OSCAR

The OSCAR Team

Bad Münster am Stein, February 18, 2019

Outline

Outline

Outline

1. What did we promise?
2. Examples: Current use of the individual cornerstone systems
3. Examples: Current use of more than one cornerstone system

Outline

Outline

Long-Term goals

(S1) Integrate all computer algebra systems, libraries and packages developed within the TRR 195 into a visionary next generation open source computer algebra system surpassing the combined mathematical capabilities of the underlying systems. pause

(S2) Boost the performance of the visionary system to a new level by parallelising fundamental algorithms.

Overview: The cornerstones and Julia

Visionary system surpassing the combined capabilities of the underlying systems

GAP: computational discrete algebra, group and representation theory, general purpose high level interpreted programming language.

Singular: polynomial computations, with emphasis on algebraic geometry, commutative algebra, and singularity theory.

Examples:

Multigraded equivariant Cox rings of toric varieties over number fields

Graphs of groups in division algebras

Matrix groups over polynomial rings with coefficients in number fields

Gröbner fans over fields with discrete valuations

polymake: convex polytopes, polyhedral and stacky fans, simplicial complexes and related objects from combinatorics and geometry.

ANTIC: number theoretic software featuring computations in and with number fields and generic finitely presented rings.

The cornerstones

We have two kinds of cornerstones

The cornerstones

We have two kinds of cornerstones

- Interfaces from Julia to GAP, Polymake, and Singular, which form "interface modules"

The cornerstones

We have two kinds of cornerstones

- Interfaces from Julia to GAP, Polymake, and Singular, which form "interface modules"
- Nemo/AbstractAlgebra/Hecke provide functionality in Julia

The cornerstones

We have two kinds of cornerstones

- Interfaces from Julia to GAP, Polymake, and Singular, which form "interface modules"
- Nemo/AbstractAlgebra/Hecke provide functionality in Julia

So we talk about interfaced functionality for GAP/Singular/Polymake, and about provided functionality by Nemo/AbstractAlgebra/Hecke

Polymake.jl example

```
julia> p = perlobj( "Polytope",
  Dict( "INEQUALITIES" => [ 0 1 0 ; 0 0 1 ; -1 1 1 ] ) );
julia> l = p.LATTICE_POINTS_GENERATORS;
julia> l[1]
pm::Matrix<pm::Integer>
1 0 1
1 1 0
julia> l[2]
pm::Matrix<pm::Integer>
0 1 0
0 0 1
julia> polytope.intersection( p, polytope.cube( 2 ) ).VERTICES
pm::Matrix<pm::Rational>
1 1 0
1 1 1
1 0 1
```

Polymake.jl features

Most functionality of Polymake is exported to Julia:

Polymake.jl features

Most functionality of Polymake is exported to Julia:

- Many data types are mapped into Julia (manually)

Polymake.jl features

Most functionality of Polymake is exported to Julia:

- Many data types are mapped into Julia (manually)
- Small objects (e.g., matrices, vectors) fulfill their corresponding abstract Julia types,

Polymake.jl features

Most functionality of Polymake is exported to Julia:

- Many data types are mapped into Julia (manually)
- Small objects (e.g., matrices, vectors) fulfill their corresponding abstract Julia types,so they can be used like Julia objects, but also converted into native Julia objects.
- All types of "Big Objects" (e.g., Polytopes, Fans) can be created and manipulated in Julia

Polymake.jl features

Most functionality of Polymake is exported to Julia:

- Many data types are mapped into Julia (manually)
- Small objects (e.g., matrices, vectors) fulfill their corresponding abstract Julia types,so they can be used like Julia objects, but also converted into native Julia objects.
- All types of "Big Objects" (e.g., Polytopes, Fans) can be created and manipulated in Julia
- All properties and functions are automatically exported to Julia, and are mirrored as Julia functions

Singular.jl example

```
julia> R, (x, y) = PolynomialRing(QQ, ["x", "y"]);
julia> I = Ideal(R, x + 1, x^2*y + 1)
Singular Ideal over Singular Polynomial Ring
  (QQ),(x,y),(dp(2),C) with generators (x+1, x^2*y+1)
julia> G = std(I)
Singular Ideal over Singular Polynomial Ring
  (QQ),(x,y),(dp(2),C) with generators (y+1, x+1)
julia> fr = fres(G, 0)
..omitted..
julia> fr=minres(fr)
Singular Resolution:
R^1 <- R^2 <- R^1
julia> B=betti(fr)
1×3 Array{Int32,2}:
 1  2  1
```

