

# arrangement

**A GAP extension for Arrangements**

Package Version 0.69

Dean Serenevy

# Contents

<b>1</b>	<b>The Arrangement Package</b>	<b>3</b>	4.7	Fundamental Group of a Hyperplane Arrangement . . . . .	12
1.1	Description . . . . .	3	4.8	Homology and Cohomology Invariants	12
1.2	Synopsis . . . . .	3	<b>5</b>	<b>The Fox Calculus</b>	<b>13</b>
1.3	Objects and Types . . . . .	4	5.1	Fox Calculus Helpful Conversions . . . . .	14
1.4	Defined Categories . . . . .	5	5.2	Fox Calculus Utility Functions . . . . .	14
<b>2</b>	<b>Specifying Options in GAP</b>	<b>6</b>	<b>6</b>	<b>Utility Commands</b>	<b>17</b>
<b>3</b>	<b>Arrangements</b>	<b>7</b>	6.1	General Utility . . . . .	17
3.1	Constructing Arrangements from Scratch . . . . .	7	6.2	Multivariable Polynomials . . . . .	20
3.2	Arrangements from Other Arrangements . . . . .	7	<b>7</b>	<b>Internally Used Commands</b>	<b>21</b>
3.3	Computing Arrangement Invariants	8	7.1	File related commands . . . . .	21
3.4	Arrangement Properties . . . . .	8	7.2	User interaction . . . . .	21
3.5	Arrangement Invariants . . . . .	8	7.3	Arrangement.g . . . . .	21
3.6	Defining Polynomials of an Arrangement . . . . .	8	7.4	braid.g . . . . .	22
3.7	Display and Output . . . . .	9	7.5	graphs.g . . . . .	22
<b>4</b>	<b>Hyperplane Arrangements</b>	<b>10</b>	7.6	cohomology.g . . . . .	24
4.1	Constructing Hyperplane Arrangements from Scratch . . . . .	10	7.7	face lattice.g . . . . .	24
4.2	Hyperplane Arrangements from Other Arrangements . . . . .	12	7.8	matroid.g . . . . .	25
4.3	Hyperplane Arrangement Filters . . . . .	12	7.9	polynomials.g . . . . .	27
4.4	Hyperplane Arrangement Invariants . . . . .	12	7.10	Utilities.g . . . . .	28
4.5	Defining Polynomials of a Hyperplane Arrangement . . . . .	12	<b>Bibliography</b>		<b>30</b>
4.6	Intersection Lattice of a Hyperplane Arrangement . . . . .	12	<b>Index</b>		<b>31</b>

# 1

# The Arrangement Package

## 1.1 Description

This package provides a collection of methods for working with arrangements. These methods are capable of computing many topological and combinatorial objects associated to an arrangement. Computed data is cached within the arrangement object.

When there is more than one method of computing a particular invariant, then the package will try to find the most efficient means. It will take into consideration both previously computed information as well as computation time for new data.

## 1.2 Synopsis

```
# The arrangement:  $x(x+y)(x+y+z)$ 
#-----
gap> A := CentralHyperplaneArrangement( [ [1,0,0], [1,1,0], [1,1,1] ] );
< 3-hyperplane arrangement on 3 lines >

# (Signed) Graphic Arrangements
#-----
# order within pairs doesn't matter ([i,i] is not allowed)
gap> A := HyperplaneArrangement( : Graph      := [[1,2],[3,1],[1,4]] );
< signed graphic 4-arrangement on 3 hyperplanes >

# order within pairs matters ([i,i] is allowed)
gap> A := HyperplaneArrangement( : SignedGraph := [[1,2],[3,1],[1,1]] );
< signed graphic 3-arrangement on 3 hyperplanes >

# Invariants
#-----
gap> Dimension(A);
3
gap> NrCurves(A);
3
gap> Rank(A);
3
gap> FundamentalGroup(A);
<fp group on the generators [ f1, f2, f3 ]>

gap> PolynomialMatrix(A);
[ [ -1, 1, 0 ], [ 1, 0, 1 ], [ 1, 0, 0 ] ]
gap> A.1;
-x+y
```

```

gap> IntersectionLattice(A);
[ [ [1], [2], [3] ],      # a list of sets of lattice points
  [ [1,2], [1,3], [2,3] ], # indexed by codimension
  [ [1,2,3] ]
]
gap> DisplayIntersectionLattice(A); # A nicer format
1   1, 2, 3
2   12, 13, 23
3   123

# More invariants
#-----
gap> B := BraidArrangement(3);
< Braid arrangement of rank 3 >
gap> IsDecomposable(B);
false
gap> IsSupersolvable(B);
true
gap> MaximalChainOfModularElements(B);
[ [ 1 ], [ 1, 2, 4 ], [ 1, 2, 3, 4, 5, 6 ] ]
gap> LCSQuotientRanks(B,5);
[ 6, 4, 10, 21, 54 ]

# Display and output
#-----
gap> Display(A); # Pretty-Print basic information
gap> Display(A, ["IntersectionLattice"]); # Just some information
gap> Display(A : Properties); # Known properties of A

gap> Dump(A); # Pretty-Print all known information
gap> ExtRepOfObj(A); # A record of all information about A

```

### 1.3 Objects and Types

Throughout this manual certain objects and types will be used as arguments to the various methods in this package. Most of these are not really objects, but are standard GAP data types that must have a particular form. Here we describe the types that we will use throughout the rest of the manual.

#### *Arrangement*

A GAP object which satisfies the filter `IsArrangement`.

#### *element/coefficient list*

Sometimes called an `ExtRep/coefficient list`. A list whose odd components contain something like the letter representation of a word in a free group (See `LetterRepAssocWord`, 35.6.8 in the GAP reference manual) and whose even components contain coefficients for the word immediately preceding them. For example:

```
[ [word list], coeff, [word list], coeff, ... ]
```

Where a “word list” is a list of signed integers  $x_1^2 \cdot x_2^{-1} \rightarrow [1, 1, -2]$ .

#### *Graph*

A standard  $n \times 2$  GAP integral matrix more conveniently thought of as a list of pairs  $[i, j]$  representing an edge connecting the nodes  $i$  and  $j$ .

In the context of a signed graph, the order of  $i$  and  $j$  is significant. If  $i < j$  then the pair represents a positive edge. If  $i > j$  then the pair represents a negative edge. If  $i = j$  then the pair represents a loop at the node  $i$ .

Each edge of a graph contributes one linear factor to the polynomial of an arrangement. Positive edges contribute a factor of  $(x_j - x_i)$ , negative edges contribute a factor of  $(x_j + x_i)$ , and loops contribute a factor of  $x_i$ .

#### *HyperplaneArrangement*

A GAP object which satisfies the filter `IsHyperplaneArrangement`.

#### *IntersectionLattice*

For hyperplane arrangements an intersection lattice is a GAP list of lists of lists. If  $L$  is an intersection lattice, then  $L[i]$  is a list of rank  $i$  lattice points. Each lattice point is a list of indices representing the hyperplanes intersecting in a codimension  $i$  component of the arrangement. The indices in each lattice point correspond to an index in the list of polynomial factors.

#### *PolynomialList*

A List of standard GAP multivariable polynomials. Each polynomial is expected to be irreducible and each polynomial is expected to have a unique set of zeros (that is, the polynomial of the arrangement must be square-free).

#### *PolynomialMatrix*

For hyperplane arrangements a polynomial matrix is a standard GAP matrix of rational numbers. Each row of the matrix is the coefficients of a single linear factor of the polynomial of the arrangement. If the arrangement is central then the dimension of the arrangement will correspond to the number of columns in the polynomial matrix. If the arrangement is affine then there will be an additional column for the constant term.

The polynomial of the arrangement must be square-free, so no two rows of the polynomial matrix may be linearly dependant. Also, there may be no constant factors, so in the affine case there can be no rows which are all zero except for the last column.

## 1.4 Defined Categories

- |     |  |   |
|-----|--|---|
| 1 ► | <code>IsArrangement( Obj )</code>  | C |
|     | Returns <code>true</code> if <code>Obj</code> is an arrangement.           |   |
| 2 ► | <code>IsHyperplaneArrangement( Obj )</code>                                | C |
|     | Returns <code>true</code> if <code>Obj</code> is a hyperplane arrangement. |   |

# 2

# Specifying Options in GAP

GAP has a feature that allows the passing of named parameters called `Options`. These options act recursively in that options given to a function will be available to all functions called by that function. The GAP documentation on `Options` is a bit unsatisfying so I am providing a bit about it here.

Using named parameters is useful in this context since we wish to allow the creation of arrangements from various different sources. Named parameters are more convenient way of doing this than using records since less typing is required and the function calls look more natural.

In GAP, named parameters are passed after a colon that comes after all positional parameters. These named parameters may either have a value, or simply set a flag. Here are some examples of valid option passing.

```
gap> MyPoly := [[1,0,0],[0,1,0],[0,0,1],[0,-1,1],[0,2,3]];
gap> MyGraph := [[1,2],[3,4],[1,4],[3,1],[1,1]];

# A standard function call
gap> HyperplaneArrangement( MyPoly );
< 2-hyperplane arrangement on 5 lines >

# A function call with an option
gap> HyperplaneArrangement( MyPoly : Dimension := 3 );
< 3-hyperplane arrangement on 5 lines >

# This is equivalent to the option : Central := true
gap> HyperplaneArrangement( MyPoly : Central );
< 3-hyperplane arrangement on 5 lines >

# If no positional parameters are passed, the colon is still required
gap> HyperplaneArrangement( : SignedGraph := MyGraph, Dimension := 5 );
< signed graphic 5-arrangement on 5 hyperplanes >
```

Normally something like `Dimension` would be computed automatically, however, in the last example we used the `Dimension` option to artificially inflate the dimension of the constructed arrangement. Without using named parameters, the positional arguments would need to be processed by the function and issues could arise in ambiguous cases.

