Polymake.jl: A New Interface to `polymake`

Marek Kaluba[1,2], Benjamin Lorenz[1], and Sascha Timme[1(✉)]

[1] Chair of Discrete Mathematics/Geometry, Technische Universität Berlin, Berlin, Germany
timme@math.tu-berlin.de
[2] Adam Mickiewicz University in Poznań, Poznań, Poland

Abstract. We present the Julia interface `Polymake.jl` to `polymake`, a software for research in polyhedral geometry. We describe the technical design and how the integration into Julia makes it possible to combine `polymake` with state-of-the-art numerical software.

Keywords: Polymake · Julia

1 Introduction

`polymake` is an open source software system for computing with a wide range of objects from polyhedral geometry and related areas [4]. This includes convex polytopes and polyhedral fans as well as matroids, finite permutation groups, ideals in polynomial rings and tropical varieties. The user is interfacing the `polymake` library through Perl language.

In this note we provide a brief overview of a new interface `Polymake.jl`, which allows the use of `polymake` in Julia [1]. Julia is a high-level, dynamic programming language. Distinctive aspects of Julia's design include a type system with parametric polymorphism and multiple dispatch as its core programming paradigm. The package `Polymake.jl` can be installed, without any preparations, using the build-in package manager that comes with Julia. The source code is available at https://github.com/oscar-system/Polymake.jl.

2 Functionality

In `polymake` the objects that a user encounters can be roughly divided into the following three classes

M. Kaluba—The author was supported by the National Science Center, Poland grant 2017/26/D/ST1/00103. This research is carried out in the framework of the DFG funded Cluster of Excellence EXC 2046 MATH+: *The Berlin Mathematics Research Center* within the Emerging Fields area.
S. Timme—The author was supported by the Deutsche Forschungsgemeinschaft (German Research Foundation) Graduiertenkolleg *Facets of Complexity* (GRK 2434).

A. M. Bigatti et al. (Eds.): ICMS 2020, LNCS 12097, pp. 377–385, 2020.
https://doi.org/10.1007/978-3-030-52200-1_37

– *big objects* (e.g., cones, polytopes, simplicial complexes),
– *small objects* (e.g., matrices, polynomials, tropical numbers),
– *user functions*.

Broadly speaking, *big objects* correspond to mathematical concepts with well defined semantics. These can be queried, accumulate information (e.g., a polytope defined by a set of points can "learn" its hyperplane representation), and are constructed usually in Perl. Big objects implement *methods*, i.e., functions which operate on, and perform computations specific to the corresponding object. *Small objects* correspond to types or data structures which are implemented in C++. Standalone *user functions* are exposed to the user via the Perl interpreter.

These entities are mapped to Julia in the following way:

– big objects are exposed as opaque Perl objects that can be queried for their properties (they are only computed when queried for the first time),
– small objects are wrapped through an intermediate C++ layer between Julia and `libpolymake` generated by `CxxWrap.jl`,
– methods and user functions are mapped to Julia functions, in the case of methods, the parent object being the first argument.

A unique feature of `Polymake.jl` is based on the affinity of Julia to C and C++ programming languages. As Julia provides the possibility to call functions from dynamic libraries directly, one can call any function from the `polymake` library as long as the function symbol is exported. In `polymake`, due to extensive use of templates in the C++ library, the precise definition of a function needs to be often explicitly instantiated. Such instantiaton can be easily added to the `Polymake.jl` C++ wrapper. An example of such functionality is

```
Polymake.solve_LP(inequalities, equalities, objective; sense=max)
```

a function, which directly taps into the `polymake` framework for linear programming. It is worth pointing that the signature of the exposed `solve_LP` will accept any instances of Julias **`AbstractMatrix`** or **`AbstractVector`** (where appropriate) in the paradigm of generic programming.