Singular.jl features

Singular.jl exports and abstracts Singular functionality in Julia

Singular.jl features

Singular.jl exports and abstracts Singular functionality in Julia

- All kernel types and many kernel functions are exported into Julia

Singular.jl features

Singular.jl exports and abstracts Singular functionality in Julia

- All kernel types and many kernel functions are exported into Julia
- Singular.jl contains abstractions from pure Singular, to make use of Julia's type system and get rid of implicit states (e.g., the current ring)

Singular.jl features

Singular.jl exports and abstracts Singular functionality in Julia

- All kernel types and many kernel functions are exported into Julia
- Singular.jl contains abstractions from pure Singular, to make use of Julia's type system and get rid of implicit states (e.g., the current ring)
- Rings defined in Julia can be used as coefficient rings for Singular polynomial rings

Furthermore, Singular.jl will interface the latest Singular features, developed in the TRR

- New non-commutative Groebner basis and algebra in Singular:Plural (Zerz, Levandovskyy, ...)
- Massive shared memory and multi-node parallelization in Singular via pSingular and GPI-Space (Behrends, Böhm, Steenpass, ...)

GAPJulia example

```
julia> S5 = GAP.Globals.SymmetricGroup( 5 )
GAP: SymmetricGroup( [ 1 .. 5 ] )
julia> orb = GAP.Globals.Orbit( S5, 1, GAP.Globals.OnPoints )
GAP: [ 1, 2, 3, 4, 5 ]
julia> g1 = GAP.Globals.GeneratorsOfGroup( S5 )[ 1 ]
GAP: (1,2,3,4,5)
julia> 4^g1
5
```

GAPJulia example, reversed

A GAP lin. comb. of 4620th roots of 1 (about $-3.3 \cdot 10^{-35}$), numerically approximated by arb-library via Nemo: Arb (interval arithmetic):

```
gap> a:= EY(5);;  b:= EY(7);;  c:= EY(11);;  d:= EY(12);;
gap> z:= [ -12230241886849032, -27721673763224765,
>  19808983844326917, 5079707604555803 ] * [a,b,c,d];;
gap> IsPositiveRealPartCyclotomic( z : ShowPrecision );
#I  precision needed: 256
false
```

Creating and factoring a polynomial via Nemo:

```
gap> R := Nemo_PolynomialRing( Nemo_QQ, "x" );;
gap> x := JuliaPointer(Nemo_Polynomial( R, [ 0, 1 ] ));
<Julia: x>
gap> Julia.Nemo.factor(x^10-1);
<Julia: 1 * (x^4-x^3+x^2-x+1) * (x-1) *
       (x^4+x^3+x^2+x+1) * (x+1)>
```

GAPJulia features

GAPJulia is the bidirectional interface between GAP and Julia:

GAPJulia features

GAPJulia is the bidirectional interface between GAP and Julia:

- Uses a unified GC for GAP and Julia

GAPJulia features

GAPJulia is the bidirectional interface between GAP and Julia:

- Uses a unified GC for GAP and Julia
- All objects can be transparently shared between GAP and Julia

GAPJulia features

GAPJulia is the bidirectional interface between GAP and Julia:

- Uses a unified GC for GAP and Julia
- All objects can be transparently shared between GAP and Julia
- All functions can be called from either side

GAPJulia features

GAPJulia is the bidirectional interface between GAP and Julia:

- Uses a unified GC for GAP and Julia
- All objects can be transparently shared between GAP and Julia
- All functions can be called from either side
- No function call or object handling/conversion overhead

Nemo/Hecke example

```
julia> R, x = PolynomialRing(QQ, "x")
(Univariate Polynomial Ring in x over Rational Field, x)
julia> crt(x-1, x^2+1, x+2, x^2+2)
-3*x^2+x-4
julia> rem(ans, x^2+1), rem(ans, x^2+2)
(x-1, x+2)
julia> S = ResidueRing(R, (x^2+1)*(x^2+2))
Residue ring of Univariate Polynomial Ring in x
   over Rational Field modulo x^4+3*x^2+2
julia> inv(S(x+2))
-1//30*x^3+1//15*x^2-7//30*x+7//15
```

