# polymaking

## Interfacing the geometry software polymake

Version 0.8.9

9 April 2026

**Marc Roeder**

**Marc Roeder**
Email: roeder.marc@gmail.com

## Abstract

This package provides a very basic interface to the polymake program by Ewgenij Gawrilow, Michael Joswig et al. [GJ]. The polymake program itself is not included.

## Copyright



## Acknowledgements

This work has been supported by Marie Curie Grant No. MTKD-CT-2006-042685

# Contents

# Chapter 1

# Installation and Preface

To install the package, just unpack it in your packages directory (usually `~/gap/pkg` for local installation). To use polymaking, you need a working installation of the program polymake https://polymake.org. The package has been tested on linux and Mac OS X (10.4, 10.5 and 10.6). But it should be as platform independent as GAP and polymake.

The interaction with polymake is restricted to writing files and carrying out simple operations. These looked like

`polymake file KEYWORD1 KEYWORD2 KEYWORD3`

on the command line for polymake versions before 4. The keywords are polymake methods without arguments. Since polymake no longer supports this interface the polymaking package provides the script `lib/pm_script_arg.pl` to emulate this.

`polymake --script lib/pm_script_arg.pl KEYWORD1 KEYWORD2 KEYWORD3`

Using custom scripts is not supported.

Every call to polymake will re-start the program anew. This causes considerable overhead. The number of calls to polymake is reduced by caching the results in the so-called `PolymakeObject` in GAP. As of polymaking version 0.8.0, old versions of polymake (i.e. versions before 2.7.9) are not supported anymore.

## 1.1 A few words about the installation of polymake

polymaking will try to guess the location of polymake. If this fails, a warning is issued at load time (`InfoWarning` level 1). Note that the guessing procedure is suppressed when `POLYMAKE_COMMAND` (3.2.1) is set manually (see 1.2.3).

`setenv PATH ${PATH}:<your polymakepath>`

The general rule is: If polymaking finds polymake by itself, there is nothing to worry about.

## 1.2 Setting variables for external programs

As polymaking uses the program polymake, it needs to know where this program lives. The communication with polymake is done by writing files for polymake and reading its output (as returned to standard output "the prompt"). Note that the interface does not read any polymake file.

### 1.2.1 SetPolymakeDataDirectory

▷ `SetPolymakeDataDirectory(`*`dir`*`)` (method)

Sets the directory in which all polymake files are created to *`dir`*. The standard place for these files is a temporary directory generated when the package is loaded. This manipulates `POLYMAKE_DATA_DIR` (3.2.2).

### 1.2.2 SetPolymakeCommand

▷ `SetPolymakeCommand(`*`command`*`)` (method)

Sets the name for the polymake program to *`command`*. This manipulates `POLYMAKE_COMMAND` (3.2.1).

### 1.2.3 Setting variables permanently

To permanently set the values of `POLYMAKE_COMMAND` (3.2.1), and `POLYMAKE_DATA_DIR` (3.2.2), add the lines

```
POLYMAKE_DATA_DIR:=Directory("/home/mypolymakedatadir");
POLYMAKE_COMMAND:=Filename(Directory("/home/mypolymakebindir/"),"polymake");
```

to your `.gaprc` file (see **Reference: The gap.ini and gaprc files**). Note that these have to be *before* the `LoadPackage("polymaking");` line. Or you can change the values of the above variables by editing `lib/environment.gi`

# Chapter 2

# Polymake interaction

## 2.1 Creating Polymake Objects

The interaction with the polymake program is done via files. A `PolymakeObject` is mainly a pointer to a file and a list of known properties of the object. These properties need not be stored in the file. Whenever polymake is called, the returned value is read from standard output and stored in the `PolymakeObject` corresponding to the file for which polymake is called. The files for polymake are written in the old (non-xml) format. The first run of polymake converts them into the new (xml) format. This means that changes to the file by means of the methods outlined below after the first run of polymake will probably lead to corrupted files.