3 Technical Contribution

The `Polymake.jl` interface is based on `CxxWrap.jl`[1], a Julia package which aims to provide a seamless interoperability between C++ and Julia. The interface is separated into two parts: a C++ wrapper library and a Julia package. The former, `libpolymake`, a dynamic library, wraps the data structures (*small types*) in a Julia-compatible way and exposes functions from the callable C++ `polymake` library. It is then loaded through `CxxWrap.jl` where the Julia part of the package generates functions accessible from Julia.

The installation of `Polymake.jl` is performed through Julia's package manager with the help of `BinaryBuilder.jl` infrastructure. Thanks to this infrastructure

[1] Available at https://github.com/JuliaInterop/CxxWrap.jl.

it is not necessary for the user to perform any preparations except for installing Julia itself. All dependencies of `Polymake.jl` (including the `polymake` library, the Perl interpreter and supplementary libraries) are installed in a binary form. The complete installation of `Polymake.jl` should take no longer than 5 minutes on modern hardware

Due to extensive use of metaprogramming, relatively little code was necessary to make most of the functionality of `polymake` available in Julia: as of version 0.4.1 `Polymake.jl` consists of about 1200 lines of C++ code and 1600 lines of Julia code. In particular, only the small objects need to be manually wrapped, while functions, constructors for big objects and their methods are generated automatically from the information provided by `polymake` itself. This automatic code generation takes place during precompilation which is done only once during the installation. Loading `Polymake.jl` brings the familiar `polymake` welcome banner.

```
julia> using Pkg; Pkg.add("Polymake")
  Updating registry at ~/.julia/registries/General
  [ ... ]
  Building Polymake → ~/.julia/packages/Polymake/[...]/deps/build.log
julia> using Polymake
[ Info: Precompiling Polymake [d720cf60-89b5-51f5-aff5-213f193123e7]
[ Info: Generating module common
[ Info: Generating module ideal
[ Info: Generating module graph
[ Info: Generating module fulton
[ Info: Generating module fan
[ Info: Generating module group
[ Info: Generating module polytope
[ Info: Generating module topaz
[ Info: Generating module tropical
[ Info: Generating module matroid

polymake version 4.0
Copyright (c) 1997-2020
Ewgenij Gawrilow, Michael Joswig, and the polymake team
Technische Universität Berlin, Germany
https://polymake.org

This is free software licensed under GPL; see the source for
copying conditions.
There is NO warranty; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.
```

The latest version of `Polymake.jl` is 0.4.1 which is compatible with at Julia 1.3 and newer. The latest `polymake` version is available in `Polymake.jl` within two weeks of release (currently `polymake` 4.0).

Big Objects

All big objects are constructed by direct calls to their constructors, e.g.

```
polytope.Polytope(POINTS=[1 1 2; 1 3 4])
```

constructs a rational polytope from (homogeneous) coordinates of points given row-wise. We attach the `polymake` docstring to the structure such that the documentation is readily available in Julia.

```
help?> polytope.Polytope
Not necessarily bounded convex polyhedron, i.e., the feasible region of a linear
      program.
[...]
```

Template parameters can also be passed to big objects, e.g., to construct a polytope with floating point precision it is sufficient to call

```
polytope.Polytope{Float64}(...) .
```

A caveat is that all parameters must be valid Julia objects.[2] The properties of big objects are accessible through the `bigobject.property` syntax which mirrors the `$bigobject->property` syntax in `polymake`. Note that some properties of big objects in `polymake` are indeed methods with no arguments and therefore in Julia they are only available as such.

Small Objects

The list of small objects available in `Polymake.jl` includes basic types such as arbitrary size integers (subtypes `Integer`, rationals (subtypes `Real`, vectors and matrices (subtypes of `AbstractArray`), and many more. These data types can be converted to appropriate Julia types, but are also subtypes of the corresponding Julia abstract types (as indicated above). This allows to use `Polymake.jl` types in generic methods, which is the paradigm of Julia programming.