# 3

# Arrangements

These are the user-level functions. See section 1.3 (Objects and Types) for a description of the type names used in this section.

## 3.1 Constructing Arrangements from Scratch

- 1 ▶ `Arrangement( PolynomialList )` F
- ▶ `Arrangement( PolynomialList : Options )` F
- ▶ `Arrangement( : Options )` F

Construct a new arrangement object. The default way to construct a arrangement is from a polynomial list. However, additional information may be provided though value options. If the provided options are sufficient to describe the arrangement, then you may omit the polynomial list.

Any options which are capable of serving as arrangement constructors also have a corresponding `ArrangementFrom*` function (for instance, `ArrangementFromIntersectionLattice`). All of the “From” functions may also take any of the options that the `Arrangement` function accepts.

### Constructive Options

This is the list of options which provide sufficient information to define (for the purposes of this package) an arrangement.

`PolynomialList := PolynomialList`

The List of (irreducible) polynomials which define the components of the arrangement. The arrangement is assumed to be square-free so each entry of the `PolynomialList` should be distinct. Also, no entry should be a constant polynomial.

### Non Constructive Options

This is the list of options which serve as additional information about an arrangement, but do not themselves provide sufficient information to define an arrangement object.

`BaseField := Field`

Currently, only the Cyclotomics is usable here

`CoefficientsRing := Ring`

Ring may be one of: Integers, Rationals, `_arr_Reals`, Cyclotomics

`Dimension := Integer`

`NrCurves := Integer`

## 3.2 Arrangements from Other Arrangements

- 1 ▶ `Deletion( Arrangement, Integer )` O
- ▶ `Deletion( Arrangement, ListOfIntegers )` O

Constructs a new arrangement from the original arrangement by removing the indicated list of curves. As much information as is possible is maintained in the newly constructed arrangement.

### 3.3 Computing Arrangement Invariants

All invariant functions follow the same rules for return values. If the computation succeeds, then the requested invariant will be returned. If the computation fails for some technical reason (For instance, the package does not know how to compute the invariant from the current list of known invariants) then `fail` will be returned. If the invariant is requested again, then the construction paths will be re-evaluated and the computation retried. If, however, there is reason that the requested invariant is mathematically invalid or impossible, then `false` will be returned and any future requests for the invariant will simply return `false` without retrying the computation.

There is one option which is universally recognized by the invariant computation routines. See section 2 (Specifying Options in GAP) for details on how to specify options in a function call, and 8 for information on permanently setting options.

#### Explain

Explain the algorithm being used to do perform the requested computations. Works best if the `ArrangementInfo` info level is set to at least 7, `SetInfoLevel( ArrangementInfo, 7 )`; . Not all functions make use of the `Explain` option.

### 3.4 Arrangement Properties

- 1 ▶ `IsEssentialArrangement( Arrangement )` P

True if the dimension of the arrangement is equal to the rank of the arrangement.

### 3.5 Arrangement Invariants

- 1 ▶ `BaseField( Arrangement )` A

Returns the ring over which the polynomials defining the arrangement are defined. Currently, the `BaseField` is always `Cyclotomics`.

- 2 ▶ `CoefficientsRing( Arrangement )` A

The ring containing the coefficients of the polynomials which define the components of the arrangement. Is currently one of: `Integers`, `Rationals`, or `Cyclotomics`.

- 3 ▶ `Dimension( Arrangement )` A

Returns the dimension of the arrangement. This package allows non-essential arrangements.

- 4 ▶ `NrCurves( Arrangement )` A

- ▶ `Cardinality( Arrangement )` M

Returns the number of irreducible components in the arrangement.

### 3.6 Defining Polynomials of an Arrangement

If  $A$  is an arrangement for which the polynomials are known, then `A.1`, `A.2`, ... will return the polynomial of the corresponding component of  $A$ . Correspondingly, `IsBound( A.3 )` will return true if and only if  $A$  has at least 3 components.

- 1 ▶ `CurveDegrees( Arrangement )` A

Returns a list of the degrees of the corresponding components of the arrangement.

- 2 ▶ `Indeterminates( Arrangement )` A
- ▶ `IndeterminateNumbers( Arrangement )` A
- Returns a list of the indeterminates (or their corresponding GAP internal numbers) that are used in the polynomial of the arrangement.
- 3 ▶ `Polynomial( Arrangement )` A
- ▶ `Polynomial( Arrangement, ListOfIndeterminates )` A
- The product of the polynomials which define the components of the *Arrangement*. If a list of indeterminates (may also be the internal GAP indeterminate number) is provided then the polynomial will be expressed in terms of the provided variables.
- 4 ▶ `PolynomialList( Arrangement )` A
- ▶ `PolynomialList( Arrangement, ListOfIndeterminates )` A
- The list of (irreducible) polynomials which define the components of the arrangement. The *PolynomialList* of an arrangement contains no duplicates. If a list of indeterminates (may also be the internal GAP indeterminate number) is provided then the polynomial will be expressed in terms of the provided variables.

### 3.7 Display and Output

- 1 ▶ `Display( Arrangement )` M
- ▶ `Display( Arrangement : Options )` M
- ▶ `Display( Arrangement, List )` M
- ▶ `Display( Arrangement, List : Options )` M

When called with a single argument and no options, all simple information (data consisting of just numbers, polynomials, or lists of numbers) about the arrangement is pretty-printed to the screen. If a *List* is a list of properties or attributes then only the known information that is a subset of that list will actually be displayed. Finally, the following options may be defined which will allow a particular class of traits to be displayed.

**All**

Display all known information about the arrangement.

**Attributes**

Adds all attributes of the arrangement to the displayed output.

**Components**

Adds all components of the arrangement to the displayed output. This will include all internally used data, not all of which is documented.

**Properties**

Adds all properties of the arrangement to the displayed output.

**Simple**

Adds information that is simple to look at, this includes: `Polynomial`, `Multiplicities`, `BettiNumbers`, `Rank`, `Dimension`, `LCSQuotients`, `NrLowIndexSubgroupsPlusConjugates`, `NrLowIndexNormalSubgroups`, `CurveDegrees`, `ArrExponents`

- 2 ▶ `Dump( Arrangement )` M
- Shortcut for `Display(A : All)`
- 3 ▶ `ExtRepOfObj( Arrangement )` M
- Returns a record that is a structural copy of *Arrangement*. All primary objects (lists, matrices, records, ...) will be structural copies.
- 4 ▶ `ViewObj( Arrangement )` M
- Displays basic information about the arrangement. This is the default GAP display function. The output string is some variation of the string `< 3-arrangement on 5 curves >`.

# 4

# Hyperplane Arrangements

These are the user-level functions. See section 1.3 (Objects and Types) for a description of the type names used in this section.

## 4.1 Constructing Hyperplane Arrangements from Scratch

- 1 ▶ `HyperplaneArrangement( PolynomialMatrix )` F
- ▶ `HyperplaneArrangement( PolynomialMatrix : Options )` F
- ▶ `HyperplaneArrangement( : Options )` F
  
- 2 ▶ `CentralHyperplaneArrangement( PolynomialMatrix )` F
- ▶ `CentralHyperplaneArrangement( PolynomialMatrix : Options )` F
- ▶ `CentralHyperplaneArrangement( : Options )` F

Construct a new hyperplane arrangement object. The default way to construct a hyperplane arrangement is from a polynomial matrix. However, additional information may be provided through value options. If the provided options are sufficient to describe the arrangement, then you may omit the polynomial matrix.

Any options which are capable of serving as arrangement constructors (for instance the graph of a graphic arrangement) also have a corresponding `HyperplaneArrangementFrom*` and `CentralHyperplaneArrangementFrom*` functions (for instance, `HyperplaneArrangementFromIntersectionLattice`). All of the “From” functions may also take any of the options that the `HyperplaneArrangement` functions accept.

### Constructive Options

This is the list of options which provide sufficient information to define (for the purposes of this package) a hyperplane arrangement. It may happen that the documentation will fall behind reality, the variable `_arr_hyparr_constructive_options` holds a list of all constructive options known to your version of the package.

`AdjacencyMatrix` := *Matrix*

`AdjacencyVector` := *List*

`AdjacencyVectorIndices` := *List*

These construct a graphic or signed graphic arrangement from the adjacency matrix of its graph. A 1 in position  $(i, j)$  of the adjacency matrix corresponds to the equation  $x_i = \pm x_j$ . Entries above the diagonal correspond to positive edges, entries below the diagonal contribute negative edges, and entries on the diagonal contribute loops (just the component  $x_i = 0$ ).

The adjacency vector is simply the concatenation of the rows of an adjacency matrix, and the adjacency vector indices is simply a list of the nonzero entries in an adjacency vector. Both of the vector constructors will choose the smallest possible dimension that makes sense given your input, but to be sure, you should always specify the dimension when constructing an arrangement from an adjacency vector.