Nemo/Hecke example

```
julia> K, s10 = quadratic_field(10);
julia> c, mc = class_group(K)
(GrpAb: Z/2, ClassGroup map of Set of ideals ...‘
julia> Z_K = maximal_order(K);
julia> P = 2*Z_K + Z_K(s10)*Z_K
<2, s10>
julia> isprime(P), isprincipal(P)
(true, false)
julia> H = number_field(hilbert_class_field(K))
non-simple Relative number field over
  Number field over Rational Field with
    defining polynomial x^2-10
    with defining polynomials ... [x_1^2+(-2)]
```

Nemo/Hecke features

Basic features:

- Integers, rationals, Z/nZ, GF(p), finite fields, padics, qadics, real/complex ball arithmetic

Nemo/Hecke features

Basic features:

- Integers, rationals, Z/nZ, GF(p), finite fields, padics, qadics, real/complex ball arithmetic
- Number field arithmetic

Nemo/Hecke features

Basic features:

- Integers, rationals, Z/nZ, GF(p), finite fields, padics, qadics, real/complex ball arithmetic
- Number field arithmetic
- Univariate and multivariate polynomial rings, Laurent series, Puiseux series, rational functions, residue rings

Nemo/Hecke features

Basic features:

- Integers, rationals, Z/nZ, GF(p), finite fields, padics, qadics, real/complex ball arithmetic
- Number field arithmetic
- Univariate and multivariate polynomial rings, Laurent series, Puiseux series, rational functions, residue rings
- Matrices (both matrix spaces and matrix algebras)

Nemo/Hecke features

Basic features:

- Integers, rationals, Z/nZ, GF(p), finite fields, padics, qadics, real/complex ball arithmetic
- Number field arithmetic
- Univariate and multivariate polynomial rings, Laurent series, Puiseux series, rational functions, residue rings
- Matrices (both matrix spaces and matrix algebras)
- Factorization, (x)gcd, resultant, coprime factorisation, crt, Farey lift
- Ideals

Nemo/Hecke features

Advanced features:

- Class and unit groups

Nemo/Hecke features

Advanced features:

- Class and unit groups
- absolute and relative fields

Nemo/Hecke features

Advanced features:

- Class and unit groups
- absolute and relative fields
- certified conjugates (ARB)

Nemo/Hecke features

Advanced features:

- Class and unit groups
- absolute and relative fields
- certified conjugates (ARB)
- lattices, sparse and dense linear algebra

Nemo/Hecke features

Advanced features:

- Class and unit groups
- absolute and relative fields
- certified conjugates (ARB)
- lattices, sparse and dense linear algebra
- class field theory

Nemo/Hecke features

Advanced features:

- Class and unit groups
- absolute and relative fields
- certified conjugates (ARB)
- lattices, sparse and dense linear algebra
- class field theory
- abelian groups

Nemo/Hecke features

Advanced features:

- Class and unit groups
- absolute and relative fields
- certified conjugates (ARB)
- lattices, sparse and dense linear algebra
- class field theory
- abelian groups
- associative algebras

Nemo/Hecke features

Advanced features:

- Class and unit groups
- absolute and relative fields
- certified conjugates (ARB)
- lattices, sparse and dense linear algebra
- class field theory
- abelian groups
- associative algebras
- elliptic curves

Extension fields

Sircana: Computing extensions of $\mathbb{Q}$ with certain Galois groups

Extension fields

Sircana: Computing extensions of $\mathbb{Q}$ with certain Galois groups

- Starts with a (permutation) Group G from GAP

Extension fields

Sircana: Computing extensions of $\mathbb{Q}$ with certain Galois groups

- Starts with a (permutation) Group G from GAP
- Compute derived series of Group using GAP

Extension fields

Sircana: Computing extensions of $\mathbb{Q}$ with certain Galois groups

- Starts with a (permutation) Group G from GAP
- Compute derived series of Group using GAP
- Use groups in series to construct extension fields in Hecke

Extension fields

Sircana: Computing extensions of $\mathbb{Q}$ with certain Galois groups

- Starts with a (permutation) Group G from GAP
- Compute derived series of Group using GAP
- Use groups in series to construct extension fields in Hecke

Benefits from the integration of GAP group theory into Julia!