### 2.1.1 CreateEmptyFile

▷ `CreateEmptyFile(`*`filename`*`)` (method)

**Returns:** nothing

Creates an empty file with name *`filename`*. Note that *`filename`* has to include the full path and the directory for the file must exist.

### 2.1.2 CreatePolymakeObject

▷ `CreatePolymakeObject(`*`[prefix][,] [dir][,] [appvertyp]`*`)` (method)

**Returns:** `PolymakeObject`

If called without arguments, this method generates an empty file in the directory defined by `POLYMAKE_DATA_DIR` (3.2.2). If a directory *`dir`* is given (this directory must exist), an empty file is generated in this directory. If *`prefix`* is not given, the file is called `polyN` where `N` is the current runtime. If a file of this name already exists, a number is appended separated by a dot (example: "poly1340" and "poly1340.1"). If *`prefix`* is given, the filename starts with this prefix. Optionally, the file can be generated with a header specifying application, version and type of the object. This is done by passing the triple of strings *`appvertyp`* to `CreatePolymakeObject`. A valid triple is `["polytope","2.3","RationalPolytope"]`. Validity is checked by `CheckAppVerTypList` (2.1.3).

### 2.1.3 CheckAppVerTypList

▷ `CheckAppVerTypList(`*`appvertyp`*`)` (method)

**Returns:** `bool`

Checks if the triple *`arppvertyp`* of strings specifies an application and type of polymake version 2.3. More specifically, the first entry has to be an application from `["polytope","surface","topaz"]` and the third entry has to be a type corresponding to the application given in the first entry. The second entry is not checked.

### 2.1.4 CreatePolymakeObjectFromFile

▷ `CreatePolymakeObjectFromFile(`*`[dir, ]filename`*`)` (method)

**Returns:** `PolymakeObject`

This method generates a `PolymakeObject` corresponding to the file *`filename`* in the directory *`dir`*. If *`dir`* is not given, the `POLYMAKE_DATA_DIR` is used.If no file with name *`filename`* exists in *`dir`* (or `POLYMAKE_DATA_DIR`, respectively), an empty file is created. Note that the contents of the file do not matter for the generation of the object. In particular, the object does not know any of the properties that might be encoded in the file. The only way to transfer information from files to `PolymakeObjects` is via `Polymake` (2.5.1).

## 2.2 Accessing Properties of Polymake Objects

A `PolymakeObject` contains information about the directory of its file, the name of its file and about properties calculated by calling `Polymake` (2.5.1). The properties returned by the `polymake` program are stored under the name `polymake` assigns to them (that is, the name of the data block in the corresponding file). The following methods can be used to access the information stored in a `PolymakeObject`. But be careful! All functions return the actual object. No copies are made. So if you change one of the returned objects, you change the `PolymakeObject` itself.

### 2.2.1 DirectoryOfPolymakeObject

▷ `DirectoryOfPolymakeObject(`*`poly`*`)` (method)

**Returns:** Directory

Returns the directory of the file associated with *`poly`*.

### 2.2.2 FilenameOfPolymakeObject

▷ `FilenameOfPolymakeObject(`*`poly`*`)` (method)

**Returns:** String

Returns the name of the file associated with *`poly`*. This does only mean the name of the *file*, not the full path. For the full path and file name see `FullFilenameOfPolymakeObject` (2.2.3)

### 2.2.3 FullFilenameOfPolymakeObject

▷ `FullFilenameOfPolymakeObject(`*`poly`*`)` (method)

**Returns:** String

Returns the file associated with the `PolymakeObject` *`poly`* with its complete path.

### 2.2.4 NamesKnownPropertiesOfPolymakeObject

▷ `NamesKnownPropertiesOfPolymakeObject(`*poly*`)` (method)

**Returns:** List of Strings

Returns a list of the names of all known properties. This does only include the properties returned by `Polymake` (2.5.1), `"dir"` and `"filename"` are not included. If no properties are known, `fail` is returned.

### 2.2.5 KnownPropertiesOfPolymakeObject

▷ `KnownPropertiesOfPolymakeObject(`*poly*`)` (method)

**Returns:** Record

Returns the record of all known properties. If no properties are known, `fail` is returned.