`BraidRank` := *Integer*

*Integer* is the rank of a braid arrangement.

`Graph` := *Graph*

`IntersectionLattice` := *IntersectionLattice*

Provide complete intersection lattice of the arrangement starting at rank 1. The arrangement is assumed to be essential if the dimension is not specified.

`MaximalRank2DegenerateSets` := *DependentSets*

Provide only the non-double point elements of the rank 2 lattice of a central 3-arrangement.

`PolynomialList` := *PolynomialList*

`PolynomialMatrix` := *PolynomialMatrix*

`Rank2Lattice` := *LatticeRow*

Provide only the rank 2 lattice of a central 3-arrangement.

`Rank2ShelledLattice` := *LatticeRow*

Provide only the rank 2 lattice of a central 3-arrangement, but guarantee that the order and labeling of the lattice points are consistent with a shelling of the arrangement.

`SignedGraph` := *Graph*

Construct the arrangement associated to the given *Graph*. If the `Graph` keyword is used then the ordering within each pair will be ignored and all edges will be treated as positive edges. If the `SignedGraph` keyword is used then the ordering within each pair is significant (denoting positive or negative edges depending on whether the values increase or decrease respectively). Also, loops (edges of the form  $[i, i]$ ) are allowed. For a `SignedGraph`, if for all *Graph* pairs,  $[i, j]$ ,  $i < j$  then the returned arrangement will satisfy the filter `IsGraphicArrangement` as if the `Graph` option had been used.

### Non Constructive Options

This is the list of options which serve as additional information about an arrangement, but do not themselves provide sufficient information to define a hyperplane arrangement. It may happen that the documentation will fall behind reality, the variable `_arr_hyparr_nonconstructive_options` holds a list of all non-constructive options known to your version of the package.

`BaseField` := *Field*

Currently, only the Cyclotomics is usable here

`Central`

Requires no parameter, specifies that the arrangement is central. Specifying this option may affect how other options are interpreted.

`CoefficientsRing` := *Ring*

Ring may be one of: Integers, Rationals, `_arr_Reals`, Cyclotomics

`Dimension` := *Integer*

`NrLines` := *Integer*

### Examples

```
# The following two are equivalent
gap> HyperplaneArrangement( : BraidRank := 3 );
< Braid arrangement of rank 3 >
gap> HyperplaneArrangementFromBraidRank( 3 );
< Braid arrangement of rank 3 >
```

```

# Options work both in the HyperplaneArrangement function as well as
# in the corresponding ‘‘From’’ function.
gap> MyPoly := [[1,0,0],[0,1,0],[0,0,1],[0,-1,1],[0,2,3]];;
gap> HyperplaneArrangement( : PolynomialMatrix := MyPoly, Dimension := 3 );
< 3-hyperplane arrangement on 5 lines >
gap> HyperplaneArrangementFromPolynomialMatrix( MyPoly : Dimension := 3 );
< 3-hyperplane arrangement on 5 lines >

# Or equivalently
gap> HyperplaneArrangementFromPolynomialMatrix( MyPoly : Central );
< 3-hyperplane arrangement on 5 lines >

```

## 4.2 Hyperplane Arrangements from Other Arrangements

The following functions are described in the Arrangements chapter (Chapter 3), however can also be applied to hyperplane arrangements: Deletion (3.2.1)

## 4.3 Hyperplane Arrangement Filters

The following filters are described in the chapter (Chapter 3), however can also be applied to hyperplane arrangements: IsEssentialArrangement (3.4.1), IsPlaneCurve (“arrangement:isplanecurve”)

- |     |  |   |
|-----|--|---|
| 1 ▶ | IsCentralArrangement( <i>Arrangement</i> ) | P |
| ▶   | IsAffineArrangement( <i>Arrangement</i> )  | O |

An arrangement is central if each component has a zero constant term Otherwise the arrangement is affine.

- |     |   |   |
|-----|---|---|
| 2 ▶ | IsSupersolvable( <i>HyperplaneArrangement</i> ) | P |
|-----|---|---|

True if the lattice of the arrangement has a maximal chain of modular elements.

## 4.4 Hyperplane Arrangement Invariants

The following functions are described in the Arrangements chapter (Chapter 3), however can also be applied to hyperplane arrangements: Dimension (3.5.3), NrCurves (3.5.4), Cardinality (3.5.4).

## 4.5 Defining Polynomials of a Hyperplane Arrangement

If  $A$  is an arrangement for which the polynomials are known, then  $A.1$ ,  $A.2$ ,  $\dots$  will return the polynomial of the corresponding component of  $A$ . Correspondingly,  $\text{IsBound}( A.3 )$  will return true if and only if  $A$  has at least 3 components.

The following functions are described in the Arrangements chapter (Chapter 3), however can also be applied to hyperplane arrangements: PolynomialList (3.6.4), Polynomial (3.6.3)

## 4.6 Intersection Lattice of a Hyperplane Arrangement

## 4.7 Fundamental Group of a Hyperplane Arrangement

## 4.8 Homology and Cohomology Invariants

# 5

# The Fox Calculus

1 ▶ `Augmentation( Matrix )` M

Applies the augmentation map to each element of a matrix. The augmentation map takes all generators of a group ring to One.

2 ▶ `FoxDerivative( g, i[, F ] )` F

▶ `FreeFoxDerivative( g, i[, F ] )` F

Computes the Fox derivative of  $g$  which may be an element of a group or of a group ring with respect to  $i$ .  $i$  may be a group generator, an integer, or a list of group generators or integers. The derivatives will be taken sequentially starting from the front.

$F$  is a list of converters which will be applied to the resulting derivative. The first converter must accept groups elements.

The `FoxDerivative` works in the group ring  $\mathbb{Z}G$  (where  $G$  is the group containing  $g$ ) whereas the `FreeFoxDerivative` computes in the group ring  $\mathbb{Z}F$  where  $F$  is the free group on the generators of  $G$ .

3 ▶ `FoxHessian( r, n[, F ] )` F

▶ `FreeFoxHessian( r, n[, F ] )` F

Computes the Fox Hessian of  $g$  which may be an element or relator of a group or element of a group ring. The integer  $n$  controls the size of the Hessian computed. The result of this function is a matrix,

$$H_{ij} = \frac{\partial}{\partial g_j} \frac{\partial}{\partial g_i} r$$

Where  $g_1, \dots, g_n$  are generators of the group containing  $r$ .

$F$  is a list of converters which will be applied to the resulting derivatives. The first converter must accept groups elements.

The `FoxHessian` works in the group ring  $\mathbb{Z}G$  (where  $G$  is the group containing  $g$ ) whereas the `FreeFoxHessian` computes in the group ring  $\mathbb{Z}F$  where  $F$  is the free group on the generators of  $G$ .

4 ▶ `FoxJacobian( FpGroup[, F ] )` M

▶ `FoxJacobian( Arrangement[, F ] )` M

▶ `FreeFoxJacobian( FpGroup[, F ] )` M

▶ `FreeFoxJacobian( Arrangement[, F ] )` M

Computes the Fox Jacobian of a group  $G$  or of the fundamental group of the complement of an arrangement. The result of this function is a matrix,

$$J_{ij} = \frac{\partial r_i}{\partial g_j}$$

Where  $g_1, \dots, g_n$  are generators and  $r_1, \dots, r_m$  are relators of the group.

$F$  is a list of converters which will be applied to the resulting derivatives. The first converter must accept groups elements.

The `FoxJacobian` works in the group ring  $\mathbb{Z}G$  whereas the `FreeFoxJacobian` computes in the group ring  $\mathbb{Z}F$  where  $F$  is the free group on the generators of  $G$ .

5 ▶ `QuotientHomomorphism( FpGroup )`

Returns the homomorphism which projects the free group onto the given group.

## 5.1 Fox Calculus Helpful Conversions

- 1 ▶ `_fox_augmentation()` F  
 Produces the augmentation function:  $r \rightarrow 1$ .
- 2 ▶ `_fox_group2gens( L )` F  
 Produces function converting group elements to the object generated by  $L$  by taking the group generators  $g_i \mapsto L[i]$ .
- 3 ▶ `_fox_group2group( G )` F  
 Produces function converting group elements to elements of another group  $G$  by taking the generators of existing group  $H$  to the generators of  $G$ .
- 4 ▶ `_fox_ints2gens( L )` F  
 Produces function converting a letter representation to elements of the object generated by  $L$  by taking  $i \mapsto L[i]^{(\text{Sign}(i) * 1)}$ . Thus, The elements of  $L$  must be invertable.
- 5 ▶ `_fox_ints2group( G )` F  
 Produces function converting a letter representation to elements of the group  $G$ .
- 6 ▶ `_fox_ints2ring( R )` F  
 Produces function converting a letter representation to elements of the ring  $R$ .

## 5.2 Fox Calculus Utility Functions

- 1 ▶ `__FoxJacobian( G, L )` M  
 ▶ `__FoxJacobian( G, L : AsRingElements )` M  
 ▶ `__FoxDerivative( e, i, L )` M  
 ▶ `__FoxDerivative( e, i, L : AsRingElements )` M  
 ▶ `__FoxHessian( e, n, L )` M  
 ▶ `__FoxHessian( e, n, L : AsRingElements )` M

Computes the indicated operations and then applies the functions given in  $L$  to each word in the result. The form of elements in the group ring in the absence of any functions in  $L$  is a list of the form, [ `[word list], coeff, [word list], coeff, ...` ] with each “arrangement:word list” being a list of signed integers  $x_1^{i_1} x_2^{i_2} \dots x_n^{i_n}$  (I.E. `LetterRepAssocWord(p)`) and `coeff` is the coefficient in the group ring.