GIT Fans

Böhm, Breuer, Gutsche, Paffenholz, Ren: Computing GIT fans

GIT Fans

Böhm, Breuer, Gutsche, Paffenholz, Ren: Computing GIT fans

- Needs matrices from Nemo

GIT Fans

Böhm, Breuer, Gutsche, Paffenholz, Ren: Computing GIT fans

- Needs matrices from Nemo
  which are operated on by GAP permutation groups.

GIT Fans

Böhm, Breuer, Gutsche, Paffenholz, Ren: Computing GIT fans

- Needs matrices from Nemo
  which are operated on by GAP permutation groups.
- The orbits are used to compute Singular ideals for testing

GIT Fans

Böhm, Breuer, Gutsche, Paffenholz, Ren: Computing GIT fans

- Needs matrices from Nemo
  which are operated on by GAP permutation groups.
- The orbits are used to compute Singular ideals for testing
  which orbits should be turned into Polymake cones.

GIT Fans

Böhm, Breuer, Gutsche, Paffenholz, Ren: Computing GIT fans

- Needs matrices from Nemo
  which are operated on by GAP permutation groups.
- The orbits are used to compute Singular ideals for testing which orbits should be turned into Polymake cones.
- The cones and the Group are used to compute the GIT fan which is stored in a specific Julia data structure.

GIT Fans

Böhm, Breuer, Gutsche, Paffenholz, Ren: Computing GIT fans

- Needs matrices from Nemo
  which are operated on by GAP permutation groups.
- The orbits are used to compute Singular ideals for testing
  which orbits should be turned into Polymake cones.
- The cones and the Group are used to compute the GIT fan which is stored in a specific Julia data structure.

All cornerstones are used!

Loewy structure of $A(q, n, e)$

Breuer: Compute the Loewy structure of the Singer algebra $A(q, n, e)$

Loewy structure of $A(q, n, e)$

Breuer: Compute the Loewy structure of the Singer algebra $A(q, n, e)$

- For finding and testing conjectures, one has to look at many examples.

Loewy structure of $A(q, n, e)$

Breuer: Compute the Loewy structure of the Singer algebra $A(q, n, e)$

- For finding and testing conjectures, one has to look at many examples.
- Here the focus changes from algebraic computations to combinatorial ones (big loops).

Loewy structure of $A(q, n, e)$

Breuer: Compute the Loewy structure of the Singer algebra $A(q, n, e)$

- For finding and testing conjectures, one has to look at many examples.
- Here the focus changes from algebraic computations to combinatorial ones (big loops).
- With GAPJulia, Thomas could write code to solve combinatorial problems in Julia, and call it within his GAP session

Loewy structure of $A(q, n, e)$

Breuer: Compute the Loewy structure of the Singer algebra $A(q, n, e)$

- For finding and testing conjectures, one has to look at many examples.
- Here the focus changes from algebraic computations to combinatorial ones (big loops).
- With GAPJulia, Thomas could write code to solve combinatorial problems in Julia, and call it within his GAP session
- Contrary to GAP code, Julia code achieves similar performance as C, but has richer data structures (e.g., iterators)

Loewy structure of $A(q, n, e)$

Breuer: Compute the Loewy structure of the Singer algebra $A(q, n, e)$

- For finding and testing conjectures, one has to look at many examples.
- Here the focus changes from algebraic computations to combinatorial ones (big loops).
- With GAPJulia, Thomas could write code to solve combinatorial problems in Julia, and call it within his GAP session
- Contrary to GAP code, Julia code achieves similar performance as C, but has richer data structures (e.g., iterators)
- The tendency is to move more functionality (auxiliary functions, caches) from GAP to Julia.