### 2.2.6 PropertyOfPolymakeObject

▷ `PropertyOfPolymakeObject(`*poly*`, `*name*`)` (method)

Returns the value of the property *name* if it is known. If the value is not known, `fail` is returned. *name* must be a `String`.

## 2.3 Example: Creating and Accessing Polymake Objects

Suppose the file `/tmp/threecube.poly` contains the three dimensional cube in polymake form. Now generate a `PolymakeObject` from this file and call `Polymake` (2.5.1) to make the vertices of the cube known to the object.

Example

```
### suppose we have a polymake file /tmp/threecube.poly
### containing a cube in three dimensions
gap> cube:=CreatePolymakeObjectFromFile(Directory("/tmp"),"threecube.poly");
<polymake object. No properties known>
gap> FilenameOfPolymakeObject(cube);
"threecube.poly"
gap> FullFilenameOfPolymakeObject(cube);
"/tmp/threecube.poly"
   #nothing is known about the cube:
gap> NamesKnownPropertiesOfPolymakeObject(cube);
fail
gap> Polymake(cube,"VERTICES");
[ [ -1, -1, -1 ], [ 1, -1, -1 ], [ -1, 1, -1 ], [ 1, 1, -1 ], [ -1, -1, 1 ],
  [ 1, -1, 1 ], [ -1, 1, 1 ], [ 1, 1, 1 ] ]
   # Now <cube> knows its vertices:
gap> Print(cube);
<polymake object threecube.poly. Properties known: [ "VERTICES" ]>
gap> PropertyOfPolymakeObject(cube,"VERTICES");
[ [ -1, -1, -1 ], [ 1, -1, -1 ], [ -1, 1, -1 ], [ 1, 1, -1 ], [ -1, -1, 1 ],
  [ 1, -1, 1 ], [ -1, 1, 1 ], [ 1, 1, 1 ] ]
gap> KnownPropertiesOfPolymakeObject(cube);
rec(
```

```
    VERTICES := [ [ -1, -1, -1 ], [ 1, -1, -1 ], [ -1, 1, -1 ], [ 1, 1, -1 ],
        [ -1, -1, 1 ], [ 1, -1, 1 ], [ -1, 1, 1 ], [ 1, 1, 1 ] ] )
```

## 2.4 Writing to Polymake Objects

To transfer data from GAP to polymake, the following methods can be used. But bear in mind that none of these functions test if the resulting polymake file is still consistent.

### 2.4.1 AppendToPolymakeObject

▷ `AppendToPolymakeObject(`*`poly, string`*`)` (method)

**Returns:** nothing

This appends the string *`string`* to the file associated to the `PolymakeObject` *`poly`*. It is not tested if the string is syntactically correct as a part of a polymake file. It is also not tested if the string is compatible with the data already contained in the file.

INEQUALITIES, POINTS and VERTICES can be appended to a polymake object using the following functions:

### 2.4.2 AppendPointlistToPolymakeObject

▷ `AppendPointlistToPolymakeObject(`*`poly, pointlist`*`)` (method)

**Returns:** nothing

Takes a list *`pointlist`* of vectors and converts it into a string which represents a polymake block labeled "POINTS". This string is then added to the file associated with *`poly`*. The "POINTS" block of the file associated with *`poly`* then contains points with leading ones, as polymake uses affine notation.

### 2.4.3 AppendVertexlistToPolymakeObject

▷ `AppendVertexlistToPolymakeObject(`*`poly, pointlist`*`)` (method)

**Returns:** nothing

Does the same as `AppendPointlistToPolymakeObject`, but with "VERTICES" instead of "POINTS".

### 2.4.4 AppendInequalitiesToPolymakeObject

▷ `AppendInequalitiesToPolymakeObject(`*`poly, ineqlist`*`)` (method)

**Returns:** nothing

Just appends the inequalities given in *`ineqlist`* to the polymake object *`poly`* (with caption "INEQUALITIES"). Note that this does not check if an "INEQUALITIES" section does already exist in the file associated with *`poly`*.

