OSCAR: An introduction

Max Horn (RPTU Kaiserslautern-Landau)

mhorn@rptu.de

What is OSCAR?

OSCAR is a new **O**pen **S**ource **C**omputer **A**lgebra **R**esearch system written in Julia, developed as central software project of the SFB-TRR 195 *Symbolic Tools in Mathematics and their Application* funded by the German Research Foundation (DFG). OSCAR is a collaborative effort with many contributors, and currently lead by Wolfram Decker, Claus Fieker, Michael Joswig and myself. The full source code is available at

https://www.oscar-system.org/

Cornerstones
OSCAR is built on *four cornerstones*, each providing state of the art capabilities in different domains:

- ANTIC (Nemo and Hecke): number theory
- GAP: group and representation theory
- Polymake: polyhedral and tropical geometry
- Singular: commutative and non-commutative algebra, algebraic geometry

These are not merely software packages used by OSCAR, rather they are *integral components* whose development is interlocked with that of OSCAR.

OSCAR relies on further components such as Arb, Calcium, FLINT, msolve and more. Yet it is more than just the sum of its components: it fuses them together into a comprehensive computer algebra system with a consistent user interface driven by mathematics only. The intention is to provide users with high-level objects closely paralleling what you might find in a text book. As an example, you can work with schemes and varieties without touching Gröbner bases.

Julia
The cornerstones are tied together by a ton of new code written in the Julia [1] programming language. Julia is relatively young, first appearing in the public in 2012. Originally developed at MIT for fast numerical mathematics, it enjoys a rapidly growing user base in many mathematical disciplines. Thanks to a just-in-time compiler (JIT) it can be used to produce highly efficient code that is competitive with pure C code, while at the same time being a high-level safe programming language. This addresses the “two language problem”: instead of writing most code in one (high-level) language (e.g. Python, Perl), but performance critical parts in another (e.g. C, C++, Fortran), all code can be written in the same language. Its excellent support for calling into existing C/C++ code was critical for developing the bridges to GAP, Polymake and Singular (ANTIC is written in Julia and thus needs no interfacing). Julia has many other compelling features, such as a REPL (read-eval-print loop, i.e., one can use it interactively with a prompt, multi-line editing, persistent history, tab completion, etc.), its package manager, or its support for multiple dispatch. There are also high quality interfaces to Python, R and other systems should one need to call code in these systems.

One downside of Julia’s JIT compiler is that the first time a function is executed, there can be a significant delay while OSCAR is being compiled. However, the Julia developers have made great strides towards improving this in the past few years, and we are confident this will improve further.

Installing

To install OSCAR, the following steps are required:

1. Install a C/C++ compiler.
2. Install Julia version 1.6 or later[1].
3. Start Julia and execute the following:

```
julia> using Pkg
julia> Pkg.add("Oscar")
```

[1] available from https://julialang.org/; avoid installing it .via apt-get or similar package managers

On Windows, you have to perform these steps in the *Windows Subsystem for Linux* (WSL).[2] Detailed installation instructions for all platforms are available at

`https://oscar-system.org/install/`

Now, you can use `OSCAR` in a `Julia` session:

```
julia> using Oscar
[ ... a banner is printed ... ]
```

The `OSCAR` team is working on providing a website on which one can try out `OSCAR` without installation.

Documentation and a warning

Below I will give a few short examples that showcase how `OSCAR` allows combining the cornerstones. However I will only scratch the surface of what the system can do, and the selection here is doubtlessly biased by my own background as a group theorist. To get a better impression of the existing capabilities and how to use them, please take a look at the reference manual at

`https://docs.oscar-system.org/`

One more caveat: The following examples were done using the current development version of `OSCAR`, which will eventually become version 0.12.0. Hopefully that version will have been released by the time you are reading this. Be advised that some examples might fail with older `OSCAR` versions.

A quick tour across OSCAR

We start our tour by creating a matrix representation of the dihedral group of order 12 over a number field.

```
F, t = quadratic_field(3)
a = diagonal_matrix(F.([1, -1, -1]))
b = matrix(F, [1//2 -t//2 0
               t//2  1//2 0
                  0     0 1])
G = matrix_group(a, b)
```

The field F in this example is provided by `ANTIC`, but matrix groups are handled by `GAP` in the background. `OSCAR` takes care of all necessary wrapping and conversions. Consequently we can forget about the nitty-gritty implementation details and focus on what we really want to do instead. For instance, we can verify that the group G as defined above is indeed a dihedral group of order 12.

```
julia> order(G)
12

julia> describe(G)
"D12"
```

At this point, we may wish to understand the group action a bit better by visualizing it. Here we take the orbit of a vector in F^3 and visualize its convex hull with some help by `Polymake`.

