GAP Share Package AREP
version 1.0

Sebastian Egner
Markus Püschel
Institut für Algorithmen und Kognitive Systeme
Universitt Karlsruhe

3. Aug. 1998

Contents

Chapter 1

AREP

The share package AREP provides an infrastructure and high level functions to do efficient calculations in constructive representation theory. By the term "constructive" we mean that group representations are constructed and manipulated up to equality – not only up to equivalence as it is done by using characters. Hence you can think of it as working with matrix representations, but in a very efficient way using the special structure of the matrices occuring in representation theory of finite groups. The package is named after its most important class **ARep** (see 1.66) (**A**bstract **Rep**resentations)[1] implementing this idea.

A striking application of constructive representation theory is the decomposition of matrices representing discrete signal transforms into a product of highly structured sparse matrices (realized in 1.147). This decomposition can be viewed as a fast algorithm for the signal transform. Another application is the construction of fast Fourier transforms for solvable groups (realized in 1.123). The package has evolved out of this area of application into a more general tool.

The package AREP consists of the following parts:

- **Monomial Matrices:** A monomial matrix is matrix containing exactly one non-zero entry in every row and column. Hence storing and computing with monomial matrices can be done efficiently. This is realized in the class **Mon**, Sections 1.2 – 1.21.
- **Structured Matrices:** The class **AMat**, Sections 1.22 – 1.65, is created to represent and calculate with structured matrices, like e.g. $2 \cdot (A \oplus B)^C \otimes D \cdot E^2$, where A, B, C, D, E are matrices of compatible size and characteristic.
- **Group Representations:** The class **ARep**, Sections 1.66 – 1.123, is created to represent and manipulate structured representations up to equality, like e.g. $(\phi \uparrow_T G)^M \otimes \psi$. Special care is taken of monomial representations.

[1] A note on the name: We have chosen "abstract" because we manipulate expressions for representations, not constants. However, "concrete" would also be right because the representations are given with respect to a fixed basis of the underlying vector space. The name ARep is thus, for historical reasons, somewhat misleading.

- **Symmetry of Matrices:** In Sections 1.124 – 1.127 functions are provided to compute certain kinds of symmetry of a given matrix. Symmetry allows to describe structure contained in a matrix.

- **Discrete Signal Transforms:** Sections 1.128 – 1.146 describe functions to construct many well-known discrete signal transforms.

- **Matrix Decomposition:** Sections 1.147 – 1.149 describe functions to decompose a discrete signal transform into a product of highly structured sparse matrices.

- **Tools for Complex Numbers, Matrices and Permutations:** Sections 1.151 – 1.173 describe useful tools for the computation with complex numbers, matrices and permutations.

All functions described are written entirely in the GAP language. The functions for the computation of the symmetry of a matrix (see 1.124) may use the external C program `desauto` written by J. Leon and contained in the share package GUAVA. However, the use of this program is optional and will only influence the speed and not the executability of the functions.

The package AREP was created in the framework of our theses where the background of constructive representation theory (see [Püs98]) and searching for symmetry of matrices (see [Egn97]) can be found.

1.1 Loading AREP

After having started GAP the AREP package needs to be loaded. This is done by typing:

```
gap> RequirePackage("arep");

  ___   ___   ___  ___
 |   | |   | |    |   |    Version 1.0, 16 Mar 1998
 |___| |___| |___ |___|
 |   | |  \  |    |        by Sebastian Egner
 |   | |   \ |___ |           Markus Pueschel

Abstract REPresentations
```

If AREP isn't already in memory it is loaded and its banner is displayed. If you are a frequent user of AREP you might consider putting this line into your `.gaprc` file.

1.2 Mons

The class **Mon** is created to represent and calculate efficiently with monomial matrices. A monomial matrix is a matrix which contains exactly one non-zero entry in every row and every column. Hence monomial matrices are always invertible and a generalization of permutation matrices. The elements of the class **Mon** are called "mons". A mon m is a record with at least the following fields.

