

— Manual v0.32 —

Benjamin Jurke and Thorsten Rahn

in collaboration with

Ralph Blumenhagen, Helmut Roschy

October 13, 2026

Contact info

Please send all C/C++ implementation (**cohomCalg** core program) related or other technical questions to Benjamin Jurke, reachable via eMail: mail@benjaminjurke.com.

For all questions regarding the **cohomCalg Koszul** extension please refer to Thorsten Rahn (thorsten.rahn@gmail.com).

Introduction

cohomCalg (an acronym pronounced "cohom-calc") is an implementation of the sheaf cohomology computation algorithm for line bundles on toric spaces presented in detail in [1] and subsequently proven in [2, 3]. Several applications are discussed in [4, 5]. The program is freely available under the GNU General Public Licence version 3.0 (GPLv3) on the repository site

https://github.com/BenjaminJurke/cohomCalg.

The main program is written in C++ and consists of a single main program executable file. Due to the implementation structure of the program and the ever-increasing spreading of 64-bit capable processors and operating systems, we consider the 64-bit version of the application to be the natural choice with the 32-bit version provided only as a fallback option for legacy computers. Furthermore, the 64-bit version is considerably faster compared to the 32-bit version on the same platform.

The Koszul extension of the **cohomCalg** package is a *Mathematica 7* and *8* script and consists of a single file as well, but depends of the main program's executable.

Note: Throughout this manual we assume some basic knowledge of toric geometry. A concise introduction to the subject can be found in [1].

Acknowledgement: Thanks to Doug Torrance for several improvements and reorganizing **cohomCalg** into a Debian package.

External dependencies: The **cohomCalg** package makes use of the following external libraries:

- The Polyhedral Library (`PolyLib`); GPLv3 license; mathematical library for computations involving polyhedra, used in the counting of monomials; available for download at http://icps.u-strasbg.fr/polylib/

Contents

1 Installation Guidelines **4**
- 1.1 Main program . . . 4
- 1.2 Koszul extension . . . 4
- 1.3 Package contents . . . 5
- 1.4 Version history . . . 5

I Main cohomCalg program **6**

2 Usage **6**
- 2.1 Command line parameters . . . 6
- 2.2 Input format and structure . . . 9
- 2.3 Output format . . . 10
- 2.4 Obtaining geometric data . . . 12

3 Implementation **13**
- 3.1 Internal design . . . 13
- 3.2 Potential improvements . . . 14
- 3.3 Known problems . . . 14

II cohomCalg Koszul extension **16**

4 Usage **16**
- 4.1 Preparations . . . 16
- 4.2 Input format and structure . . . 16
- 4.3 Bundle types . . . 17
  - 4.3.1 Line bundles . . . 17
  - 4.3.2 Equivariant Cohomology of Line Bundles . . . 18
  - 4.3.3 Tangent bundle of a subvariety . . . 19
  - 4.3.4 Λ^k(Cotangent Bundle) of a subvariety for $k = 0, 1, 2$ . . . 19
  - 4.3.5 Λ^k(Monad bundle) of a subvariety for $k = 1, 2$ . . . 20
  - 4.3.6 Hodge diamond of a subvariety . . . 21
  - 4.3.7 Endomorphism bundle of the tangent bundle of a subvariety 22
  - 4.3.8 Endomorphism Bundle of the monad bundle of a Subvariety 22
- 4.4 Verbose Level and Output Format . . . 23

1 Installation Guidelines

1.1 Main program

Windows platform

The package comes with precompiled 64-bit and 32-bit binaries called *cohomcalg.exe* or *cohomcalg32.exe* in the '`/bin/`' subdirectory for Microsoft Windows, which are statically linked to all required run-time libraries in order to avoid any troubles you might have running them. No further steps are required.

Linux/Unix and Mac

On a Linux/Unix or MacOS X-platform you have to compile the binaries from the source code yourself, which is a fully automated process. Providing precompiled binaries would enlarge the package unnecessarily and not make use of platform-specific speedups. In order to compile **cohomCalg** from source under a Unix-based platform (e.g. Linux or MacOS X), just open a terminal, change into the **cohomCalg** directory and use the

```
make
```

command. Now the script will compile the source files, link them together and produce the new executable *cohomcalg* (without any extension) in the '`/bin/`' subdirectory, where the shipped Windows binaries are found as well. Using

```
make clean
```

deletes the temporarily generated subdirectory '`/build/`' which contains the intermediate object files from the compilation process.

General remarks

Unfortunately, there is no such automated build process available for the Microsoft Windows environment. The program code is fully cross-platform compatible, provided a few minor modifications (a limited number of preprocessor definitions in the source header *platform.h*) are applied for other platforms.

1.2 Koszul extension

The Koszul extension of the **cohomCalg** package requires a running copy of *Mathematica 7* or *8*.[1] Simply open the *Mathematica* notebook file from the program

[1] The script does not use any commands special to Version 7 or 8 of *Mathematica*, i.e. in principle there are no obstacles to using an older version. However, lacking such older software, we were unable to test such usage. Please understand that we cannot offer any help regarding older versions of *Mathematica*.

and make sure that the **cohomCalg** main program is in the same directory. Right in the beginning of the notebook file you find a variable containing the name of the executable file that is used by the script. Change this variable to the correct name of the main executable, e.g. `cohomcalg.exe` or `cohomcalg32.exe`. Then save the file for future usage.