As already mentioned, these small objects need to be manually wrapped in the C++ part of `Polymake.jl`. In particular, all possible combinations of such types, e.g., an array of sets of rationals, need to be explicitly wrapped. Note that `polymake` is able to generate dynamically any combination of small objects. Thus, we cannot guarantee that all small objects a user will encounter is covered. However, the small objects available in `Polymake.jl` are sufficient for the most common use cases.

Functions

A function in `Polymake.jl` calling `polymake` may return either a big or a small object, and the generic return type (`PropertyValue`, a container opaque to Julia) is transparently converted to one of the known (small) data types[3]. If the data type of the returned function value is not known to `Polymake.jl`, the conversion fails and an instance of `PropertyValue` is returned. It can be either passed back as an argument to a `Polymake.jl` function, or converted to a known type using the `@convert_to` macro.

[2] For advanced use (when this is not the case) we provide the `@pm` macro.

[3] This conversion can be deactivated by passing `PropertyValue` type as the first argument to function/method call.

```
julia> K5 = graph.complete(5);
julia> K5.MAX_CLIQUES
PropertyValue wrapping pm::PowerSet<long, pm::operations::cmp>
{{0 1 2 3 4}}
julia> @convert_to Array{Set} K5.MAX_CLIQUES
pm::Array<pm::Set<long, pm::operations::cmp> >
{0 1 2 3 4}
```

All user functions from `polymake` are available in modules corresponding to their applications, e.g. `homology` functions from the application `topaz` can be called as `topaz.homology(...)` in Julia. Moreover `polymake` docstrings for functions are available in Julia to allow for easy help[4]:

```
julia> ?topaz.homology
  homology(complex, co; Options)

   Calculate the reduced (co-)homology groups of a simplicial complex.

  Arguments:
    Array<Set<Int>> complex
    Bool co set to true for cohomology

  Options:
    dim_low => Int narrows the dimension range of interest, with negative values being
      treated as co-dimensions
    dim_high => Int see dim_low

    [ ... ]
```

Function Arguments. Functions in `Polymake.jl` accept the following as their arguments: simple data types (bools, machine integers, floats), wrapped native types, or objects returned by `polymake` (e.g., `BigObject`, or `PropertyValue`). Due to the easy extendability of methods in Julia, a foreign type could be passed seamlessly to `Polymake.jl` function if an appropriate `Base.convert` method, which return one of the above types, is defined:

```
Base.convert(::Type{Polymake.PolymakeType}, x::ForeignType)
```

`Polymake.jl` also wraps the extensive visualization methods of `polymake` which can be used to produce images and animations of geometric objects. These include the interactive visualizations using `three.js`. Due to the convenient extendability of Julia, the visualization also integrates seamlessly with Jupyter notebooks.

4 Example

This section demonstrates the interface of `Polymake.jl` on a concrete example. An advantage of the package is that it allows effortless combination of computations in polyhedral geometry with e.g., state-of-the-art numerical software. Here we combine `Polymake.jl` with `HomotopyContinuation.jl` [3], a Julia package for

[4] The documentation currently uses the Perl syntax.

numerically solving systems of polynomial equations. In particular, we test a theoretical result from Soprunova and Sottile [8] on non-trivial lower bounds for the number of real solutions to sparse polynomial systems.

The results show how we can construct a sparse polynomial system that has a non-trivial lower bound on the number of real solutions starting from an integral point configuration. We start with the 10 lattice points $A = \{a_1, \ldots, a_{10}\} \subset \mathbb{Z}^2$ of the scaled two-dimensional simplex $3\Delta_2$ and look at the regular triangulation $\mathcal{T}$ induced by the lifting $\lambda = [12, 3, 0, 0, 8, 1, 0, 9, 5, 15]$.

```
julia> A = polytope.lattice_points(polytope.simplex(2,3));
julia> λ = [12, 3, 0, 0, 8, 1, 0, 9, 5, 15];
julia> F = polytope.regular_subdivision(A, λ);
julia> T = topaz.GeometricSimplicialComplex(COORDINATES = A[:,2:end], FACETS = F)
type: GeometricSimplicialComplex<Rational>

COORDINATES
        0 0
        0 1
        [ ... ]

FACETS
  pm::Set<long, pm::operations::cmp>
{5 6 8}
  pm::Set<long, pm::operations::cmp>
{5 7 8}
  [ ... ]
```