Loewy structure of $A(q, n, e)$

Breuer: Compute the Loewy structure of the Singer algebra $A(q, n, e)$

- For finding and testing conjectures, one has to look at many examples.
- Here the focus changes from algebraic computations to combinatorial ones (big loops).
- With GAPJulia, Thomas could write code to solve combinatorial problems in Julia, and call it within his GAP session
- Contrary to GAP code, Julia code achieves similar performance as C, but has richer data structures (e.g., iterators)
- The tendency is to move more functionality (auxiliary functions, caches) from GAP to Julia.

Loewy structure of $A(q, n, e)$

Breuer: Compute the Loewy structure of the Singer algebra $A(q, n, e)$

- For finding and testing conjectures, one has to look at many examples.
- Here the focus changes from algebraic computations to combinatorial ones (big loops).
- With GAPJulia, Thomas could write code to solve combinatorial problems in Julia, and call it within his GAP session
- Contrary to GAP code, Julia code achieves similar performance as C, but has richer data structures (e.g., iterators)
- The tendency is to move more functionality (auxiliary functions, caches) from GAP to Julia.

So using Julia can be used to speed up GAP computations.

The key of OSCAR: Consistent mathematical model

- The most important feature of a CAS: Composability!

The key of OSCAR: Consistent mathematical model

- The most important feature of a CAS: Composability!
- Composability means that constructed data can be used as input for all applicable functions

The key of OSCAR: Consistent mathematical model

- The most important feature of a CAS: Composability!
- Composability means that constructed data can be used as input for all applicable functions
- To achieve that, rigorous interfaces need to be defined, and "mathematical data structures" need to be designed

The key of OSCAR: Consistent mathematical model

- The most important feature of a CAS: Composability!
- Composability means that constructed data can be used as input for all applicable functions
- To achieve that, rigorous interfaces need to be defined, and "mathematical data structures" need to be designed
- So a main goal of OSCAR is providing well-defined, rigorous, and compatible data structures on top of the cornerstone interfaces

OscarPolytope.jl

OscarPolytope is the OSCAR component that defines convex geometry objects

- OscarPolytope.jl provides data structures for Polytopes

OscarPolytope.jl

OscarPolytope is the OSCAR component that defines convex geometry objects

- OscarPolytope.jl provides data structures for Polytopes
- Inspired by the needs of users:
  - Polytopes are modeled on top of Polymake.jl, but with inhomogeneous coordinates (more natural)

OscarPolytope.jl

OscarPolytope is the OSCAR component that defines convex geometry objects

- OscarPolytope.jl provides data structures for Polytopes
- Inspired by the needs of users:
  - Polytopes are modeled on top of Polymake.jl, but with inhomogeneous coordinates (more natural)
  - One can compute lattice points, Hilbert Bases, and ILP solutions

OscarPolytope.jl

OscarPolytope is the OSCAR component that defines convex geometry objects

- OscarPolytope.jl provides data structures for Polytopes
- Inspired by the needs of users:
  - Polytopes are modeled on top of Polymake.jl, but with inhomogeneous coordinates (more natural)
  - One can compute lattice points, Hilbert Bases, and ILP solutions
  - Everything is translated into "natural" coordinates

Oscar.jl

Besides being a module that loads the OSCAR components, Oscar.jl defines data structures for commutative algebra

Oscar.jl

Besides being a module that loads the OSCAR components, Oscar.jl defines data structures for commutative algebra

- Ideals that behave like ideals, not like lists of polynomials,

Oscar.jl

Besides being a module that loads the OSCAR components, Oscar.jl defines data structures for commutative algebra

- Ideals that behave like ideals, not like lists of polynomials, (e.g., set-theoretic equality instead of lists of polynomials)

Oscar.jl

Besides being a module that loads the OSCAR components, Oscar.jl defines data structures for commutative algebra

- Ideals that behave like ideals, not like lists of polynomials, (e.g., set-theoretic equality instead of lists of polynomials)
- Soon: Interfaces for modules

Oscar.jl

Besides being a module that loads the OSCAR components, Oscar.jl defines data structures for commutative algebra

- Ideals that behave like ideals, not like lists of polynomials, (e.g., set-theoretic equality instead of lists of polynomials)
- Soon: Interfaces for modules
    - Defined mathematical operations, e.g., `isfree` or `istorsionfree`

Oscar.jl

Besides being a module that loads the OSCAR components, Oscar.jl defines data structures for commutative algebra

