = polyhedron(m, CDDLibrary(:exact))
```

In fact, the MathProgBase representation of the feasible set of a linear program:

\[
\begin{align*}
lb & \leq Ax \leq ub \\
l & \leq x & \leq u
\end{align*}
\]

has `LPHRepresentation` as a corresponding H-representation. A JuMP Model can be converted to this representation using `LPHRepresentation(m)`.

Projecting a polyhedron

Consider the polyhedron created in the beginning of this section. As a reminder, it represents the following H-representation:

\[
\begin{align*}
x_1 + x_2 & \leq 1 \\
x_1 - x_2 & \leq 0 \\
x_1 & \geq 0.
\end{align*}
\]

One can verify that for any \( 0 \leq x_2 \leq 1 \), there exists a value \( x_1 \) such that \((x_1, x_2)\) is in this polyhedron. This means that the H-representation obtained by eliminating \( x_1 \) is:

\[
\begin{align*}
x_1 & \leq 1 \\
x_1 & \geq 0.
\end{align*}
\]

where \( x_1 \) in the H-representation above represents \( x_2 \) in the previous one. This can be obtained as follows:

```julia
julia> poly_x2 = eliminate(poly, [1])
julia> hrep(poly_x2)
H-representation
begin
  2 2 rational
  1/1 -1/1
  0/1 1/1
end
```
There is two methods of computing the elimination implemented in CDDLib: Fourier-Motzkin elimination and block elimination. As written by K. Fukuda in CDD’s documentation, “[Block elimination] might be a faster way to eliminate variables than the repeated [Fourier-Motzkin elimination] when the number of variables to eliminate is large”. You can specify the method to use as a third argument, e.g. `eliminate(poly, [1], :FourierMotzkin), eliminate(poly, [1], :BlockElimination)`. A third method can be chosen: :ProjectGenerators. It computes the V-representation and then project each of its elements. This is the method of choice when the V-representation is already computed.

If nothing is specified as in the block of code above, the behavior depends on the polyhedral library. If neither Fourier-Motzkin nor block elimination is implemented or if the V-representation is already computed then :ProjectGenerators is chosen. Otherwise, Polyhedra lets the library decide. In CDDLib, :FourierMotzkin is chosen when only the last dimension needs to be eliminated and :BlockElimination is chosen otherwise. Note that CDDLib only supports projecting the last trailing dimensions.
p
Polyhedra, 3
P

Polyhedra (module), 1