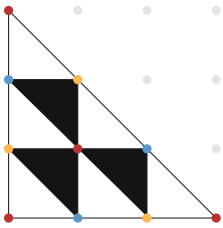

Fig. 1. Foldable subdivision of $3\Delta_2$.

The triangulation $\mathcal{T}$ is very special in that it is *foldable* (or "balanced"), i.e., the dual graph is bipartite. This means that the triangles can be colored, say, black and white such that no two triangles of the same color share an edge. See Fig. 1 for an illustration. The *signature* $\sigma(\mathcal{T})$ of a balanced triangulation of a polygon is the absolute value of the difference of the number of black triangles and the number of the white triangles whose normalized volume is odd. The vertices of a foldable triangulation can be colored by $d + 1$ colors [6] (such that vertices of the same color do not share an edge), where d is the dimension. Here $d = 2$, so 3 colors suffice. We can check both properties with `polymake`.

```
julia> (foldable = T.FOLDABLE, signature = T.SIGNATURE)
  (foldable = true, signature = 3)
```

Now, a *Wroński polynomial* $W_{\mathcal{T},s}(x)$ has the lifted lattice points as exponents, and only one non-zero coefficient $c_i \in \mathbb{R}$ per color class of vertices of the triangulation

$$W_{\mathcal{T},s}(x) = \sum_{i=0}^{d} c_i \left(\sum_{j:\ \mathrm{color}(a_j)=i} s^{\lambda_i} x^{a_j} \right) .$$

A *Wroński system* consists of d Wroński polynomials with respect to the same lattice points A and lifting λ such that for general $s = s_0 \in [0,1]$ it has precisely $d!\,\mathrm{vol}(\mathrm{conv}(A))$ distinct complex solutions, which is the highest possible number by Kushnirenko's Theorem [7].

Soprunova and Sottile showed that a Wroński system has at least $\sigma(\mathcal{T})$ distinct real solutions if two conditions are satisfied. First, a certain double cover of the real toric variety associated with A must be orientable. This is the case here. Second, the *Wroński center ideal*, a zero-dimensional ideal in coordinates x_1, x_2 and s depending on $\mathcal{T}$, has no real roots with s coordinate between 0 and 1. Let us verify this condition using `HomotopyContinuation.jl`. Luckily for us, `polymake` already has an implementation of the Wroński center ideal. However, we have to convert the ideal returned by `Polymake.jl` to a polynomial system which `HomotopyContinuation.jl` understands. This can be accomplished with a simple routine.

```
julia> using HomotopyContinuation
julia> function hc_poly(f, vars)
          M = Polymake.monomials_as_matrix(f)
          monomials = [prod(vars.^m) for m in eachrow(M)]
          coeffs = Int.(Polymake.coefficients_as_vector(f))
          sum(map(*, coeffs, monomials))
       end;
julia> I = polytope.wronski_center_ideal(A, λ)
julia> @polyvar x[1:2] s;
julia> HC_I = [hc_poly(f, [x;s]) for f in I.GENERATORS]
  3-element Array{Polynomial{true,Int},1}:
    x1^3*s^15 + s^12 + x1*x2*s + x2^3
    x1^2*s^9 + x2*s^3 + x1*x2^2
    x1*s^8 + x1^2*x2*s^5 + x2^2
```

Since we are only interested in solutions in the algebraic torus $(\mathbb{C}^*)^3$ we can use polyhedral homotopy [5] to efficiently compute the solutions.

```
julia> @time res = solve(HC_I; start_system = :polyhedral, only_torus = true)
  0.010595 seconds (3.03 k allocations: 215.766 KiB)
Result{Array{Complex{Float64},1}} with 54 solutions
====================================================
* 54 non-singular solutions (2 real)
* 0 singular solutions (0 real)
* 54 paths tracked
* random seed: 782949
```

Out of the 54 complex roots only two solutions are real. Strictly speaking, this is here only checked heuristically by looking at the size of the imaginary parts.