However, in all cases, the functions in  $L$  are only ever given individual words (terms without the coefficients). This means that  $L$  can contain things like group homomorphisms. This package defines several `_fox_*` functions that can be useful in this context.

If the application of the functions in  $L$  result in ring elements (that is, addition is allowed) then the option `AsRingElements` may be provided and the terms and coefficients will be combined to produce a proper matrix.

The value of `i` may be either an index, list of indices, generator, or list of generators and represent the derivatives which are to be taken. For the Fox Hessian, `n` is the number of generators of the group (or group ring) containing `e`.

```

gap> F := FreeGroup( 3 );
gap> G := F / [ F.1*F.2^2*F.1^-1*F.2, F.1*F.3^5*F.1^-1*F.3^3 ];
gap> f := MaximalAbelianQuotient(G);
gap> __FoxJacobian(G, []);
gap> __FoxJacobian(G, [_fox_ints2group(G)]);
gap> __FoxJacobian(G, [_fox_ints2group(G), f]);

```

2 ▶ `_arr_canonical_generators_of_algebra( A )` F

The GAP command `GeneratorsOfAlgebra` returns a list `[ One(A), A.1, A.1^-1, ... ]`. This function returns just the list `[ A.1, A.2, ... ]`.

3 ▶ `_arr_combine_like_terms( L )` F

▶ `_arr_combine_like_terms( L : FoxThoroughCombineLikeTerms )` F

Takes an element/coefficient list (`[ element, coeff, ... ]`) and combines the coefficients of identical elements. If any of the elements satisfy the filter `IsElementOfMagmaRingModuloRelations` or the option `FoxThoroughCombineLikeTerms` is provided then elements will be compared using an equality test `=`. Otherwise, elements will be compared using a plain string equality test that does not attempt any simplification and is consequentially likely to be much faster.

4 ▶ `_arr_compute_fox_derivative( e, x )` F

Compute the derivative of  $e$  with respect to  $x$ .  $x$  must be a single element (not a list), however,  $e$  and  $x$  are allowed to be in any form (group `ExtRep`/coefficient lists, elements, ...).

5 ▶ `_arr_compute_fox_hessian( e, n )` F

Compute the hessian of the element  $e$ .  $e$  is allowed to be in any form (group `ExtRep`/coefficient lists, elements, ...).

$$H_{ij} = \frac{\partial}{\partial g_j} \frac{\partial}{\partial g_i} e$$

6 ▶ `_arr_compute_fox_jacobian( rels, n )` F

Return a matrix of derivatives of each relation in `rels`,

$$J_{ij} = \frac{\partial rels_i}{\partial g_j}$$

where  $g_j$  generate the group whose relators are in `rels`. `rels` are allowed to be in any form (group `ExtRep`/coefficient lists, elements, ...).

7 ▶ `_arr_container_of_fox_thingun( x )` F

If  $x$  is a group element or a list whose first element is a group element then this function returns the wholeGroup  $G$  containing  $x$ . (NOTE: Even if  $x$  is expressed as an element of a subgroup  $H < G$  then this function will return  $G$  and not  $H$ )

8 ▶ `_arr_fox_apply_maps( e, [ f_1, f_2, ..., f_n ] )` F

Computes  $(f_n \circ f_{n-1} \circ \dots \circ f_1)(e)$  even if  $e$  is an element/coefficient list (`[ element, coeff, ... ]`). The return value is always a list. The return value will be a list of a single element if  $e$  is a normal element.

9 ▶ `_arr_map_matrix( M, f )` F

Apply the function  $f$  to each element of the matrix  $M$  and return the resulting matrix.

- 10 ▶ `_arr_prod_sum( e )` F  
 ▶ `_arr_prod_sum( e : Zero := zero )` F

Flattens  $e$  to an actual element when  $e$  is given as an element/coefficient list (`[ element, coeff, ... ]`). The elements of the list need to be elements of a ring for this operation to succeed. If  $e$  is empty `zero` will be returned if it is provided, otherwise 0 is returned.  $e$  must be a list of even length.

- 11 ▶ `_arr_words_to_list( x )` F

Ensures that  $x$  is represented as an element/coefficient list with the elements expressed in their letter representation. That is, they must have the form:

`[ [word list], coeff, [word list], coeff, ... ]`

Where a “word list” is a list of signed integers  $x_1^2 * x_2^{-1} \rightarrow [1, 1, -2]$  as in the output of “`arrangement:letterrepassocword`”.

$x$  is allowed to be already in this form, a word, an element of an fp group, or an element of a magma ring modulo relations (“`arrangement:iselementofmagramaringmodulorelations`”).

- 12 ▶ `_fox_group2ring( R )` F

Produces function converting group elements to elements of the ring  $R$  by taking the generators of  $G$  to the generators of  $R$ .

- 13 ▶ `_fox_ring2gens( L )` F

Produces function converting ring elements to the object generated by  $L$  by taking the ring generators  $r_i \mapsto L[i]$ .

- 14 ▶ `_fox_ring2group( G )` F

Produces function converting ring elements to elements of the group  $G$  by taking the generators of  $R$  to the generators of  $G$ .

- 15 ▶ `_fox_ring2ints()` F

Produces function converting ring elements to a letter representation (list of integers representing generator numbers) as “`arrangement:letterrepassocword`” does for words.

# 6

# Utility Commands

These commands were created for this package, but may be useful more generally as well.

## 6.1 General Utility

1 ▶ `Ceil( num )` F

Returns the smallest integer  $c$  so that  $num \leq c$ .

2 ▶ `Chop( List )` F

▶ `Chop( List, Integer )` F

▶ `ChopMat( Matrix )` F

▶ `ChopMat( Matrix, Integer )` F

Removes the last *Integer* entries (resp. columns) from *List* (resp. *Matrix*). If *Integer* is not specified, then only the last element (column) is removed.

The return of `Chop` depends on whether *Integer* was provided. The removed element is returned if *Integer* is absent. Otherwise a list of the chopped elements is returned (even if *Integer* = 1). `ChopMat` returns a list of the corresponding return values of `Chop` (thus the return value will be a list if *Integer* is not provided, and a matrix if it is).

Note: *Integer* columns will be removed from each row of *Matrix* even if some rows are longer than others. Thus, the “column” picture will not be entirely accurate if *Matrix* is a `Table` and not a `Matrix`.

3 ▶ `DeepCopyListOfNumbers( List )` F

▶ `DeepCopyListOfNumbers( List, Integer )` F

Makes a deep copy of the *List* up to *Integer* levels deep. The default is a maximum of 100 levels. `DeepCopyListOfNumbers( L, 1 )` is the same thing as `ShallowCopy( L )`. All entries and decedents of *List* must satisfy either the filter `IsList` or `IsCyclotomic` (which includes `IsInt` and `IsRat`).

4 ▶ `DeepSort( List )` F

Sort a list deeply. This can be useful for a poor-man’s canonicalization of a matroid or lattice. Just be sure to call `List(L, DeepSort)` for lattices or else your ranks will get rearranged.

5 ▶ `DeepSortedList( List )` F

Sort a list deeply making a deep copy at the same time. This can be useful for a poor-man’s canonicalization of a matroid or lattice. Just be sure to call `List(L, DeepSort)` for lattices or else your ranks will get rearranged.

6 ▶ `EvenPositions( List )` F

▶ `OddPositions( List )` F

Returns a sublist of *List* consisting only of the Even (resp. Odd) entries.

7 ▶ `FirstPos( List, Function )` F

Return the position of the first element of *List* for which *Function* returns true.

- 8 ► `FisherYatesShuffle( List )` F  
 ► `FisherYatesShuffleDestructive( List )` F

Perform a Fisher-Yates shuffle on the *List*. If the non-destructive form is called, a shallow copy is made before shuffling.

- 9 ► `FpGroupDirectProductFpGroups( G1, G2, ... )` F

Construct a direct product of finitely presented groups and return a new group whose generators are the generators of the *Gi*'s and relations are the relations of the *Gi*'s together with commutation relations between the groups.

Note that the generators of the groups are not actually identical to the generators of the direct product. You can create a homomorphism from each group into the direct product using the `GroupHomomorphismByImages` command:

```
gap> G := Image(IsomorphismFpGroup(Group( (1,2), (1,2,5,7) )));
<fp group of size 24 on the generators [ F1, F2, F3, F4 ]>
gap> H := Image(IsomorphismFpGroup(Group( (1,5), (2,3,1), (1,9) )));
<fp group of size 120 on the generators [ F1, F2, F3 ]>
gap> P := FpGroupDirectProductFpGroups( G, H );
<fp group on the generators [ f1, f2, f3, f4, f5, f6, f7 ]>

gap> genG := GeneratorsOfGroup(G);
[ F1, F2, F3, F4 ]
gap> genH := GeneratorsOfGroup(H);
[ F1, F2, F3 ]
gap> genP := GeneratorsOfGroup(P);
[ f1, f2, f3, f4, f5, f6, f7 ]

gap> GtoP := GroupHomomorphismByImagesNC( G, P, genG, genP{[1..4]} );
[ F1, F2, F3, F4 ] -> [ f1, f2, f3, f4 ]
gap> HtoP := GroupHomomorphismByImagesNC( H, P, genH, genP{[5..7]} );
[ F1, F2, F3 ] -> [ f5, f6, f7 ]
```

- 10 ► `KroneckerDelta( i, j )` F

For *i* and *j* any objects, returns 1 if *i* = *j* and 0 otherwise.