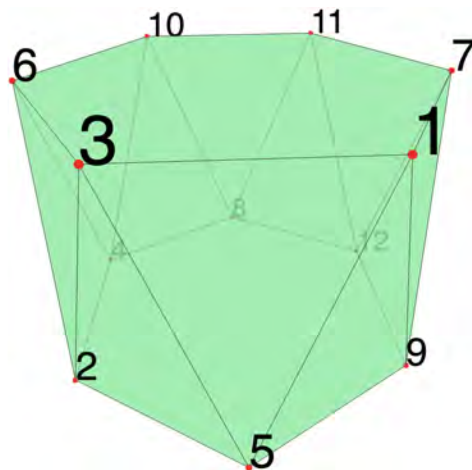
```
julia> o = orbit(G, *, F.([1,1,1]));

julia> vert = matrix(F, collect(o));

julia> ph = convex_hull(vert)
Polyhedron in ambient dimension 3

julia> visualize(ph)
```

This opens a web browser window with an interactive rendering of the polyhedron. It looks like this:



We can then compute the full combinatorial automorphism group of this polytope, given by its action on the vertices. Naturally it contains (an isomorphic copy of) our starting group G, but it got a bit bigger (the vertex labels are shifted by 1 here).

```
julia> aut = automorphism_group(ph;
                 action=:on_vertices);

julia> small_generating_set(aut)
3-element Vector{PermGroupElem}:
 (1,3,6,10,11,7)(2,4,8,12,9,5)
 (2,12)(3,7)(4,8)(5,9)(6,11)
 (1,2)(3,5)(4,7)(6,9)(8,11)(10,12)

julia> describe(aut)
"D24"
```

Naturally we can also study other actions, e.g. the one induced naturally on the polynomial ring $F[x, y, z]$.

[2]see `https://learn.microsoft.com/en-us/windows/wsl/install`

```
julia> R, (x,y,z) = F["x", "y", "z"];

julia> f = x^2 + 3y;

julia> f^gen(G,1)
x^2 - 3*y

julia> o = orbit(G, ^, f); sum(o)
3*x^2 + 3*y^2
```

Thanks to Singular, the full power of computational commutative algebra is available at our fingertips. As with everything else, we are just scratching the tip of the iceberg with the following example: we compute the ideal generated by the orbit (the collect is necessary because the orbit is just an iterator). Then we can quotient that ideal out and determine the Krull dimension of the quotient. Again, OSCAR seamlessly translates data for us as necessary in the background.

```
julia> I = ideal(collect(o));

julia> A, proj_hom = quo(R, I);

julia> dim(A)
1
```

A classical topic that brings together group theory and commutative algebra is invariant theory. OSCAR has extensive support for this. Here is a quick example.

```
julia> IR = invariant_ring(G);

julia> fundamental_invariants(IR)
4-element Vector:
 x3^2
 x1^2 + x2^2
 x1^6 + 5*x1^4*x2^2 + 5//3*x1^2*x2^4
   + 11//9*x2^6
 x1^5*x2*x3 - 10//3*x1^3*x2^3*x3
   + x1*x2^5*x3
```

Each of the cornerstones is a powerful tool on its own, but of course there are many cross-cutting applications that really need the expertise of multiple or all of them. The invariant theory example above is just one such instance. Another is described in the article “Toric Geometry in OSCAR” on page 20. Yet another is our state of the art implementation of Galois group computations over number fields, function fields and more.

```
R, x = QQ["x"];
f = x^6 - 366x^4 - 878x^3 + 4329x^2
                  + 14874x + 10471;
g, ctx = galois_group(f);
```

The result is a permutation group g together with a context object how precisely it acts as Galois group. We can of course treat g like any other group, but we can also e.g. obtain a p-adic approximation of the roots. OSCAR automatically picks a suitable prime for this, here $p = 13$. It is also possible to obtain complex approximations for teaching, but for practical applications the p-adic approach is superior.

```
julia> describe(g)
"C3 x S3"

julia> roots(ctx, 5)[1]
13^0 + 2*13^1 + 4*13^2 + 2*13^3
      + 7*13^4 + O(13^5)
```

Deep dive: the low-level interfaces

As mentioned, if necessary users can directly access functionality of the cornerstones. Let me emphasize that this is intended as a last resort. To use these low-level interfaces, you usually need some expertise in the corresponding cornerstone, and things become more tedious. If at all possible, we prefer to provide a “nice” high-level interface. So if something is missing, please let us know (see the final section to learn how).