1.3 Package contents

Naturally, you can find the pre-compiled binaries in the directory '`/bin/`' of the package. The Windows binaries *cohomcalg.exe* and *cohomcalg32.exe* can be distinguished for the Unix/Linux/MacOS X binary *cohomcalg* by the file extension '*.exe*', which is not required to be typed explicitly in the Windows command prompt — Windows only considers files with certain extensions to be executable. The binaries directory also contains the *Mathematica 7* or *8* notebook file of the Koszul extension.

Under '`/source/`' the source code of **cohomCalg** is found. It also includes the modified version of the `PolyLib` under '`/source/polylib_mod/`', which was stripped down to all necessary source files required for compilation.

1.4 Version history

Like most other open source projects, we started **cohomCalg** with a version number below v1.0 in order to reflect ongoing development before reaching a suitable level of stability. Each increase of the first digit after the dot reflects a major upgrade, the second digit is used for medium changes that warrant a manual upgrade. Furthermore, a lower-case letter may be added (like in v0.21d) to reflect minor internal bug-fixing or upgrades that do not require any change on the user's side.

v0.31 (May 25, 2011): Routines for discrete group actions, multi-core support, significant performance improvements.

v0.21 (October 18, 2010): **cohomCalg Koszul** extension added.

v0.13 (July 23, 2010): Integration mode added.

v0.12 (June 25, 2010): Several minor bugfixes and improvements.

v0.11 (May 4, 2010): Original release of the **cohomCalg** C++ implementation.

v0.04 (March 29, 2010): Original public release of the *Mathematica 7* script, which is still available from the website.

Part I
Main **cohomCalg** program

2 Usage

2.1 Command line parameters

The program can be entirely controlled directly from the terminal. However, due to the usually somewhat lengthy input and output data, it is strongly suggested to use text input files. We make it a habit to append the extension *.in* to such input files, e.g. *dP3.in*, however you are free to choose whatever name you like. Furthermore, it is recommended to redirect the program's standard output to a file via '`> dP3.out`'. The most basic command line therefore takes the form

```
cohomcalg dP3.in > dP3.out
```

and effective uses the data in *dP3.in* for input and prints the output into *dP3.out*. Instead of using an input file, you can also use the option '`-in="..."`', where everything between the quotation marks is treated exactly like the data from an input file.

All command line options are always specified before the input file name, i.e. the general command line syntax is

```
cohomcalg [-option1] [...]  [-optionN] [inputfile] [> outputfile]
```

and calling the program without any additional parameters shows a concise overview of all options. You can specify any combination of options from the following list:

`-?`, `/?`, `-?`, `-help`, `/help`, `-help` Shows a concise list of command line options and the general syntax structure. The program will terminate after showing the list, so this option should not be paired with anything else.

`-in="..."` Treats everything between the quotation marks like input data from a file. You may use both an input file and the '`-in`' option, in which case the input data from the command line is effectively analyzed after the file contents. This is useful if you want to specify the geometry of a variety in a file and want to compute a number of different line bundle cohomologies subsequently without having to change the file each time.

`-nomonomfile` As the size of the Stanley-Reisner ideal powerset grows exponentially with the number of generators, traversing all elements to compute the secondary/remnant sequences is a rather time consuming process for

complicated geometries. Therefore, the program automatically looks for a file *inputfilename.monoms*, e.g. *dP3.in.monoms* in order to skip this computation. The filename can be changed using the '`monomialfile`' input command, see sec. 2.2. Furthermore, in case this file is missing or could not be read the program automatically saves the computed secondary sequences to this file. Using the '`-nomonomfile`' command line option deactivates the usage of this file and also deactivates the generation of a new file. If due to the sole usage of the '`-in`' option without an input file no filename has been specified via the '`monomialfile`' input command, the program automatically deactivates the usage of those intermediate files.

`-checkserre` When using this option, the program automatically computes the cohomology for the Serre-dual line bundle and compares the results according to $H^k(\mathcal{O}(D)) \cong H^{n-k}(K \otimes \mathcal{O}(-D))^*$. Note that this effectively doubles the computation time in the second part of the program.

`-noreduction` Deactivates the Serre-duality reduction, which tries to reduce the number of ambiguously contributing monomials by comparing to Serre-dual complement monomials. If this reduction is deactivated and ambiguous contributions are found, both the entire range of all possible "normal" and Serre-dual cohomologies has to be computed and compared in order to identify a consistent pair. Note that this requires huge amounts of memory and might easily lead to the program quitting prematurely, see sec. 3.3. Use with caution!

`-hideinput` Deactivates the formatted output of the entire input data. Remember that it might be rather useful to have the results along with the input data in the same file, in particular if you make usage of the '`-in`' option to supply additional input data.

`-showtime` Activates output of computation time statistics even for very short runtimes. This option is automatically activated if the computation takes longer than one second.

The following command line options are only useful for debugging or if you want to know a little bit more, what is happening on the inside:

`-showbits` Activates an explicit output of the bit-masks used internally.

`-mathematica` Outputs the input data and the output data in a form directly suitable for Copy&Paste into the legacy *Mathematica 7* script.