- 11 ► `Last( List[, Function ] )` F

Either returns the last element of *List*, or the last element for which *Function* returns true.

- 12 ► `LastPos( List, Function )` F

Return the position of the last element of *List* for which *Function* returns true.

- 13 ► `LatticeDiagram( IntersectionLattice )` F

Returns a record with two entries: `lattice` and `connections`. The `lattice` entry is nothing but a mutable copy of the original intersection lattice. The `connections` however, are a list of connections [ *i*, *j* ] which says that the *j*th object in the lattice contains the *i*th entry of the lattice, where we label `L[1][1] -> 1`, `L[1][2] -> 2`, and so on, essentially flattening the first dimension of *L*.

- 14 ► `Reduce( L, f )` F

Reduces a list *L* by calling a function *f* multiple times with two arguments each time. The first call will be with the first two elements of the list, subsequent calls will be done by setting the first argument to the result of the previous call and the second argument to the next element in the list.

Returns the result of the last call to  $f$ . If  $L$  is empty then `fail` is returned. If  $L$  only contains one element then that element is returned and  $f$  is not executed.

```
ints := List([1..10], i -> Random(Integers));
sum  := Reduce(ints, \+);
gcd  := Reduce(ints, GCD_INT); # common GCD
min  := Reduce(ints, function(a,b) if a < b then return a; else return b; fi; end);
```

15 ▶ `ReduceTensorRank2( RectangularTable )` F

Reduce the rank of a generalized tensor product (with rank  $r \geq 4$ ) with one of rank  $r - 2$  (to use the Mathematica terminology).

```
gap> MM := [ [ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 5, 6 ], [ 7, 8 ] ] ],
>          [ [ [ 9, 10 ], [ 11, 12 ] ], [ [ 13, 14 ], [ 15, 16 ] ] ] ];
gap> PrintArray(MM);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 5, 6 ], [ 7, 8 ] ] ],
[ [ [ 9, 10 ], [ 11, 12 ] ], [ [ 13, 14 ], [ 15, 16 ] ] ]
gap> PrintArray(ReduceTensorRank2(MM));
[ [ 1, 2, 5, 6 ],
[ 3, 4, 7, 8 ],
[ 9, 10, 13, 14 ],
[ 11, 12, 15, 16 ] ]
```

16 ▶ `Shift( MutableList )` O

Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns `fail`.

17 ▶ `SortArrangementLattice( Lattice )` F

Canonicalizes the intersection lattice of a hyperplane arrangement so that it can be compared to other lattices using GAP's "arrangement:=".

18 ▶ `Splice( MutableList[, Offset[, Length[, List2 ] ] ] )` F

Removes the elements designated by *Offset* and *Length* from a mutable list, and replaces them with the elements of *List2*, if any. Returns the elements removed from the list. The list grows or shrinks as necessary. If *Offset* is negative then it starts that far from the end of the array. If *Length* is omitted, removes everything from *Offset* onward. If *Length* is negative, removes the elements from *Offset* onward except for  $-Length$  elements at the end of the array. If both *Offset* and *Length* are omitted, removes everything.

19 ▶ `WittFormula( k, n )` F

Returns the  $k, n$  Witt number:  $\frac{1}{k} \sum_{d|k} \mu(d) n^{\frac{k}{d}}$ .

20 ▶ `Zip( L1, L2, ... )` F

Zip the lists together like a zipper. That is, take the first element from each list followed by the second element from each list, .... Any lists that are shorter than the other lists are simply bypassed when their elements are exhausted.

```
gap> Zip([1,2], [], [3,4,5]);
[ 1, 3, 2, 4, 5 ]
```

## 6.2 Multivariable Polynomials

- 1 ► `ConstantTerm( P )` M  
 Returns the constant term of the (possibly multivariate) polynomial, cyclotomic or finite field element  $P$ .
- 2 ► `Degree( P )` M  
 Compute the total degree of a (possibly multivariate) polynomial  $P$ .
- ```
gap> x := Indeterminate(Integers, "x");;
gap> y := Indeterminate(Integers, "y");;
gap> Degree( x*y^2 + 3*x - 2*y );
3
```
- 3 ► `IndeterminateNumbers( P )` M  
 Return the list of indeterminate numbers (see “arrangement:indeterminate”) of a (possibly multivariate) polynomial, rational function, cyclotomic or finite field element  $P$ .
- 4 ► `Indeterminates( P )` M  
 Return the list of indeterminates (see “arrangement:indeterminate”) of a (possibly multivariate) polynomial, rational function, cyclotomic or finite field element  $P$ .
- 5 ► `IsHomogeneous( P )` M  
 Returns `true` if the the degrees of each term of the (possibly multivariate) polynomial  $P$  are all identical.
- 6 ► `IsMonomial( F )` M  
 Returns `true` if the polynomial or rational function  $F$  is a monomial.
- 7 ► `MonomialDenominatorLCM( L )` F  
 If  $L$  is a list of rational functions (or numbers) whose denominators are all monomials then this function will return the `MonomialLCM` of the denominators.
- 8 ► `MonomialLCM( L )` F  
 Given a list of monomials, this function computes their Least Common Multiple. The unique monomial  $m$  of lowest total degree for which each of the monomials in  $L$  divides  $m$ . If  $L$  is empty then 1 is returned.
- 9 ► `NrIndeterminates( P )` M  
 Compute the number of unique indeterminates in the (possibly multivariate) polynomial, rational function, cyclotomic or finite field element  $P$ .

# 7

# Internally Used Commands

Most of the internally used commands start with the phrase: `_arr_`.

## 7.1 File related commands

- 1 ▶ `_arr_Use( string )` F  
Finds the file *string* in “arrangement:—arr—dirs” and Reads the file.
- 2 ▶ `_arr_files V`  
The list of support files which should be loaded by the package. The `.gd` and `.gi` extensions should be left off of the entries in this array.
- 3 ▶ `_arr_dirs V`  
A list of directories which may contain the files listed in “arrangement:—arr—files”. This variable is only useful if the package is being loaded via the `Read` command rather than loaded as a proper package using `LoadPackage` or `RequirePackage`.

## 7.2 User interaction

- 1 ▶ `ArrangementInfo V`  
The info class for the arrangement functions and methods (see “arrangement:info functions”).

## 7.3 Arrangement.g

- 1 ▶ `_arr_arrangement_from_hash( Record )` F
  - ▶ `_arr_arrangement_from_hash( Record, ListofFilters )` F

Blesses a record into the category of an arrangement. If a second argument is provided, then the indicated filters will be forced to be satisfied.
- 2 ▶ `_arr_delete_line( Record, Integer )` F  
This function does the work of deleting a curve from an arrangement. It attempts to preserve as much information as is possible. The return value is a new record with the data modified to contain one less curve. All indices (for instance in the intersection lattice) are re-indexed by shifting all values greater than *Integer* down by one.
- 3 ▶ `_arr_hyperplane_arrangement_from_hash( Record )` F
  - ▶ `_arr_hyperplane_arrangement_from_hash( Record, ListofFilters )` F

Blesses a record into the category of a hyperplane arrangement. If a second argument is provided, then the indicated filters will be forced to be satisfied.

- 4 ▶ `_arr_register_constructor( Name, Constructor )` F  
 ▶ `_arr_register_constructor( Name, Constructor : NonConstructive )` F

Define a new option that can be passed to the `Arrangement` user-level constructor. *Name* is the string name of the option and *Constructor* is a function which takes a single record argument. The *Constructor* should modify this argument based on the values of the `ValueOptions` when the constructor is called. The *Constructor* should also return a list of filters which the final arrangement will be known to satisfy.

Registered *Constructors* are assumed to provide some minimal information about an arrangement. In particular, the dimension and the number of curves. If this is not the case, then the option `NonConstructive` should be included to prevent the creation of global `From` functions of the name `ArrangementFromName` and `CentralArrangementFromName` (respectively `HyperplaneArrangementFromName` and `CentralHyperplaneArrangementFromName`).

- 5 ▶ `_arr_register_hyparr_constructor( Name, Constructor )` F  
 ▶ `_arr_register_hyparr_constructor( Name, Constructor : NonConstructive )` F

Define a new option that can be passed to the `HyperplaneArrangement` user-level constructor. *Name* is the string name of the option and *Constructor* is a function which takes a single record argument. The *Constructor* should modify this argument based on the values of the `ValueOptions` when the constructor is called. The *Constructor* should also return a list of filters which the final arrangement will be known to satisfy.

Registered *Constructors* are assumed to provide some minimal information about an arrangement. In particular, the dimension and the number of curves. If this is not the case, then the option `NonConstructive` should be included to prevent the creation of global `From` functions of the name `ArrangementFromName` and `CentralArrangementFromName` (respectively `HyperplaneArrangementFromName` and `CentralHyperplaneArrangementFromName`).

## 7.4 braid.g

- 1 ▶ `_brd_braid_monodromy_from_shelled_lattice( IntersectionLattice )` F

We only make use of the rank 2 sublattice. The return value is a record with two entries: `group`, a braid group, and `monodromy`, words in the `group` representing the braid monodromy.