### 2.4.5 ConvertMatrixToPolymakeString

▷ `ConvertMatrixToPolymakeString(`*`name, matrix`*`)` (method)

**Returns:** String

This function takes a matrix *matrix* and converts it to a string. This string can then be appended to a polymake file via AppendToPolymakeObject (2.4.1) to form a block of data labeled *name*. This may be used to write blocks like INEQUALITIES or FACETS.

### 2.4.6 ClearPolymakeObject

▷ ClearPolymakeObject(*poly[, appvertyp]*) (method)

**Returns:** nothing

Deletes all known properties of the PolymakeObject *poly* and replaces its file with an empty one.
If the triple of strings *appvertyp* specifying application, version and type (see CheckAppVerTypList (2.1.3)) is given, the file is replaced with a file that contains only a header specifying application, version and type of the polymake object.

There are also methods to manipulate the known values without touching the file of the PolymakeObject:

### 2.4.7 WriteKnownPropertyToPolymakeObject

▷ WriteKnownPropertyToPolymakeObject(*poly, name, data*) (method)

Takes the object *data* and writes it to the known properties section of the PolymakeObject *poly*. The string *name* is used as the name of the property. If a property with that name already exists, it is overwritten. Again, there is no check if *data* is consistent, correct or meaningful.

### 2.4.8 UnbindKnownPropertyOfPolymakeObject

▷ UnbindKnownPropertyOfPolymakeObject(*poly, name*) (method)

If the PolymakeObject *poly* has a property with name *name*, that property is unbound. If there is no such property, fail is returned.

## 2.5 Calling Polymake and converting its output

### 2.5.1 Polymake

▷ Polymake(*poly, option:  PolymakeNolookup*) (method)

This method calls the polymake program (see POLYMAKE_COMMAND (3.2.1)) with the option *option*. You may use several keywords such as "FACETS VERTICES" as an option. The returned value is cut into blocks starting with keywords (which are taken from output and not looked up in *option*). Each block is then interpreted and translated into GAP readable form. This translation is done using the functions given in ObjectConverters (4.1.4). The first line of each block of polymake output is taken as a keyword and the according entry in ObjectConverters (4.1.4) is called to convert the block into GAP readable form. If no conversion function is known, an info string is printed and fail is returned. If only one keyword has been given as *option*, Polymake returns the result of the conversion operation. If more than one keyword has been given or the output consists of more than one block, Polymake returns fail. In any case, the calculated values for each block are

stored as known properties of the PolymakeObject *poly* as long as they are not fail. If Polymake is called with an option that corresponds to a name of a known property of *poly*, the known property is returned. In this case, there is no call of the external program. (see below for suppression of this feature).

Note that the command Polymake returns fail if nothing is returned by the program polymake or more than one block of data is returned. For example, the returned value of Polymake(poly,"VISUAL") is always fail. Likewise, Polymake(poly,"POINTS VERTICES") will return fail (but may add new known properties to *poly*). For a description of the conversion functions, see chapter 4.

If the option *PolymakeNolookup* is set to anything else than false, the polymake program is called even if *poly* already has a known property with name *option*.

Note that whenever Polymake (2.5.1) returns fail, a description of the problem is stored in POLYMAKE_LAST_FAIL_REASON (3.1.2). If you call Polymake (2.5.1) with more than one keyword, POLYMAKE_LAST_FAIL_REASON (3.1.2) is changed before polymake is called. So any further reason to return fail will overwrite it.

## 2.6 An Example

Let's generate a three dimensional permutahedron.