`-integrated` If the integrated mode is activated the output passed to the standard output channel `stdout` is always a single line of the form

`{True,`*Cohomology1*`,`*Cohomology2*`,...,`*CohomologyN*`}` or

`{False,"Invalid command line parameters"}`

The first boolean value tells if the application run was either entirely successful ("`True`") or if an error occured during the run ("`False`"). If an error happens, a brief error message is supplied. In case of a successful run, a listing of all computed cohomology group dimensions and some intermediate information follows in the order requested in the input file. The cohomology data is of the form

$$\mathit{CohomologyM} = \{\overbrace{\{0,2,0\}}^{\text{cohomology group dimensions } h^\bullet}, \overbrace{\{\underbrace{\{1,1*u1*u2*u5\}}_{u_1u_2u_5 \text{ contributes with factor 1 to cohomology } h^1}, \underbrace{\{0,2*u2*u3*u5\}}_{u_2u_3u_5 \text{ contributes with factor 2 to cohomology } h^0}\}}^{\text{list of contributing denominator monomials}}\},$$

i.e. the final cohomology group dimensions $h^\bullet(\mathcal{O}(D))$ are a list at the first position inside the cohomology output vector.

Note that this option overrides any verbose level. Together with the '`-in`' option this allows for simple integration of **cohomCalg** into external applications. It should, however, be emphasized that the output format of this option is subject to potential changes in future version.

`-verbose1`, ..., `-verbose5` Activates debug output, where a higher number produces more detailed information. The different levels are:

1) Shows the ultimately computed list of all contributing denominator monomials with factors determined via the secondary/remnant cohomology and the number of rational functions. As it might become internally necessary to use information from the Serre-dual cohomology, this prints a two-column list of this data for both the "normal" and Serre-dual contributions.
2) In order to reduce computation time and memory consumption, the ambiguously contributing denominator monomials are mapped to their complement monomials. This verbose level prints the list of all contributing monomials before and after the reduction.
3) Prints the reduced list of secondary/remnant sequences, where all obvious cases (exact sequences, or sequences containing just a single non-zero entry) are removed. Note that this output only appears if the secondary/remnant cohomology is actually computed, compare option `-nomonomfile`.
4) Prints the full list of secondary/remnant sequences. Note that this output only appears if the secondary/remnant cohomology is actually computed, compare option `-nomonomfile`.
5) Shows all polyhedron condition matrices which are passed to `PolyLib` in order to compute the number of rational functions for the corresponding denominator monomial.

In most cases, passing no options at all will serve you just fine, i.e. it prints the input data and the computed cohomology group dimensions.

2.2 Input format and structure

The input format is both the same for the input file and the input data passed via terminal using the '`-in`' command. Keep in mind, that the command line data is always considered after the data read from the input file.

The general structure of the input is in the form of some rather basic data commands. Each such command has to be terminated by a semicolon, just like C/C++ code. Other than keeping the right syntax structure, you are free to format your input data using spaces, tabulators or line breaks just as you like. You may also use comments in the input file using the '`%`' character. Everything following a '`%`' character till the end of the line is completely ignored. Note that this effectively prohibits usage of comments when passing data via the command line, as the entire remainder of the data after the percent sign will be ignored. The following commands for providing input data are available:

`vertex` [name] `| GLSM: (`[GLSM charge 1]`, ...,` [GLSM charge r]`);` This command specifies a new coordinate (or vertex of the fan in the corresponding toric description, thus the name) and its GLSM charges, i.e. for each coordinate you use one corresponding '`vertex`' command. The name of the coordinate must be an alphanumeric sequence of characters not starting with a number. The number of GLSM charges must be equal for all coordinates and the GLSM charge value has to stay within a certain range. As an example, the following commands specify the coordinates and GLSM charges of the del-Pezzo 1 surface:

```
vertex u1 | GLSM: ( 1, 0 );
vertex u2 | GLSM: ( 1, 0 );
vertex u3 | GLSM: ( 1, 1 );
vertex u4 | GLSM: ( 0, 1 );
```

Note that due to internal limitations the maximum number of vertices is limited to 63.

`srideal [`[SR generator 1]`,...,`[SR generator N]`];` This command specifies the Stanley-Reisner ideal, i.e. you have to specify the generators as products of the coordinates like in the following dP_1 example:

```
srideal [u1*u2, u3*u4];
```

Note that the coordinates used in the products have to be previously declared, i.e. it only makes sense to use the '`srideal`' command after the '`vertex`' commands. The number of Stanley-Reisner ideal generators N is

limited to 63, however, on modern desktops computational time will explode at around 40 generators, which is due to the exponential 2^N growth of the powerset of the Stanley-Reisner ideal generators.

`ambientcohom O(`[charge 1]`, ...,`[charge r]`);` Using this command you can specify the GLSM charges of the target divisor D that determines the line bundle $\mathcal{O}(D)$ for which the sheaf cohomology on the ambient space is computed. Please note the upper-case letter 'O' behind the command. For example,

```
ambientcohom O(0, 0);
```

computes the ambient space sheaf cohomology for the holomorphic line bundle $\mathcal{O}(0,0) = \mathcal{O}$. Obviously, the number of charges r has to be equal to the number of GLSM charges specified in the '`vertex`' commands. You may specify several target line bundles for batch computation by simply using the '`ambientcohom`' command multiple times.

`monomialfile "`[filename]`";` This command allows you to specify a different filename for the intermediate monomial file between the quotation marks:

```
monomialfile "my-dP1-monomial-file.dat";
```

This filename is both used for reading and saving of the monomial file. Please keep in mind, that the intermediate data saved to file is crucially dependant on the geometry data specified via the other commands, see sec. 3.3. You can therefore turn off the usage and generation of those files using the input command line

```
monomialfile off;
```

which might come in handy, when you frequently change the geometry of the variety specified in the input file and want to avoid the usage of the '`-nomonomfile`' parameter.