- Ideals that behave like ideals, not like lists of polynomials, (e.g., set-theoretic equality instead of lists of polynomials)
- Soon: Interfaces for modules
    - Defined mathematical operations, e.g., `isfree` or `istorsionfree`
    - Several data structures: Quotients, subquotients, Dedekind modules

Oscar.jl

Besides being a module that loads the OSCAR components, Oscar.jl defines data structures for commutative algebra

- Ideals that behave like ideals, not like lists of polynomials, (e.g., set-theoretic equality instead of lists of polynomials)
- Soon: Interfaces for modules
  - Defined mathematical operations, e.g., `isfree` or `istorsionfree`
  - Several data structures: Quotients, subquotients, Dedekind modules
  - Mathematical objects, not only collections of matrices

Key feature: User interface

Another key feature in the modern computing environment is accessibility:

Key feature: User interface

Another key feature in the modern computing environment is accessibility:

- Graphical user interfaces lower the entry barrier for users

Key feature: User interface

Another key feature in the modern computing environment is accessibility:

- Graphical user interfaces lower the entry barrier for users
- Currently, all systems can be used from Jupyter, so we get the infrastructure for free

Key feature: User interface

Another key feature in the modern computing environment is accessibility:

- Graphical user interfaces lower the entry barrier for users
- Currently, all systems can be used from Jupyter, so we get the infrastructure for free
- Jupyter allows for rich output (e.g, formulas displayed nicely, foldable coefficients or matrices, graphics)

Key feature: User interface

Another key feature in the modern computing environment is accessibility:

- Graphical user interfaces lower the entry barrier for users
- Currently, all systems can be used from Jupyter, so we get the infrastructure for free
- Jupyter allows for rich output (e.g, formulas displayed nicely, foldable coefficients or matrices, graphics)
- In both Julia and GAP we can directly bring our outputs into a rich displayed shape

Key feature: User interface

Another key feature in the modern computing environment is accessibility:

- Graphical user interfaces lower the entry barrier for users
- Currently, all systems can be used from Jupyter, so we get the infrastructure for free
- Jupyter allows for rich output (e.g, formulas displayed nicely, foldable coefficients or matrices, graphics)
- In both Julia and GAP we can directly bring our outputs into a rich displayed shape

So OSCAR will come fully equipped with a graphical interface!

The cube

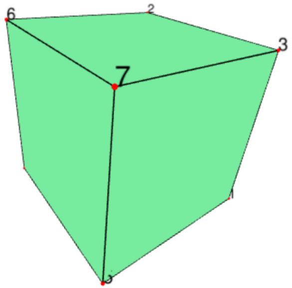
A notebook about Sebastians favorite polymake command

In [3]:
```
c = polytope.cube(3);
```

`c` is a cube in $\mathbb{R}^3$

In [7]:
```
polytope.VISUAL(c)
```

Out[7]:



Future plans

Near future

- Explore the possibilities of the Julia type system

Future plans

Near future

- Explore the possibilities of the Julia type system
- Create interfaces and implementation for many mathematical objects

Future plans

Near future

- Explore the possibilities of the Julia type system
- Create interfaces and implementation for many mathematical objects
- Stabilize the interfaces more

Future plans

Near future

- Explore the possibilities of the Julia type system
- Create interfaces and implementation for many mathematical objects
- Stabilize the interfaces more

Then

- Gather language features that are needed, and explore possibilities of GAP and Julia

Future plans

Near future

- Explore the possibilities of the Julia type system
- Create interfaces and implementation for many mathematical objects
- Stabilize the interfaces more

Then

- Gather language features that are needed, and explore possibilities of GAP and Julia
- Implement higher mathematical algorithms

Future plans

Near future

- Explore the possibilities of the Julia type system
- Create interfaces and implementation for many mathematical objects
- Stabilize the interfaces more

Then

- Gather language features that are needed, and explore possibilities of GAP and Julia
- Implement higher mathematical algorithms
- Provide releases and training!

Future plans

Near future

- Explore the possibilities of the Julia type system
- Create interfaces and implementation for many mathematical objects
- Stabilize the interfaces more

Then

- Gather language features that are needed, and explore possibilities of GAP and Julia
- Implement higher mathematical algorithms
- Provide releases and training!
- ...