However, a certified version can be obtained by using `alphaCertified`[2]. By closer inspection, we see that no solution has the s-coordinate in $(0, 1)$.

```
julia> HomotopyContinuation.real_solutions(res)
2-element Array{Array{Float64,1},1}:
 [-0.2117580095433453, -215.72260079314424, 4.411470567441922]
 [-0.6943590430596768, -0.41424188458258815, -0.8952189506082179]
```

Therefore, the Wroński system with respect to A and λ for $s = 1$ has at least $\sigma(\mathcal{T}) = 3$ real solutions. Let us verify this on an example.

```
julia> c = Vector{Polymake.Rational}[[19,8,-19], [39,7,42]];
julia> W = polytope.wronski_system(A, λ, c, 1)
julia> HC_W = [hc_poly(f, x) for f in W.GENERATORS];
julia> W_res = HomotopyContinuation.solve(HC_W)
Result{Array{Complex{Float64},1}} with 9 solutions
==================================================
* 9 non-singular solutions (3 real)
* 0 singular solutions (0 real)
* 9 paths tracked
* random seed: 813729
```

Finally, we can use the `ImplicitPlots.jl` package to visualize the real solutions of the Wroński system `W` (Fig. 2).

```
julia> W_real = HomotopyContinuation.real_solutions(W_res)
julia> using ImplicitPlots, Plots;
julia> p = plot(aspect_ratio = :equal);
julia> implicit_plot!(p, HC_W[1]);
julia> implicit_plot!(p, HC_W[2]; color=:indianred);
julia> scatter!(first.(r), last.(r), markercolor=:black)
```

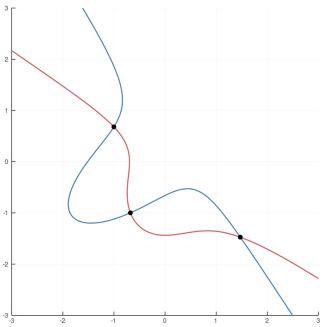

Fig. 2. Visualization of the Wroński system W and its 3 solutions.

Acknowledgements. We would like to express our thanks to Alexej Jordan and Sebastian Gutsche for all their help during the development of `Polymake.jl`.

References

1. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: a fresh approach to numerical computing. SIAM Rev. **59**(1), 65–98 (2017)
2. Hauenstein, J.D., Sottile, F.: Algorithm 921: alphaCertified: certifying solutions to polynomial systems. ACM Trans. Math. Softw. (TOMS) **38**(4), 1–20 (2012)
3. Breiding, P., Timme, S.: HomotopyContinuation.jl: a package for homotopy continuation in Julia. In: International Congress on Mathematical Software, pp. 458–465. Springer (2018)
4. Gawrilow, E., Joswig, M.: polymake: a framework for analyzing convex polytopes. Polytopes–combinatorics and computation (Oberwolfach, 1997). DMV Sem. **29**, 43–73 (2000)
5. Huber, B., Sturmfels, B.: A polyhedral method for solving sparse polynomial systems. Math. Comput. **64**(212), 1541–1555 (1995)
6. Joswig, M.: Projectivities in simplicial complexes and colorings of simple polytopes. Math. Z. **240**, 243–259 (2002)
7. Kushnirenko, A.G.: A Newton polyhedron and the number of solutions of a system of k equations in k unknowns. Usp. Math. Nauk. **30**(1975), 266–267 (1975)
8. Soprunova, E., Sottile, F.: Lower bounds for real solutions to sparse polynomial systems. Adv. Math. **204**, 116–151 (2006)