All commands are subject to a precise syntax, range and consistency check, so it is basically impossible to run the program with invalid data. In such cases, you will see a syntax error message indicating where the problem occured in the input data.

2.3 Output format

Following the prior dP_1 example, after saving the six input commands to file and running the program via '`cohomcalg dP1.in > dP1.out`', the output file contains after the obligatory header the following data:

```
Input data:
===========
    The described ambient space is of dimension 2.
    There are 4 coordinates, each having 2 GLSM charges:
        coord  1:   u1  |    1   0
        coord  2:   u2  |    1   0
        coord  3:   u3  |    1   1
        coord  4:   u4  |    0   1

    There are 2 generators of the Stanley-Reisner ideal:
        SRgen  1:  u1*u2
        SRgen  2:  u3*u4

    There is 1 ambient space sheaf cohomology requested:
        cohom  1:  H^i(A; O(   0,   0 ))

Cohomology dimensions:
======================
    dim H^i(A; O(   0,   0 )) = (   1,   0,   0 )
```

This output is fairly self-explanatory and should not require further comment. However, for more involved and complicated examples a couple of non-ideal things may happen. You may see the line

```
The Serre dualization reduction was unable to uniquely
resolve 88 of the original 1049 ambiguous monoms.
```

which indicates that the Serre-dual complement monomial reduction technique (compare the command line option '`-noreduction`') was unable to resolve all ambiguously contributing denominator monomials. This is in principle not a problem, provided that the much more computationally expensive fallback technique, which compares all possible cohomologies to the corresponding Serre dual cohomologies, is able to uniquely resolve the issue. However, if this fails as well you will see an output like

```
dim H^i(A; O(   2,   4,   1,  ...,   3,  -2,   1 )) is ambiguous
    candidate results are:    = ( 580, 353, 171,   1, 919,   0 )
                              = ( 580, 353,   0,   1,1090,   0 )
```

which gives you all possible cohomology configurations the algorithm was able to sort out. Unfortunately, you may also be graced by the message

```
dim H^i(A; O(   2,   ...,   1 )) could not be determined.
```

but in all our extensive testing, we never encountered this problem. Finally, for any computations longer than one second or if you use the '`-showtime`' option you will encounter runtime statistics like

```
Application run took 14.27 seconds, more precisely
    11.02 seconds sec for the computation of the sec. cohom.
    3.23 seconds sec for the counting of rational functions
```

which gives you an idea of the time consumption. Also, during longer runtimes, you will see an output like

```
SR powerset traverse 14.54% done (12 secs remaining)...
```

which reports the progress of computing the secondary/remnant cohomologies and gives a rough time estimate for this part of the program to finish.

2.4 Obtaining geometric data

Naturally, you need to get the input data from somewhere, and a couple of very useful packages for those tasks are available freely on the net. In order to derive the Stanley-Reisner ideal, which is a required input for the program, you may want to take a look at

`TOPCOM`: http://www.rambau.wm.uni-bayreuth.de/TOPCOM/,

which can also enumerate all possible fans for a given set of vertices. The Maple script package

`SCHUBERT`: http://folk.uib.no/nmasr/schubert/0.996/,

can be used to compute intersection numbers and further geometrical quantities of toric varieties, however, the software is somewhat dated at this point. Furthermore, there is the package

`PALP`: http://hep.itp.tuwien.ac.at/ kreuzer/CY/CYpalp.html,

which is useful for computing invariants of hypersurfaces, Mori cone vectors etc. You may also want to take a look at the

`SAGE` Library: http://www.sagemath.org/,

of freely available mathematical software. Finally, the environment

`Macaulay2`: http://www.math.uiuc.edu/Macaulay2/,

which allows similar computations, was heavily used during the development process.

3 Implementation

3.1 Internal design

This implementation heavily relies on the usage of bit-wise operations on 64-bit integer variables. The ith coordinate of the input data is represented by the ith bit of such a variable, such that a Stanley-Reisner ideal generator simply becomes a bit-mask. Via this coding scheme, computing the union of Stanley-Reisner ideal generators is equivalent to using the bit-wise non-exclusive OR operator, which is a superfast operation on any 64-bit architecture machine. Furthermore, when traversing the powerset of the Stanley-Reisner ideal, we also encode the presence of the jth generator by the jth bit, such that running through all subsets of the set of generators reduces to a simple loop running from 0 to 2^N, where N is the number of generators.

Using a couple of bit-mask buffers, those two elementary encoding techniques allowed us to gain a speedup of several orders of magnitude on any other implementation approach. On the other hand, this puts the strict upper limit of 63 on the number of coordinates and Stanley-Reisner ideal generators, but as the number of Stanley-Reisner ideal generators is usually much larger and somewhat maxes out current computers at around 40 generators, those constraints are currently rather theoretical and do not present any practical restrictions.

Since version 0.31 of the **cohomCalg** core program the computation of the secondary sequences is carried out in a multi-threaded fashion, i.e. all available (logical) processor cores of the machine are used simultaneously. Depending on the hardware, this leads to a significant speed improvement factor scaling almost linearly with the number of cores.

By computing the secondary sequences, a number of uniquely and ambiguously contributing denominator monomials is found. Ambiguously in this sense means that the secondary/remnant cohomology has more than one non-zero contribution, i.e. the number of rational functions for this denominator monomial might contribute to different groups of the cohomology, e.g. to H^1 and H^2. This ambiguity arises due to the fact that the mappings in the secondary/remnant sequences are currently not considered in the computation. It is therefore necessary to employ the Serre duality in order to resolve this issue, which basically means to take the complement monomial — the monomial consisting of all the remaining coordinates — into account. This step is carried out during the Serre-duality reduction, which can be turned off using the '`-noreduction`' command line option. In most cases, this eliminates all ambiguities by turning the ambiguous contributions in unique contributions.