## 7.5 graphs.g

- 1 ▶ `_gph_add_graph_edges( A, edges )` F

Add the edges (list of unordered pairs,  $[i, j]$ ) to the adjacency matrix *A*. The size of *A* will be altered if any edges involve vertex indices not included in *A*.

- 2 ▶ `_gph_add_signed_graph_edges( A, edges )` F

Add the signed edges (list of ordered pairs,  $[i, j]$ ) to the adjacency matrix *A*. The size of *A* will be altered if any edges involve vertex indices not included in *A*.

- 3 ▶ `_gph_complete_subgraphs( A, n )` F

Returns a list of all vertex sets which form a complete graph on *n* vertices within the graph described by the adjacency matrix *A*.

Note: The `CompleteSubgraphs` function in the `GRAPE` package is faster than this subroutine for  $n > 4$ . When  $n \leq 4$  this subroutine tends to be faster but is not always. For somewhat large graphs (30 or more vertices) and larger *n*, this function performs very poorly and should not be used.

4 ▶ `_gph_distance( A, i, j )` F

Return the distance between vertex  $i$  and vertex  $j$  in the graph represented by the adjacency matrix  $A$ . If you intend to do more than a couple of distance lookups on the same graph use the `_gph_distance_matrix` instead.

5 ▶ `_gph_distance_matrix( A )` F

▶ `_gph_distance_matrix( A, m )` F

▶ `_gph_distance_matrix( A, m : DoNotWaitForLoops )` F

Return a matrix whose  $(i, j)$  component is the length of a minimal path from vertex  $i$  to vertex  $j$  in the graph represented by the adjacency matrix  $A$  (directed edges are respected). If  $m$  is provided then only paths up to length  $m$  will be considered (vertices requiring a longer path will have a 0 in the corresponding entry of the distance matrix).

If the option `DoNotWaitForLoops` is present then the diagonal of the distance matrix will be ignored by this function and will be whatever value appears in the adjacency matrix.

6 ▶ `_gph_graph_adjacency_matrix( G )` F

▶ `_gph_graph_adjacency_matrix( G, n )` F

Construct an adjacency matrix for the unsigned graph (possibly with loops)  $G$ . The constructed matrix  $A \in \{0, 1\}^{n^2}$  has a 1 in the  $(i, j)$  position if and only if there is an edge  $[i, j]$  or  $[j, i]$  in  $G$ . If  $n$  is not provided it is taken to be the maximal vertex in  $G$ ,

7 ▶ `_gph_is_connected( A )` F

Return true if the graph is strongly connected and false otherwise.

8 ▶ `_gph_nr_complete_subgraphs( A, n )` F

Returns the number of all vertex sets which form a complete graph on  $n$  vertices within the graph described by the adjacency matrix  $A$ .

Note: The `CompleteSubgraphs` function in the GRAPE package is faster than this subroutine for  $n > 4$ . When  $n \leq 4$  this subroutine tends to be faster but is not always. For somewhat large graphs (30 or more vertices) and larger  $n$ , this function performs very poorly and should not be used.

9 ▶ `_gph_signed_graph_adjacency_matrix( G )` F

▶ `_gph_signed_graph_adjacency_matrix( G, n )` F

Construct an adjacency matrix for the signed graph (possibly with loops)  $G$ . The constructed matrix  $A \in \{0, 1\}^{n^2}$  has a 1 in the  $(i, j)$  position if and only if there is an edge  $[i, j]$  in  $G$ . If  $n$  is not provided it is taken to be the maximal vertex in  $G$ ,

10 ▶ `_gph_width( A )` F

▶ `_gph_width( A : DoNotWaitForLoops )` F

Return the width (diameter) of the graph represented by the adjacency matrix  $A$ . The width of a connected graph is the largest entry in the distance matrix. If the graph is not connected this function returns `fail`.

If the option `DoNotWaitForLoops` is present then the diagonal of the distance matrix (paths from a vertex to itself) will be ignored by this function. You will probably want to use this for normal undirected graphs without vertex loops since in that situation any vertex with an edge would register as having distance 2 from itself.

Note: This function seems to be considerably faster than the `Diameter` method in the GRAPE package.

## 7.6 cohomology.g

- 1 ▶ `_arr_lattice_boundary_operator( ListOfIntegers )` F  
 ▶ `_arr_lattice_boundary_operator( ListOfIntegers, String )` F  
 ▶ `_arr_lattice_boundary_operator( ListOfIntegers, ListOfStrings )` F  
 ▶ `_arr_lattice_boundary_operator( ListOfIntegers, ListOfRingGenerators )` F

Computes the boundary operator on a list of integers. That is,  $\partial(e_{i_1} \wedge e_{i_2} \wedge \dots \wedge e_{i_n}) := \sum_{j=1}^n (-1)^{j-1} \prod_{k \neq j} e_{i_k}$ . The list  $[i_1, \dots, i_n]$  should be given as the first argument to this function. The return value depends on the second argument. In the first case, the return value is (nearly) the output of `ExtRepOfObj` on an element of a finitely presented associative algebra with one. The second and third forms return a string suitable for pasting into other applications (for instance Macaulay). The last form returns an actual ring element.

```
gap> _arr_lattice_boundary_operator( [1,2,3] );
[ [ 2, 1, 3, 1 ], 1, [ 1, 1, 3, 1 ], -1, [ 1, 1, 2, 1 ], 1 ]
gap> _arr_lattice_boundary_operator( [1,2,3], "e_" );
"e_2*e_3-e_1*e_3+e_1*e_2"
gap> _arr_lattice_boundary_operator( [1,2,3], "e" );
"e2*e3-e1*e3+e1*e2"
gap> _arr_lattice_boundary_operator( [1,2,3], ["a","b","c"] );
"b*c-a*c+a*b"

gap> R := FreeAssociativeAlgebraWithOne(Integers, 3);
<free left module over Integers, and ring-with-one, with 3 generators>
gap> gens := GeneratorsOfAlgebraWithOne(R);
[ (1)*x.1, (1)*x.2, (1)*x.3 ]
gap> _arr_lattice_boundary_operator( [1,2,3], gens );
(1)*x.1*x.2+(-1)*x.1*x.3+(1)*x.2*x.3
```

## 7.7 face lattice.g

- 1 ▶ `_arr_face_lattice( L, M )` F

Constructs the face lattice from the intersection lattice  $L$  and polynomial matrix  $M$ . The arrangement must be real and essential. The returned lattice is a list (of length  $1 + \text{Rank}(L)$ ) of records whose keys are the face vectors and values are records with entries `Flat`, `LatticePos`, and `Representative`. The `Representative` is a point in  $\mathbb{C}^n$  whose face vector corresponds to the face vector of the record. The other two values relate the record back to the original lattice. `Flat` is the intersection lattice flat that contains the face and `LatticePos` is the coordinates  $[i, j]$  of the flat within the intersection lattice.

- 2 ▶ `_arr_face_lattice_merge( v, pfv, pR, MM, FL, flat, i, j )` F

Arguments are: a vector  $v$ , parent face vector  $pfv$ , parent face record  $pR$ , augmented matrix (with constants at the end of each row)  $MM$ , and record of face records  $FL$  (of rank 1 less than  $pfv$ ), flat from the intersection lattice which the new face should lie in  $flat$ , the coordinates  $i, j$  of the flat within the intersection lattice. Then  $P := pR.Representative + \epsilon v$  will be inserted into  $FL$ . If the face represented by  $P$  is already represented in  $FL$ , this function will try to determine which representative is nicer.

- 3 ▶ `_arr_face_vector_not_too_far( u, v )` F

A very particular function. If  $u$  and  $v$  are two face vectors with  $u > v$  then this subroutine will return `true` if they are adjacent, and `false` otherwise.

Since this function does not take the defining forms, or even the intersection lattice, into consideration, the best check that it can make is that the only difference between the vectors is that some of the zeroes in  $u$  are  $+$  or  $-$  in  $v$ . If  $u$  is known to be a degeneration of  $v$  then this test is sufficient. Otherwise, it doesn't tell you anything.

4 ▶ `_arr_lattice_orthogonal_vector( W, V, M )` F

If  $V$  and  $W$  are lattice flats (lists of integers) such that  $V > W$  and  $\text{rank}(V) = \text{rank}(W) + 1$  then this function constructs a vector  $v$  normal to the subspace  $V$  so that  $v$  also lies in the subspace  $W$ . This vector is unique up to scalar multiple.  $M$  is the defining forms of the arrangement.

5 ▶ `_arr_sign_vector( p, M )` F

Compute the sign vector of a point  $p$  given a matrix of real linear forms  $M$  and returns the list of signs as a string composed of the characters '+', '-', and '0'.

```
gap> M := [[1,2,-3],[1,2,1],[1,0,1]];
[ [ 1, 2, -3 ], [ 1, 2, 1 ], [ 1, 0, 1 ] ]
gap> V := [1,-2,3];
[ 1, -2, 3 ]
gap> _arr_sign_vector( V, M );
"-0+"
```

## 7.8 matroid.g

1 ▶ `_arr_canonicalize_polynomial( P )` F

Very basic function which turns constant rational functions into scalars. Otherwise,  $P$  is returned unmodified.

2 ▶ `_arr_find_four_in_general_position( M, n )` F

If  $M$  is a rank 3 matroid expressed as minimal dependent sets (triples of integers) and  $n$  is the number of points in the matroid, then this function will return a list of four integers for which no three are dependent. If this is not possible for  $M$  then the function will return `false`.