But realistically, GAP, Polymake and Singular together with their many extension packages provide a vast reservoir in capabilities in many highly specialized domains, and wrapping them all with all their intricate details will take approximately forever. Thus it is crucial that there is an escape hatch that allows users to reach these capabilities if they need to.

For GAP you can evaluate arbitrary GAP code using GAP.evalstr. E.g. we do not (yet!) have Lie algebras in OSCAR, but we can get them from GAP:

```
julia> L = GAP.evalstr("""
           SimpleLieAlgebra("A", 2,
           Rationals)""")
GAP: <Lie algebra of dimension 10 ...>
```

This approach makes it cumbersome to pass around data. But direct access to GAP variables and functions is possible.

```
julia> R = GAP.Globals.RootSystem(L)
GAP: <root system of rank 2>
```

In principle, all packages of the GAP distribution are available. However, those packages that require compilation require some extra care. Please consult the GAP.jl documentation for more details.[3] We will address this limitation in a future release.

[3] see https://oscar-system.github.io/GAP.jl/stable/packages

```
julia> GAP.Packages.load("sla")
true

julia> GAP.Globals.
         WeylGroupAsPermGroup(R)
GAP: Group([ (1,4)(2,3)(5,6),
             (1,3)(2,5)(4,6) ])
```

Lastly, die-hard GAP fans can even get a fully functional GAP prompt from within OSCAR, which of course also provides full access to all Julia objects.

```
julia> GAP.prompt()

gap> IsLieNilpotent(Julia.L);
false
gap> quit;  # return to Julia
```

Similarly, there is a Polymake prompt that can be reached by pressing the $ key at the start of a Julia prompt; exit it by pressing the backspace key.

```
julia> Polymake.Shell.j = 42
42

common > print($j);
42
common > $c = polytope::cube(3);

julia> C = Polymake.Shell.c
name: c
type: Polytope<Rational>
description: cube of dimension 3
[...]
```

By the way, with the Polymake prompt we can conveniently reproduce the examples from the article "Hands-on Tropical Geometry" on page 10.

Programmatic access from Julia is also possible.

```
julia> Polymake.polytope.dim(C)
3

julia> C.F_VECTOR
pm::Vector<pm::Integer>
8 12 6
```

Note that in this particular example, it would be better to use OSCAR's high-level cube and dim methods.

Finally, for Singular we do not offer access to a REPL, but one can call into it as needed, and e.g. invoke library functions. For example, Singular.LibDeRham.deRhamCohomology is a wrapper for the function deRhamCohomology from deRham.lib. However the details for this interface are a bit more complicated and would require a lengthier discussion, so we omit them here – suffice it to say that access *is* possible.

That completes our quick tour. Once more let me recommend the article "Toric Geometry in OSCAR" on page 20 for a proper mathematical example. There have also been prior articles in the Rundbrief on some OSCAR components [3, 4] which are still quite relevant. Finally, we are currently preparing a book about OSCAR [2].

We want *you!*

We are eager to hear from you if you are using OSCAR or are generally interested in it. We constantly strive to improve OSCAR, and as such look forward to bug reports, feature request, and code contributions. Users of all experience levels are welcome.

If you are experimenting with OSCAR and have questions, I heartily recommend our Slack chat, where many people participate and provide answers. You can join via

https://oscar-system.org/slack

We also have a mailing list, a GitHub discussion forum and more, see

https://oscar-system.org/community/

for details. It also explains how to best report issues (spoiler: via our issue tracker). And if you prefer personal communication, don't hesitate to reach out to me personally and e.g. email me at mhorn@rptu.de.

Last but not least, you can get the latest source code from our Git repository at

https://github.com/oscar-system/Oscar.jl

Acknowledgement

This work was supported by the SFB-TRR 195 *Symbolic Tools in Mathematics and their Application* of the German Research Foundation (DFG).

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM review **59** (2017), no. 1, 65–98.

[2] W. Decker, C. Eder, C. Fieker, M. Horn, M. Joswig, *The OSCAR book*, Springer, 2024.

[3] C. Fieker, C. Sircana, T. Hofmann, *Hecke: A Number Theory Package*, Computeralgebra Rundbrief **66** (2020), 12–14.

[4] W. Hart, *ANTIC: Algebraic Number Theory in C*, Computeralgebra Rundbrief **56** (2015), 10–11.