In the next step, the number of rational functions is computed for each monomial using The Polyhedron Library.[2] Basically, with `PolyLib` we are constructing

[2]Note that we use a very slightly modified version of the `PolyLib`, where a number of standard output responses are removed. It is therefore not recommended to simply replace the

a polyhedron from a number of equalities and inqualities and then count the number of integer lattice points inside of it using Ehrhart polynomial approximations. For the uniquely determined contributions, the resulting cohomology group dimension is simply the factor derived from the secondary/remnant sequences times the number of rational functions. However, if there are still ambigous contributions leftover, this requires a branching of the current number of possible cohomology dimensions by the ambiguity, i.e. if you have 10 ambiguous contributions with 3 possibilities each, this already yields $3^{10} = 59049$ possible configurations. In case of ambiguities it is necessary to entirely compute all those possibilities both for the "normal" and Serre-dual configuration and then match those possibilities in order to determine compatible ones. Unfortunately, due to the rapid growth of the number of configurations, one quickly runs out of memory, which is a still unresolved issue of the implementation. Again, this problem ultimately originates in the fact, that the mappings in the secondary/remnant sequences are not implemented at the moment.

3.2 Potential improvements

The counting of monomials (rational functions) in the current algorithm is currently still implemented as a single-threaded application, i.e. it only makes use of one processor core—whereas modern desktops already may have up to 6 cores per processor with an increasing trend. While a parallelization has been attempted, it appears that the PolyLib is not implemented in a thread-safe fashion, which leads to a memory (heap) corruption as soon as more than one PolyLib thread is executed at the same time. This problem is currently investigated in more detail but may remain unsolved for the foreseeable future.

As a second improvement in the long run one might offer the option of arbitrary-length bitmasks, which would effectively remove the 63 coordinates and Stanley-Reisner generators upper limits. On the other hand, such a change would most likely impair performance quite a bit, and for the moment those limits are far out of reach for reasonable computation times.

3.3 Known problems

The implementation is rather robust aside from the potential memory-consumption issue, when the number of ambiguous contributions becomes too large. For the moment, we can only recommend the usage of the 64-bit version for such cases (which is recommended anyways), as it somewhat alleviates the problem and gives the program access to more memory. It should be mentioned however, that without enforcing this issue via the '`-noreduction`' option, the problem is mostly kept in check even in extremely complicated cases.

`PolyLib` version included in the source package by another version.

A somewhat minor problem is the instability of the monomial files. As the computed monomials are effectively stored via bit-masks, changing the geometry of the variety in the input data (i.e. the order or number of the vertices as well as the number of Stanley-Reisner generators) effectively corrupts this data. Currently, there is no sophisticated check if the intermediate monomial data stored in the file corresponds to the geometry specified via the input data. In worst case scenarios such a corruption might crash the program. In case you are worried about the results while using an intermediate monomial file try rebuilding this one by deleting or renaming the old one or turning off the monomial file usage alltogether.

Aside from that we are reasonably certain that quite a number of bugs and problems are still included. Please let us know any problems you may find, see page 1 for contact information.

Part II
cohomCalg Koszul extension

4 Usage

4.1 Preparations

The *Mathematica* Koszul extension serves as a wrapper for the **cohomCalg** program. Simply open *Mathematica 7* or *8* on your computer and load the notebook file. Make sure that the **cohomCalg** executable variable at the top of the script points to the correct executable file, e.g. *cohomcalg32.exe* on a Windows 32-bit platform. Also make sure that the correct path of the executable is inserted. There are no further parameters or options.

4.2 Input format and structure

The script in its current form consists of three parts:

1. The main portion of the file contains the actual routines used for the program. Each time you open the script file, you briefly have to re-run this part, such that the *Mathematica* kernel working in the background has all the necessary definitions etc. available. This is done, for example, by pressing `Shift+Enter` while the cursor is still in the upper part of the script.

2. The second portion is clearly marked by the words "`Ambient geometry data`". Here you have to specify the relevant toric data for the ambient space variety X. The following data is required in list form:

   - *Coordinates:* A list of the coordinate names like the seven variables in `{x1,x2,x3,x4,x5,x6,x7}`.
   - *Stanley-Reisner ideal:* A list of lists corresponding to the generators of the Stanley-Reisner ideal, e.g. `{{x1,x6},{x2,x3,x4,x5,x7}}`.
   - *GLSM relations:* A list of lists corresponding to the GLSM charges or projective relations associated to each coordinate, for example `{{1,3},{0,1},{0,1},{0,1},{0,1},{1,0},{0,1}}`. The number of those lists obviously has to be equal to the number of coordinates.

   This data is assigned to a list variable, which for book-keeping purposes should carry a name suggesting the described toric variety, e.g. `P111113`.