3 ▶ `_arr_realization_defining_vectors( N, Mo, M, L, p )` F

Returns either `fail` or a record which describes point  $p$  in terms of the already determined points listed in  $L$ . This function depends on  $M$  and  $Mo$  being deeply sorted.

4 ▶ `_arr_realization_do_dependent_sets( W, R, char, zero, one )` F

```
W - a matroid elements as triples of dependent sets [i,j,k]
R - sparse list of actual vectors
char - value which the characteristic of the realization field must divide
zero - A list which will have polynomials appended to it
one - One( appropriate ring element ) = 1 or One(x)
```

All elements in  $W$  must be composed of vector indices which are defined in  $R$ . This function computes the determinant of each triple of vectors in  $R$  and either updates  $char$  or adds the determinant,  $d$ , to  $zero$  depending on whether  $d$  is an integer or a polynomial. This function makes some attempt at simplifying  $d$  by factoring out common powers of  $x_i$  (since we need  $x_i \neq 0$  elsewhere) and searching  $zero$  for scalar multiples of  $d$  before adding  $d$  to  $zero$ . If a scalar multiple is found then the existing polynomial is replaced with an appropriate scalar multiple so that finite characteristic realizations are correct.

5► `_arr_realization_expand_determined_vectors( N, Mo, M, L, R, char, zero, one )` F

```

N := [1..n]; # n is number of points in the matroid
Mo - "Original" matroid as triples of dependent sets
      ([i,j,k] in M \<=> Det(R{[i,j,k]}) = 0)
M - "Current" / unspent matroid elements
L - list of indices of the known vectors (indices of R)
R - "sparse" list of actual vectors
char - value which the characteristic of the realization field must divide
zero - A list which will have polynomials appended to it
one - either the number 1 or One(x) (one in the polynomial ring)

```

Repeatedly applies the observation [Sturmfels/CompSynthGeo]:

If  $A = \{a, b, c\}$  and  $B = \{a, d, e\}$ ,  $A, B \in M$ ,  $a \notin L$ , and planes spanned by the vectors in  $A$  and  $B$  intersect in a line (spanned by  $a$ ) then,  $a \in \text{span}\{|bcd|e + |cbe|d\}$ .

All variables except for  $N$  and  $char$  will be altered by this function.  $M$  will have spent dependent sets removed from it,  $L$  will have new indices added to it,  $R$  will have the corresponding vectors added to it (in the appropriate locations).

The return value of this function is the new “ $char$ ”, that is, the characteristic of the field that we know we must restrict ourselves to. If ever  $char = 1$  then this function will immediately stop and return 1.

6► `_arr_realization_introduce_double_variables( N, M, L, R, X )` F

```

N := [1..n] - where n is number of points in the matroid
M - a matroid as triples of dependent sets [i,j,k] in M <=> Det(R{[i,j,k]}) = 0
L - list of indices of the known vectors (indices of R)
R - "sparse" list of actual vectors
X - list of indeterminants used so far

```

Choose a point in  $S := N \setminus L$  and realize it as the vector  $[1, x, y]$ . This function assumes that none of the points in  $S$  are in the plane spanned by any pair of points in  $L$ .

If function will append the chosen point to  $L$ , insert its realization into  $R$ , and appends two new indeterminates to  $X$ .

Note: We know that the first component of any vector not defined when this function is executed has no components which are zero (in particular the first) because all of our realizations start with  $e_1, e_2, e_3$ . Thus, any vector with a zero component lies in the span of two of these vectors. Thus, we are allowed to return  $[1, x, y]$ .

7► `_arr_realization_introduce_single_variable( N, M, L, R, char, zero, X )` F

```

N := [1..n] - where n is number of points in the matroid
M - a matroid as triples of dependent sets [i,j,k] in M <=> Det(R{[i,j,k]}) = 0
L - list of indices of the known vectors (indices of R)
R - "sparse" list of actual vectors
char - value which the characteristic of the realization field must divide
zero - A list which will have polynomials appended to it
X - list of indeterminants used so far

```

Choose a point in  $S := N \setminus L$  which is in a plane spanned by two of the vectors in  $R$ . This function assumes that none of the points in  $S$  are uniquely determined by the points in  $L$ .

If this function succeeds then it will append the chosen point to  $L$ , insert its realization into  $R$ , remove the spent relations from  $M$  and return the (possibly different) characteristic. If the function fails, then it will return `fail`. If the function determines that the matroid is not realizable, then it will return `false`.

Note: If the  $char$  is non-zero then this function will reduce its realizations mod  $char$ .

## 7.9 polynomials.g

1 ▶ `_arr_find_transformation_to_x_n( l )` F

Given a vector  $l$ , find a linear transformation,  $M$ , which will,

- transform  $l$  to a multiple of  $[0, 0, \dots, 1]$  when multiplied on the right:  $lM$
- be a matrix in the field of fractions of the ring of  $l$

2 ▶ `_arr_make_unique_indeterminants( [ R, ]n[, NAM ] )` F

Create  $n$  new indeterminates in the ring  $R$  (`Integers` by default) with the names  $prefix1, \dots, prefixn$  if  $NAM$  is a string and  $NAM[1], \dots, NAM[n]$  if  $NAM$  is a list of strings. If  $NAM$  is not provided then the names “x”, “y”, and “z” will be used if  $n \leq 3$  otherwise the names “x\_1”, ..., “x\_n” will be used.

```
gap> List(_arr_make_unique_indeterminants(4,["a","b","c","d"]), a -> Indeterminate(Integers, a));
[ a, b, c, d ]
gap> List(_arr_make_unique_indeterminants(4,"t"), a -> Indeterminate(Integers, a));
[ t1, t2, t3, t4 ]
gap> List(_arr_make_unique_indeterminants(4,"t_"), a -> Indeterminate(Integers, a));
[ t_1, t_2, t_3, t_4 ]
gap> List(_arr_make_unique_indeterminants(4,"t_"), a -> Indeterminate(Integers, a));
[ t_1, t_2, t_3, t_4 ]
gap> indices := _arr_make_unique_indeterminants(4,"t_");
```

3 ▶ `_arr_move_to_general_position( A, f )` F

Apply change of basis operations on the forms defining the hyperplane arrangement  $A$  until the genericity conditions of  $f$  are satisfied. If  $A$  is already in general position then it will not be altered.

$f$  should be a function which takes in an arrangement and returns true or false indicating whether the given arrangement is in general position.

In a half-hearted attempt to alter as few variables as possible, this function begins by replacing rows of the identity transformation with  $[1, 2^i, 3^i, \dots, n^i]$  starting with the bottom rows. This function ensures that a solution will be found by eventually using the transformations:

$$\begin{pmatrix} 1 & 2^i & 3^i & \dots & n^i \\ 1 & 2^{i+1} & 3^{i+1} & \dots & n^{i+1} \\ \vdots & & & & \vdots \\ 1 & 2^{i+n-1} & 3^{i+n-1} & \dots & n^{i+n-1} \end{pmatrix}$$

4 ▶ `_arr_polynomial_by_ExtRep_and_indeterminates( rep, vars )` F

Constructs a polynomial in terms of variables  $vars$  regardless of the variables listed in the polynomial `ExtRep`,  $rep$ .  $vars$  may be either a list of integers (interpreted as the indeterminate numbers) or a list of indeterminates themselves. The variables used in  $rep$  will be sorted by indeterminate index and assigned sequentially to the variables in  $vars$ . Thus `Length(vars)` must be greater than or equal to the number of variables used in  $rep$ . The return value will be a polynomial even if  $rep$  represents a constant polynomial as long as  $vars$  is non-empty.

Note: If  $vars$  is an empty list, then the augmentation of  $rep$  is computed. That is, 1 is substituted for each of the variables in  $rep$  and a scalar is returned.

5 ▶ `_arr_polynomial_from_affine_polynomial_matrix( P, x )` F

Given a polynomial matrix  $P$  and a list of (sufficiently many) indeterminates  $x$ , returns the polynomial  $\prod P$  in the indeterminates  $x$ . Assume that the last column of  $P$  gives a constant term for the corresponding component.

6► `_arr_polynomial_from_central_polynomial_matrix( P, x )` F

Given a polynomial matrix  $P$  and a list of (sufficiently many) indeterminants  $x$ , returns the polynomial  $\prod P$  in the indeterminants  $x$ . Assume that each column of  $P$  is the coefficient of a variable.

7► `_arr_polynomial_matrix_from_polynomial_list( A )` F  
 ► `_arr_polynomial_matrix_from_polynomial_list( L )` F

Constructs a polynomial matrix from the list of polynomials  $L$  or from the `PolynomialList` of an arrangement  $A$ . If  $L$  has length  $m$  and  $\prod L$  is a polynomial in  $n$  variables, then the result of this function will be an  $m \times n + 1$  matrix. That is, we always append a column for the constant terms (even if none of the  $L$ 's have a constant term). You can remove the final column using the 6.1.2 command.

If an arrangement is provided, then the returned polynomial matrix will be of the proper dimensions.

8► `_generic_slice( A, r )` F

Given an arrangement  $A$  and a rank  $r$ , construct an affine arrangement  $B$  of dimension  $\dim(A) - 1$  with intersection lattice identical to that of  $A$  up to rank  $d$ . The coefficients of  $B$  will lie in the field of fractions of the coefficient ring of  $A$ .