`isMon`	:=	`true`
`isGroupElement`	:=	`true`
`domain`	:=	`GroupElements`
`operations`	:=	`MonOps`
`char`	:	characteristic of the base field
`perm`	:	a permutation
`diag`	:	a list of non-zero field elements

The MonOps class is derived from the GroupElementOps class, so that groups of mons can be constructed. The monomial matrix represented by a mon m is given by

$$[\delta_{i^p j} \mid i, j \in \{1, \ldots, \texttt{Length}(m.\texttt{diag})\}] \cdot \texttt{DiagonalMat}(m.\texttt{diag}),$$

where $p = m$`.perm` and $\delta_{k\ell}$ denotes the Kronecker symbol ($\delta_{k\ell} = 1$ if $k = \ell$ and $= 0$ else). Mons are created using the function `Mon`. The following sections describe functions used for the calculation with mons.

Some remarks on the design of **Mon**: Mons cannot be mixed with GAP-matrices (which are just lists of lists of field elements); use `MonMat` (1.11) and `MatMon` (1.10) to convert explicitly. Mons are lightweighted, e.g. only the characteristic of the base field is stored. Mons are group elements but there are no efficient functions implemented to compute with mon groups. You should think of mons as being a similar thing as integers or permutations: They are just fundamental objects to work with.

The functions concerning mons are implemented in the file `"arep/lib/mon.g"`.

1.3 Comparison of Mons

m_1 `=` m_2
m_1 `<>` m_2

The equality operator `=` evaluates to `true` if the mons m_1 and m_2 are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the mons m_1 and m_2 are not equal and to `false` otherwise.

Two mons are equal iff they define the same monomial matrix. Note that the monomial matrix being represented has a certain size. The sizes must agree, too.

m_1 `<` m_2
m_1 `<=` m_2
m_1 `>=` m_2
m_1 `>` m_2

The operators `<`, `<=`, `>=`, and `>` evaluate to `true` if the mon m_1 is strictly less than, less than or equal to, greater than or equal to, and strictly greater than the mon m_2.

The ordering of mons m is defined via the ordering of the pairs `[`m`.perm, `m`.diag]`.

1.4 Basic Operations for Mons

The MonOps class is derived from the GroupElementsOps class.

m_1 * m_2
m_1 / m_2

The operators * and / evaluate to the product and quotient of the two mons m_1 and m_2. The product is defined via the product of the corresponding (monomial) matrices. Of course the mons must be of equal size and characteristic otherwise an error is signaled.

m_1 ^ m_2

The operator ^ evaluates to the conjugate $m_2{}^{-1} * m_1 * m_2$ of m_1 under m_2 for two mons m_1 and m_2. The mons must be of equal size and characteristic otherwise an error is signaled.

m ^ i

The powering operator ^ returns the i-th power of the mon m and the integer i.

`Comm(` m_1, m_2 `)`

`Comm` returns the commutator $m_1{}^{-1} * m_2{}^{-1} * m_1 * m_2$ of two mons m_1 and m_2. The operands must be of equal size and characteristic otherwise an error is signaled.

`LeftQuotient(` m_1, m_2 `)`

`LeftQuotient` returns the left quotient $m_1{}^{-1} * m_2$ of two mons m_1 and m_2. The operands must be of equal size and characteristic otherwise an error is signaled.

1.5 Mon

`Mon(` p, D `)`

Let p be a permutation and D a list of field elements $\neq 0$ of the same characteristic. `Mon` returns a mon representing the monomial matrix given by $[\delta_{i^p j} \mid i, j \in \{1, \ldots, \texttt{Length}(D)\}] \cdot$ $\texttt{DiagonalMat}(D)$, where $\delta_{k\ell}$ denotes the Kronecker symbol. The function will signal an error if the length of D is less than the largest moved point of p.

```
gap> Mon( (1,2), [1, 2, 3] );
Mon(
  (1,2),
  [ 1, 2, 3 ]
)
gap> Mon( (1,3,4), [Z(3)^0, Z(3)^2, Z(3), Z(9)]);
Mon(
  (1,3,4),
  [ Z(3)^0, Z(3)^0, Z(3), Z(3^2) ]
)
```

`Mon(` D, p `)`

`Mon` returns a mon representing the monomial matrix given by $\texttt{DiagonalMat}(D)\cdot[\delta_{i^p j} \mid i, j \in \{1,\ldots,\texttt{Length}(D)\}]$, where $\delta_{k\ell}$ denotes the Kronecker symbol. Note that in the output the diagonal is commuted to the right side, but it still represents the same monomial matrix.

```
gap> Mon( [1,2,3], (1,2) );
Mon(
  (1,2),
  [ 2, 1, 3 ]
)
gap> Mon( [Z(3)^0, Z(3)^2, Z(3), Z(9)], (1,3,4) );
Mon(
  (1,3,4),
  [ Z(3^2), Z(3)^0, Z(3)^0, Z(3) ]
)
```

`Mon(` *D* `)`

`Mon` returns a mon representing the (monomial) diagonal matrix given by the list D.