3. Finally the command part follows, where the user can access the functions provided by the **cohomCalg Koszul**extension. Here depending on what kind

of bundle one wants to calculate the program needs some additional input parameters:

- *Complete intersection charges:* A list of divisor charges for the complete intersection like `{{1,4},{1,4}}`. An empty list `{}` always returns the result for the ambient space.
- *Line bundle charges:* A list that specifies the required line bundle charges, e.g. `{22,30}`.
- *Monad bundle charges :* A list of lists corresponding to the monad bundle charges N_k in (1), for example `{{1,0},{0,1},{0,1},{0,1},{0,2}}`.
- *Monad bundle constraints:* A list of lists of the required monad bundle constraints M_i in (1), for instance `{{1,5}}`.
- V_r*:* Power of the $\mathcal{O}_S$ bundle in the monad sequence (1), e.g. `2`.

$$0 \longrightarrow \mathcal{O}_S^{\oplus V_r} \longrightarrow \bigoplus_{k=1}^{V_n} \mathcal{O}_S(N_k) \longrightarrow \bigoplus_{i=1}^{V_l} \mathcal{O}_S(M_i)\,, \tag{1}$$

- $\mathbb{Z}^n$*-action: A list that specifies how the involution acts on the coordinates, e.g.* `{x1,x2,x3,x4,x5,x6,-x7}` for a $\mathbb{Z}^2$ action

4.3 Bundle types

Various different bundle types are provided by the **cohomCalg Koszul** extension. The structure of the input is similar for all bundles and looks like:

`CohomologyOf[`[Bundle Type]`,` [Ambient Space]`,` [Additional Input]`,`
[Optional: Type of Subvariety]`,` [Optional: Verbose Level]`,`
[Optional: Output Type]`,` [Optional: Line Bundle Collector]`]`

The optional input can be inserted in an arbitrary order. In the following we describe how the different bundle types can be evaluated. Here we will not use the last two of the optional input parameters and explain their use afterwards. From the mandatory part of the input data only the [Bundle Type] and the [Additional Input] change:

4.3.1 Line bundles

- [Bundle Type]: `"LineBundle"`
- [Additional Input]: `{`*Complete intersection charges*, *Line bundle charges*`}`

This command is the main routine of the **cohomCalg Koszul** extension. Given an ambient space X and a set of divisors $S_1, \ldots, S_l \subset X$ as well as a line bundle

divisor E it computes the sheaf cohomology group dimensions $\dim H^i(S;\mathcal{O}_S(E))$ on the hypersurface or complete intersection $S = S_1 \cap \cdots \cap S_l$. Note that there is no checking if the intersection geometry is actually well-defined, i.e. the **cohomCalg Koszul** extension does not test for transversality or other conditions. Furthermore, it does not actually evaluate the mappings in the Koszul complex, but rather attempts to derive the dimensions from indirect exactness arguments and constraints. However, as those methods may not always lead to a solution, the output may contain parameters which are unresolved. The input data for the ambient space, the complete intersection and the line bundle charges has to be specified as explained above. Here the additional input is a list of two lists, namely the data specifying the complete intersection as well as the charges of the line bundle you want to compute. For the complete intersection data, an empty list `{}` of divisors return the cohomology of the ambient space itself, i.e. it computes $\dim H^i(X;\mathcal{O}_X(E))$.

Example: Consider the ambient space $\tilde{\mathbb{P}}^5_{111113}$ described before. If you type

```
␣P111113={
␣(*Coordinates*){x1,x2,x3,x4,x5,x6,x7},
␣(*Stanley␣Reisner*){{x1,x6},{x2,x3,x4,x5,x7}},
␣(*Equivalence␣Relations*){{1,3},{0,1},{0,1},{0,1},{0,1},{1,0},{0,1}}};

␣CohomologyOf["LineBundle",P111113,{{{1,4},{1,4}},{2,8}}]
␣
```

you will get the result `{487,0,0,0}`, which means that $\dim H^0(S;\mathcal{O}_S(E)) = 487$ and $\dim H^j(S;\mathcal{O}_S(E)) = 0$ for $j = 1, 2, 3$.

4.3.2 Equivariant Cohomology of Line Bundles

If you consider an ambient space that allows for a discrete action your cohomology obtains a grading into invariant and non-invariant parts under this action. You can compute this equivariant cohomology of line bundles. To do that the following input parameters are needed:

- [Bundle Type]: `"EquivariantLineBundle"`
- [Additional Input]: `{{},` *Line bundle charges*, $\mathbb{Z}^n$*-action*`}`

The program then returns two cohomology vectors where the first contains the invariant part of the cohomology, $\dim H^\bullet_{\text{inv}}(X;\mathcal{O}_X(E))$ and the non-invariant part of the cohomology $\dim H^\bullet_{\text{non-inv}}(X;\mathcal{O}_X(E))$. Note that the program does not check whether such an action is allowed on this space, so you have to figure that out yourself beforehand. You can also ask the program to print the representatives of the cohomology by using the verbose options described below.

Example: Consider the ambient space $\tilde{\mathbb{P}}^5_{111113}$ as specified above. If you evaluate the order

```
␣CohomologyOf[
(*Bundle␣specification*)"EquivariantLineBundle",
(*Ambient␣Space*)P111113,
(*Remaining␣Data:Complete␣Intersection,LineBundle␣Charges,Discrete␣Action*)
␣{{},{5,3},{x1␣x2,x3,x4,x5,x6,-x7}}
];
```

you will get the result two resulting line bundle cohomology vectors where the first one denotes the invariant part of the equivariant cohomology and the second one the non-invariant part. For this explicit example you will get `{{25,0,0,0,155,0},{11,0,0,0,250,0}}`. So we have the only non-vanishing contibutions

$$h^0_{\text{inv}}(X;\mathcal{O}(5,3)) = 25 \quad , \quad h^4_{\text{inv}}(X;\mathcal{O}(5,3)) = 155,$$
$$h^0_{\text{non-inv}}(X;\mathcal{O}(5,3)) = 11 \quad , \quad h^4_{\text{non-inv}}(X;\mathcal{O}(5,3)) = 250\,.$$

4.3.3 Tangent bundle of a subvariety

- [Bundle Type]: `"TangentBundle“`
- [Additional Input]: {*Complete intersection charges*}
- [Optional: Type of Subvariety]: `"Calabi-Yau“`/ `"Unknown"`

Computes the cohomology dimensions of the tangent bundle of S, i.e. $\dim H^i(S;T_S)$. If you choose the type of the subvariety to be "Calabi-Yau", this will be taken into account in the calculation. As always an empty list of intersecting surfaces returns the cohomology of the tangent bundle of the ambient toric variety.