Example

```
gap> S:=SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
gap> v:=[1,2,3];
[ 1, 2, 3 ]
gap> points3:=Orbit(S,v,Permuted);;
# project to reduce ambient dimension
gap> points:=points3{[1..6]}{[1,2]};;
gap> permutahedron:=CreatePolymakeObject();
<polymake object. No properties known>
gap> AppendPointlistToPolymakeObject(permutahedron,points);
gap> Polymake(permutahedron,"VOLUME");
3
gap> Polymake(permutahedron,"N_VERTICES");
6
#Now <permutahedron> knows its number of vertices, but not the vertices:
gap> PropertyOfPolymakeObject(permutahedron,"VERTICES");
fail
gap> NamesKnownPropertiesOfPolymakeObject(permutahedron);
[ "VOLUME", "N_VERTICES" ]
#Let's look at the object!
gap> Polymake(permutahedron,"VISUAL");
#I  There was no or wrong polymake output
fail
gap> Polymake(permutahedron,"DIM");
2
```

# Chapter 3

# Global Variables

## 3.1 Getting information about polymake output

### 3.1.1 InfoPolymaking

▷ `InfoPolymaking` (info class)

If set to at least 2, the output of polymake is shown. At level 1, warnings are shown. This is the default. And at level 0, the polymake package remains silent.

### 3.1.2 POLYMAKE_LAST_FAIL_REASON

▷ `POLYMAKE_LAST_FAIL_REASON` (global variable)

Contains a string that explains the last occurrence of `fail` as a return value of `Polymake` (2.5.1).

## 3.2 Variables for system interaction

The variables for interaction with the system are contained in the file `environment.gi`. Each of these variables has a function to set it, see 1.2. If `POLYMAKE_COMMAND` or `POLYMAKE_DATA_DIR` are set at startup, they are not overwritten. So if you don't want (or don't have the rights) to modify `environment.gi`, you can set the variables in your `.gaprc` file.

### 3.2.1 POLYMAKE_COMMAND

▷ `POLYMAKE_COMMAND` (global variable)

This variable should contain the name of the polymake program in the form as returned by `Filename` So a probable value is `Filename(Directory("/usr/local/bin"),"polymake")`.

### 3.2.2 POLYMAKE_DATA_DIR

▷ `POLYMAKE_DATA_DIR` (global variable)

In this directory the files for polymake will be created. By default, this generates a temporary directory using `DirectoryTemporary`

# Chapter 4

# Converting Polymake Output

## 4.1 The General Method

When polymake is called, its output is read as a string and then processed as follows:

1. the lines containing upper case letters are found. These are treated as lines containing the keywords. Each of those lines marks the beginning of a block of data.
2. The string is then cut into a list of blocks (also strings). Each block starts with a line containing the keyword and continues with some lines of data.
3. for each of the blocks, the appropriate function of `ObjectConverters` is called. Here "appropriate" just means, that the keyword of the block coincides with the name of the function.
4. The output of the conversion function is then added to the known properties of the `PolymakeObject` for which `Polymake` was called.

### 4.1.1 Converter- Philosopy

The converter functions should take meaningful polymake data into meaningful GAP data. This sometimes means that the (mathematical) representation is changed. Here is an example: polymake writes vectors as augmented affine vectors of the form `1 a1 a2 a3...` which does not go very well with the usual GAP conventions of column vectors and multiplying matrices from the right. So polymaking converts such a vector to `[a1,a2,a3,...]` and the user is left with the problem of augmentation and left or right multiplication.

Another area where the GAP object isn't a literal translation from the polymake world is combinatorics. In Polymake, list elements are enumerated starting from 0. GAP enumerates lists starting at 1. So the conversion process adds 1 to the numbers corresponding to vertices in facet lists, for example.

The conversion process is done by the following methods:

### 4.1.2 ConvertPolymakeOutputToGapNotation

▷ `ConvertPolymakeOutputToGapNotation(`*string*`)` (method)

**Returns:** Record having polymake keywords as entry names and the respective converted polymake output as entries.

Given a the output of the polymake program as a string *string*, this method first calls SplitPolymakeOutputStringIntoBlocks (4.1.3). For each of the returned blocks, the name (=first line) of the block is read and the record ObjectConverters (4.1.4) is looked up for an entry with that name. If such an entry exists, it (being a function!) is called and passed the block. The returned value is then given the name of the block and added to the record returned by ConvertPolymakeOutputToGapNotation.

### 4.1.3 SplitPolymakeOutputStringIntoBlocks

▷ SplitPolymakeOutputStringIntoBlocks(*string*) (method)

**Returns:** List of strings -- "blocks"--

The string *string* is cut at the lines starting with an upper case character and consisting only of upper case letters, numbers and underscore (_) characters. The parts are returned as a list of strings. The initial string *string* remains unchanged.