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

`Mon(` *p* `,` *d* `)`
`Mon(` *p* `,` *d* `,` *char* `)`
`Mon(` *p* `,` *d* `,` *field* `)`

Let p be a permutation and d a positive integer. `Mon` returns a mon representing the $(d \times d)$ permutation matrix corresponding to p using the convention $[\delta_{i^p j} \mid i, j \in \{1,\ldots,d\}]$, where $\delta_{k\ell}$ denotes the Kronecker symbol. As optional parameter a characteristic *char* or a *field* can be supplied. The default characteristic is zero. The function will signal an error if the degree d is less than the largest moved point of p.

```
gap> Mon( (1,2), 3 );
Mon( (1,2), 3 )
gap> Mon( (1,2,3), 3, 5 );
Mon( (1,2,3), 3, GF(5) )
```

`Mon(` *m* `)`

Let m a mon. `Mon` returns m.

```
gap> Mon( Mon( (1,2), [1, 2, 3] ) );
Mon(
  (1,2),
  [ 1, 2, 3 ]
)
```

1.6 IsMon

`IsMon(` *obj* `)`

`IsMon` returns `true` if *obj*, which may be an object of arbitrary type, is a mon, and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsMon( Mon( (1,2), [1, 2, 3] ) );
true
gap> IsMon( (1,2) );
false
```

1.7 IsPermMon

`IsPermMon(` m `)`

`IsPermMon` returns `true` if the mon m represents a permutation matrix and `false` otherwise.

```
gap> IsPermMon( Mon( (1,2), [1, 2, 3] ) );
false
gap> IsPermMon( Mon( (1,2), 2) );
true
```

1.8 IsDiagMon

`IsDiagMon(` m `)`

`IsDiagMon` returns `true` if the mon m represents a diagonal matrix and `false` otherwise.

```
gap> IsDiagMon( Mon( (1,2), 2) );
false
gap> IsDiagMon( Mon( [1, 2, 3, 4] ) );
true
```

1.9 PermMon

`PermMon(` m `)`

`PermMon` converts the mon m to a permutation if possible and returns `false` otherwise.

```
gap> PermMon( Mon( (1,2), 5) );
(1,2)
gap> PermMon( Mon( [1,2] ) );
false
```

1.10 MatMon

`MatMon(` m `)`

`MatMon` converts the mon m to a matrix (i.e. a list of lists of field elements).

```
gap> MatMon( Mon( (1,2), [1, 2, 3] ) );
[ [ 0, 2, 0 ], [ 1, 0, 0 ], [ 0, 0, 3 ] ]
gap> MatMon( Mon( (1,2), 3) );
[ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ]
```

1.11 MonMat

`MonMat(` M `)`

`MonMat` converts the matrix M to a mon if possible and returns `false` otherwise.

```
gap> MonMat( [ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ] );
Mon( (1,2), 3 )
gap> MonMat( [ [ 0, 1, 0 ], [ E(3), 0, 0 ], [ 0, 0, 4 ] ] );
Mon(
  (1,2),
  [ E(3), 1, 4 ]
)
```

1.12 DegreeMon

`DegreeMon(` m `)`

`DegreeMon` returns the degree of the mon m. The degree is the size of the represented matrix.

```
gap> DegreeMon( Mon( (1,2), [1, 2, 3] ) );
3
```

1.13 CharacteristicMon

`CharacteristicMon(` m `)`

`CharacteristicMon` returns the characteristic of the field from which the components of the mon m are.

```
gap> CharacteristicMon( Mon( [1,2] ) );
0
gap> CharacteristicMon( Mon( (1,2), 4, 5) );
5
```

1.14 OrderMon

`OrderMon(` m `)`

`OrderMon` returns the order of the monomial matrix represented by the mon m. The order of m is the least positive integer r such that m^r is the identity. Note that the order might be infinite.