Example: Consider the ambient space $\tilde{\mathbb{P}}^5_{111113}$ as specified above. If you evaluate the order

```
CohomologyOf["TangentBundle",P111113,{{1,4},{1,4}},“Calabi-Yau”];
```

you will get the result `{0,86,2,0}` which means that $\dim H^0(S;T_S) = \dim H^3(S;T_S) = 0$, $\dim H^1(S;T_S) = 86$ and $\dim H^2(S;T_S) = 2$.

4.3.4 Λ^k(Cotangent Bundle) of a subvariety for $k = 0, 1, 2$

- [Bundle Type]: `"Lambda`$\underline{k}$`CotangentBundle“`
- [Additional Input]: {*Complete intersection charges*}

- [Optional: Type of Subvariety]: `"Calabi-Yau"` / `"Unknown"`

Those three commands compute the exterior powers $k = 0, 1, 2$ of the cotangent bundle of the hypersurface or complete intersection $S = D_1 \cap \cdots \cap D_n$:

$$\dim H^i(S; \Lambda^k T_S^*),$$

The input data is exactly the same as for the tangent bundle cohomology.

Example: Consider the ambient space $\tilde{\mathbb{P}}^5_{111113}$ as specified above. If you type

```
CohomologyOf["Lambda0CotangentBundle",P111113,
                    {{1,4},{1,4}},"Calabi-Yau"];
```

you will get `{1,0,0,1}`.

4.3.5 Λ^k(Monad bundle) of a subvariety for $k = 1, 2$

- [Bundle Type]: `"Lambda`$\underline{k}$`MonadBundle"`
- [Additional Input]: `{`*Complete intersection charges, Bundle charges, Bundle Constraints, V_r*`}`

Those two commands compute the exterior powers $k = 1, 2$ of the specified monad bundle (1) of the hypersurface or complete intersection $S = S_1 \cap \cdots \cap S_l$. Here in addition to the charges of the complete intersection we also need the bundle charges N_k as well as the bundle constraints M_k and the power V_r, all defined in (1) .

Example: Consider the ambient space $\tilde{\mathbb{P}}^5_{111113}$ as specified above.

```
 CohomologyOf["Lambda1MonadBundle",P111113,{{{1,4},{1,4}},
(*BundleCharges*){{1,0},{0,1},{0,1},{0,1},{0,2}},
(*BundleConstraints*){{1,5}}},"Calabi-Yau"];
```

The result will be $\{0, 2, 102 + \mathtt{A}_{70}, \mathtt{A}_{70}\}$. Here the $\mathtt{A}_{70}$ is a constant which could not be determined using only the dimensions of the line bundle cohomologies in the long exact sequences. A method that actually evaluates the maps in order to evaluate this constant is currently not included in the program. But one can use the verbose function described below to have an explicit look at the long exact sequences. In certain cases, properties of the maps of the monad may be chosen in order to determine such a parameter in an easy way.

So far we have only considered monads that are non-exact sequences. If you want to calculate the cohomology of an exact monad, you can simply do that by putting the parameter V_r to zero. Then the non-exact sequence becomes exact and the vector bundle is given as the kernel of the corresponding map:

$$0 \longrightarrow V \longrightarrow \bigoplus_{k=1}^{V_n} \mathcal{O}_S(N_k) \longrightarrow \bigoplus_{i=1}^{V_l} \mathcal{O}_S(M_i)\,, \tag{2}$$

4.3.6 Hodge diamond of a subvariety

- [Bundle Type]: `"HodgeDiamond"`
- [Additional Input]: {*Complete intersection charges*}
- [Optional: Type of Subvariety]: `"Calabi-Yau"`/ `"Unknown"`

Computes the entire Hodge diamond of the hyper surface or complete intersection S. Up to 4 dimensions one usually gets a unique result which may not be the case for higher dimensions. The output contains the Hodge diamond, the Betti numbers as well as the Euler character of the requested subvariety.