### 4.1.4 ObjectConverters

▷ ObjectConverters (global variable)

The entries of this record are labeled by polymake keywords. Each of the entries is a function which converts a string returned by polymake to GAP format. So far, only a few converters are implemented. To see which, try RecNames(ObjectConverters);

You can define new converters using the basic functions described in section 4.2.

## 4.2 Conversion Functions

The following functions are used for the functions in ObjectConverters (4.1.4).

### 4.2.1 ConvertPolymakeNumber

▷ ConvertPolymakeNumber(*string*) (method)

The string *string* is converted to a rational number. Unlike Rat, it tests, if the number represented by *string* is a floating point number an converts it correctly. If this is the case, a warning is issued.

### 4.2.2 ConvertPolymakeScalarToGAP

▷ ConvertPolymakeScalarToGAP(*list*) (method)

If *list* contains a single string, this string is converted into a number using ConvertPolymakeNumber (4.2.1).

### 4.2.3 ConvertPolymakeMatrixOrListOfSetsToGAP

▷ ConvertPolymakeMatrixOrListOfSetsToGAP(*list*) (method)

▷ ConvertPolymakeMatrixOrListOfSetsToGAPPlusOne(*list*) (method)

Tries to decide if the list *list* of strings represents a matrix or a list of sets by testing if they start with "{". It then calls either `ConvertPolymakeMatrixToGAP` (4.2.4) or `ConvertPolymakeListOfSetsToGAP` (4.2.8). The "PlusOne" version calls `ConvertPolymakeListOfSetsToGAPPlusOne` (4.2.8) if *list* represents a list of sets.

### 4.2.4 ConvertPolymakeMatrixToGAP

▷ `ConvertPolymakeMatrixToGAP(`*list*`)` (method)
▷ `ConvertPolymakeMatrixToGAPKillOnes(`*list*`)` (method)

The list *list* of strings is interpreted as a list of row vectors and converted into a matrix. The "KillOnes" version removes the leading ones.

### 4.2.5 ConvertPolymakeVectorToGAP

▷ `ConvertPolymakeVectorToGAP(`*list*`)` (method)
▷ `ConvertPolymakeVectorToGAPKillOne(`*list*`)` (method)
▷ `ConvertPolymakeIntVectorToGAPPlusOne(`*list*`)` (method)

As the corresponding "Matrix" version. Just for vectors. `ConvertPolymakeIntVectorToGAPPlusOne` requires the vector to contain integers. It also adds 1 to every entry.

### 4.2.6 ConvertPolymakeBoolToGAP

▷ `ConvertPolymakeBoolToGAP(`*list*`)` (method)

If *list* contains a single string, which is either 0,false,1, or true this function returns `false` or `true`, respectively.

### 4.2.7 ConvertPolymakeSetToGAP

▷ `ConvertPolymakeSetToGAP(`*list*`)` (method)

Let *list* be a list containing a single string, which is a list of numbers separated by whitespaces and enclosed by `{` and `}` . The returned value is then a set of rational numbers (in the GAP sense).

### 4.2.8 ConvertPolymakeListOfSetsToGAP

▷ `ConvertPolymakeListOfSetsToGAP(`*list*`)` (method)
▷ `ConvertPolymakeListOfSetsToGAPPlusOne(`*list*`)` (method)

Let *list* be a list containing several strings representing sets. Then each of these strings is converted to a set of rational numbers and the returned value is the list of all those sets. The "PlusOne" version adds 1 to every entry.

### 4.2.9 ConvertPolymakeGraphToGAP

▷ `ConvertPolymakeGraphToGAP(`*`list`*`)` (method)

Let *`list`* be a list of strings representing sets (that is, a list of integers enclosed by `{` and `}`). Then a record is returned containing two sets named `.vertices` and `.edges`.

# References

[GJ] Ewgenij Gawrilow and Michael Joswig. polymake. https://polymake.org/. 2

# Index