## 7.10 Utilities.g

1► `_arr_cache( Arrangement, InvariantString, F )` F

Returns *InvariantString* of *Arrangement* if it exists, otherwise  $F$  is used. If  $F$  is a function then it is evaluated with the given arrangement as input and should return the requested invariant. If  $F$  is not a function then the arrangement invariant will be set to a structural copy of  $F$ .

If  $F$  is a function which requires two arguments (this does not include functions which may take a variable number of arguments) then  $F$  will be called with the *Arrangement* as its first argument and the *InvariantString* as its second argument.

2► `_arr_can_compute( Arrangement, TraitFunctions )` F  
 ► `_arr_can_compute( Arrangement, TraitFunctions : names := String )` F

Returns `true` if requesting the specified *TraitFunctions* succeeds. This means that if `_arr_can_compute( A, Foo )` returns `true`, then `Foo(A)` will immediately return a cached value. Calling this function will attempt to compute the requested *TraitFunctions* if necessary. Thus this function may not immediately return. The `names` value option should be a string list of the requested functions that will be printed in a diagnostic message if the “arrangement:arrangementinfo” level is set to a value greater than 10. Compare with `_arr_has_computed` (See 7.10.6).

3► `_arr_cmp_real( a, b )` F

Returns -1, 0, or 1 depending on whether the left argument is less than, equal to, or greater than the right argument on the real number line. Uses the `futil` package.

4► `_arr_containing_ring( L )` F  
 ► `_arr_containing_ring( e1, e2, ... )` F

If given a list of numbers, this function returns the smallest ring which contains all of them. The rings considered are: Integers, Rationals, `_arr_Reals`, Cyclotomics, and the finite fields.

5► `_arr_explain( Strings )` F

Prints strings (one on each line, followed by two newlines) if the `Explain` value option has been set. The strings should give details of the algorithm being used. Explanations should not depend on any `ArrangementInfo` at a level greater than 7.

6 ▶ `_arr_has_computed( Arrangement, StringTraits )` F

Returns `true` if the specified *StringTraits* have already been successfully computed. This means that if `_arr_has_computed( A, "Foo" )` returns `true`, then `Foo(A)` will immediately return a cached value. Calling this function does not cause any computations to actually take place. Only an inquiry is made. Compare with `_arr_can_compute` (See 7.10.2).

7 ▶ `_arr_LCM_INT( i, j )` F

Compute the Least Common Multiple of two integers.

WARNING: this function is a part of the **Arrangement** package and is not a core GAP function. In particular, this function is written in GAP code (not C) and thus will run slowly (compared to a compiled C version). Also note that using this function will make your code depend on the **Arrangement** package.

8 ▶ `_arr_positions( List, Function )` F

▶ `_arr_positions( List, Element )` F

Return indices for which *Function* is true, or that `List[i] = Element`.

9 ▶ `_arr_real_part( z )` F

▶ `_arr_imaginary_part( z )` F

Return the real (resp. imaginary) part of a cyclotomic number.

10 ▶ `_arr_sorted_tuple_of_reals( ListOfTuples, ListOfIndices, D1, D2, ... )` F

Returns *ListOfTuples* sorted by their real magnitude. If a *ListOfIndices* is provided then it should contain the columns to consider when sorting (note: the order of the elements in *ListOfIndices* will affect the sorted list). Finally, if additional lists *D1*, *D2*, ... are provided then the decimal precision will be large enough to guarantee that each *D* list would contain distinct elements. By default, a *D* list is created for each column being sorted and *D<sub>i</sub>* is the result of calling **Unique** on the corresponding columns of the tuples.

The return value is a new list where each element is exactly one (not a copy) of the input tuples.

# Bibliography

# Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., “PermutationCharacter” comes before “permutation group”.

theindex  
\_FoxDerivative', 14  
\_FoxHessian', 14  
\_FoxJacobian', 14  
\_arr\_LCM\_INT', 29  
\_arr\_Use', 21  
\_arr\_arrangement\_from\_hash', 31  
\_arr\_cache', 28  
\_arr\_can\_compute', 28  
\_arr\_canonical\_generators\_of\_algebra', 14  
\_arr\_canonicalize\_polynomial', 25  
\_arr\_cmp\_real', 28  
\_arr\_combine\_like\_terms', 15  
\_arr\_compute\_fox\_derivative', 15  
\_arr\_compute\_fox\_hessian', 15  
\_arr\_compute\_fox\_jacobian', 15  
\_arr\_container\_of\_fox\_thingun', 15  
\_arr\_containing\_ring', 28  
\_arr\_delete\_line', 21  
\_arr\_dirs V', 21  
\_arr\_explain', 28  
\_arr\_face\_lattice', 24  
\_arr\_face\_lattice\_merge', 24  
\_arr\_face\_vector\_not\_too\_far', 24  
\_arr\_files V', 21  
\_arr\_find\_four\_in\_general\_position', 25  
\_arr\_find\_transformation\_to\_x\_n', 26  
\_arr\_fox\_apply\_maps', 15  
\_arr\_has\_computed', 28  
\_arr\_hyperplane\_arrangement\_from\_hash', 21  
\_arr\_imaginary\_part', 29  
\_arr\_lattice\_boundary\_operator', 23  
\_arr\_lattice\_orthogonal\_vector', 24  
\_arr\_make\_unique\_indeterminants', 27  
\_arr\_map\_matrix', 15  
\_arr\_move\_to\_general\_position', 27  
\_arr\_polynomial\_by\_ExtRep\_and\_indeterminates', 27  
\_arr\_polynomial\_from\_affine\_polynomial\_matrix', 27  
\_arr\_polynomial\_from\_central\_polynomial\_matrix', 27  
\_arr\_polynomial\_matrix\_from\_polynomial\_list', 27  
\_arr\_positions', 29  
\_arr\_prod\_sum', 15  
\_arr\_real\_part', 29  
\_arr\_realization\_defining\_vectors', 25  
\_arr\_realization\_do\_dependent\_sets', 25  
\_arr\_realization\_expand\_determined\_vectors', 25  
\_arr\_realization\_introduce\_double\_variables', 26  
\_arr\_realization\_introduce\_single\_variable', 26  
\_arr\_register\_constructor', 21  
\_arr\_register\_hyparr\_constructor', 22  
\_arr\_sign\_vector', 25  
\_arr\_sorted\_tuple\_of\_reals', 29  
\_arr\_words\_to\_list', 16  
\_brd\_braid\_monodromy\_from\_shelled\_lattice', 22  
\_fox\_augmentation', 13  
\_fox\_group2gens', 14  
\_fox\_group2group', 14  
\_fox\_group2ring', 16  
\_fox\_ints2gens', 14  
\_fox\_ints2group', 14  
\_fox\_ints2ring', 14  
\_fox\_ring2gens', 16  
\_fox\_ring2group', 16  
\_fox\_ring2ints', 16  
\_generic\_slice', 28  
\_gph\_add\_graph\_edges', 22  
\_gph\_add\_signed\_graph\_edges', 22  
\_gph\_complete\_subgraphs', 22  
\_gph\_distance', 22  
\_gph\_distance\_matrix', 22  
\_gph\_graph\_adjacency\_matrix', 23  
\_gph\_is\_connected', 23  
\_gph\_nr\_complete\_subgraphs', 23  
\_gph\_signed\_graph\_adjacency\_matrix', 23  
\_gph\_width', 23

- Arrangement', 7
- ArrangementInfo V', 21
- A r r a n g e m e n t p a c k a g e, 3
- Augmentation', 13
- BaseField', 8
- Cardinality', 8
- Ceil', 17
- CentralHyperplaneArrangement', 10
- Chop', 17
- ChopMat', 17
- CoefficientsRing', 8
- ConstantTerm', 19
- CurveDegrees', 8
- DeepCopyListOfNumbers', 17
- DeepSort', 17
- DeepSortedList', 17
- Degree', 19
- Deletion', 7
- Dimension', 8
- Display', 9
- Dump', 9
- EvenPositions', 17
- ExtRepOfObj', 9
- FirstPos', 17
- FisherYatesShuffle', 17
- FisherYatesShuffleDestructive', 17
- FoxDerivative', 13
- FoxHessian', 13
- FoxJacobian', 13
- FpGroupDirectProductFpGroups', 18
- FreeFoxDerivative', 13
- FreeFoxHessian', 13
- FreeFoxJacobian', 13
- HyperplaneArrangement', 10
- IndeterminateNumbers', 8, 20
- Indeterminates', 8, 20
- IsAffineArrangement', 12
- IsArrangement', 5
- IsCentralArrangement', 12
- IsEssentialArrangement', 8
- IsHomogeneous', 20
- IsHyperplaneArrangement', 5
- IsMonomial', 20
- IsSupersolvable', 12
- KroneckerDelta', 18
- Last', 18
- LastPos', 18
- LatticeDiagram', 18
- MonomialDenominatorLCM', 20
- MonomialLCM', 20
- NrCurves', 8
- NrIndeterminates', 20
- OddPositions', 17
- Polynomial', 8
- PolynomialList', 9
- QuotientHomomorphism', 13
- Reduce', 18
- ReduceTensorRank2', 19
- Shift', 19
- SortArrangementLattice', 19
- Splice', 19
- ViewObj', 9
- WittFormula', 19
- Zip', 19