Example: The following example for a Calabi-Yau 4-fold is taken from the paper arXiv:0912.3524 and describes a compact complete intersection Calabi-Yau 4-fold used in the construction of F-theory GUT vacua. The toric data of the ambient space (found in table B.1 of the aforementioned paper) is specified by the following variable:

```
Example4Fold = {
  (*Coordinates*){v1, v2, v3, v4, v5, v6, v1s, v7, v8, v9, v10},
  (*Stanley Reisner*) {{v3,v9},{v5,v9},{v7,v10},{v1,v2,v3},
   {v4,v1s,v8},{v4,v7,v8},{v4,v8,v9},
   {v5,v6,v1s},{v5,v6,v10},{v1,v2,v6,v1s}},
  (*Equivalence Relations*){{3, 3, 3, 3, 0}, {2, 2, 2, 2, 0},
   {1, 0, 0, 0, 0}, {0, 0, 1, 0, 0}, {0, 0, 0, 1, 0},
   {0, 1, 0, 0, 0}, {0, 1, 1, 0, 0}, {0, 0, 1, 0, 1},
   {0, 0, 1, 0, 0}, {0,-1,-1, 1,-1}, {0, 0, 0, 0, 1}}
  };
```

The 4-fold is given by the intersection of two divisors in the ambient space, which are specified in the variable

```
ComplInt = {{6, 6, 6, 6, 0}, {0, 0, 2, 1, 1}};
```

and the actual command for the computation of the Hodge diamond is then

```
CohomologyOf["HodgeDiamond",Example4Fold,ComplInt,"Calabi-Yau"];
```

which takes around 20 seconds on a current desktop computer and computes 264 intermediate ambient space line bundle cohomologies via **cohomCalg**. The full Hodge diamond as well as the Betti numbers and the Euler character are then printed in a very readable form:

```
                1                   1
            0       0               0
        0       5       0           5
    0       0       0       0       0
1     1115    4524    1115    1  6756   χ = 6768
    0       0       0       0       0
        0       5       0           5
            0       0               0
                1                   1
```

4.3.7 Endomorphism bundle of the tangent bundle of a subvariety

- [Bundle Type]: `"EndTangentBundle"`
- [Additional Input]: {*Complete intersection charges*}

Computes the cohomology dimensions $\dim H^i(S; \mathrm{End}(T_S))$ of the endomorphism bundle $\mathrm{End}(T_S)$, which represents the bundle deformations of the hypersurface / complete intersection S. For such calculations it is usually not necessary only to consider the dimensions of the line bundles, but more is needed. The routine is still useful since you can choose a verbose level in order to obtain the long exact sequences. So if you know about certain map properties you may be able to put parameters to zero manually and see how the result changes.

4.3.8 Endomorphism Bundle of the monad bundle of a Subvariety

- [Bundle Type]: `"EndMonadBundle"`
- [Additional Input]: {*Complete intersection charges, Bundle charge, Bundle Constraints,* V_r}

Computes the cohomology dimensions $\dim H^i(S; \mathrm{End}(V))$ of the endomorphism bundle $\mathrm{End}(V)$, which represents the bundle deformations of the monad bundle (1). As for the endomorphism bundle of the tangent bundle, here one usually needs additional information about the maps in the monad.

4.4 Verbose Level and Output Format

Since it may be useful to see how the long exact sequences involved in the internal computations actually look like—especially if one does not get a unique result in the end—there is an option in the **cohomCalg Koszul** extension to do so. In total there are 6 different verbose levels.

- "`Verbose-1`": Only the final result is returned.
- "`Verbose0`": The result is printed along with a small statistic about calculation time.
- "`Verbose1`": All long exact sequences coming from the Euler sequence will be shown.
- "`Verbose2`": Like "Verbose1" with more details.
- "`Verbose3`": All long exact sequences coming from the Koszul sequence will be shown.
- "`Verbose4`": All long exact sequences coming from the Euler and the Koszul sequence will be shown.
- "`Verbose5`": Like "Verbose4" with more details.
- "`Verbose6`": This Verbose function only works for the equivariant cohomology calculation and shows you the representatives of the cohomology of the corresponding line bundle. It also prints how the $\mathbb{Z}^n$-action acts on them and which of the Laurent monomials are invariant.

There is one more optional input parameter in which you can let the program know if you are using *Mathematica 7* in a terminal or with the graphical interface. Chose `"Terminal"` for [Optional: Input Type] if you are using the terminal interface.

Proper citation

If you find the **cohomCalg** package useful to your project please have a look into the file *Proper Citation.txt* located in the package archive's main directory. There you find a ready-to-use BibTeX entry for **cohomCalg** [6].

Acknowledgements

We would like to thank Jim Halverson for bug reporting and general feedback as well as Fabian Rühle for pointing out compatibility issues with Mathematica 8. Furthermore, we are grateful to Prof. Alois Kabelschacht for his help in the distribution of this project.

References

[1] R. Blumenhagen, B. Jurke, T. Rahn, and H. Roschy, "Cohomology of Line Bundles: A Computational Algorithm," *J. Math. Phys.* **51** no. 10, (2010) 103525, arXiv:1003.5217 [hep-th].

[2] T. Rahn and H. Roschy, "Cohomology of Line Bundles: Proof of the Algorithm," *J. Math. Phys.* **51** no. 10, (2010) 103520, arXiv:1006.2392 [hep-th].

[3] S.-Y. Jow, "Cohomology of toric line bundles via simplicial Alexander duality," arXiv:1006.0780 [math.AG].

[4] R. Blumenhagen, B. Jurke, T. Rahn, and H. Roschy, "Cohomology of Line Bundles: Applications," arXiv:1010.3717 [hep-th].

[5] R. Blumenhagen, B. Jurke, and T. Rahn, "Computational Tools for Cohomology of Toric Varieties," arXiv:1104.1187 [hep-th].

[6] "**cohomCalg** package." Download link, 2010. https://github.com/BenjaminJurke/cohomCalg. High-performance line bundle cohomology computation based on [1].

Mathematica is a registered trademark of Wolfram Research, Inc. All rights reserved. *Windows* is a registered trademark of Microsoft Corporation in the United States and other countries. The lightbulb image used in the application logo was made by DragonArt and is used under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license.