```
gap> OrderMon( Mon( [1,2] ) );
"infinity"
gap> OrderMon( Mon( (1,2), [1, E(3), E(3)^2] ) );
6
```

1.15 TransposedMon

`TransposedMon(` m `)`

`TransposedMon` returns a mon representing the transposed monomial matrix of the mon m.

```
gap> TransposedMon( Mon( [1,2] ) );
Mon( [ 1, 2 ] )
gap> TransposedMon( Mon( (1,2,3), 4 ) );
Mon( (1,3,2), 4 )
```

1.16 DeterminantMon

`DeterminantMon(` m `)`

`DeterminantMon` returns the determinant of the monomial matrix represented by the mon m.

```
gap> DeterminantMon( Mon( (1,2), [1, E(3), E(3)^2] ) );
-1
gap> DeterminantMon( Mon( [1,2] ) );
2
```

1.17 TraceMon

`TraceMon(` m `)`

`TraceMon` returns the trace of the monomial matrix represented by the mon m.

```
gap> TraceMon( Mon( (1,2), 4, 5) );
Z(5)
gap> TraceMon( Mon( [1,2] ) );
3
```

1.18 GaloisMon

`GaloisMon(` m, *aut* `)`
`GaloisMon(` m, k `)`

`GaloisMon` returns a mon which is a galois conjugate of the mon m. This means that each component of the represented matrix is mapped with an automorphism of the underlying field. The conjugating automorphism may either be a field automorphism *aut* or an integer k specifying the automorphism `x -> GaloisCyc(x,` k`)` in the case characteristic $= 0$ or `x -> x^(FrobeniusAut^`k`)` in the case characteristic $= p$ prime.

```
gap> GaloisMon( Mon( (1,2), [1, E(3), E(3)^2] ), -1 );
Mon(
  (1,2),
  [ 1, E(3)^2, E(3) ]
)
gap> aut := FrobeniusAutomorphism( GF(4) );
FrobeniusAutomorphism( GF(2^2) )
gap> GaloisMon( Mon( [ Z(2)^0, Z(2^2), Z(2^2)^2 ] ), aut );
Mon( [ Z(2)^0, Z(2^2)^2, Z(2^2) ] )
```

1.19 DirectSumMon

`DirectSumMon(` m_1, `...`, m_k `)`

`DirectSumMon` returns the direct sum of the mons m_1, ..., m_k. The direct sum of mons is defined via the direct sum of the represented matrices. Note that the mons must have the same characteristic.

```
gap> m1 := Mon( (1,2), [1, E(3), E(3)^2] );
Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
)
gap> m2 := Mon( (1,2), 3);
Mon( (1,2), 3 )
gap> DirectSumMon( m1, m2 );
Mon(
  (1,2)(4,5),
  [ 1, E(3), E(3)^2, 1, 1, 1 ]
)
```

`DirectSumMon(` *list* `)`

`DirectSumMon` returns a mon representing the direct sum of the mons in *list*.

```
gap> m1 := Mon( (1,2), [1, E(3), E(3)^2] );
Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
)
gap> m2 := Mon( (1,2), 3);
Mon( (1,2), 3 )
gap> DirectSumMon( [m1, m2] );
Mon(
  (1,2)(4,5),
  [ 1, E(3), E(3)^2, 1, 1, 1 ]
)
```

1.20 TensorProductMon

`TensorProductMon(` m_1`, ...,` m_k `)`

`TensorProductMon` returns the tensor product of the mons m_1, ..., m_k. The tensor product of mons is defined via the tensor product (or Kronecker product) of the represented matrices. Note that the mons must have the same characteristic.

```
gap> m1 := Mon( (1,2), [1, E(3), E(3)^2] );
Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
)
gap> m2 := Mon( (1,2), 3);
Mon( (1,2), 3 )
gap> TensorProductMon( m1, m2 );
Mon(
  (1,5)(2,4)(3,6)(7,8),
  [ 1, 1, 1, E(3), E(3), E(3), E(3)^2, E(3)^2, E(3)^2 ]
)
```

`TensorProductMon(` *list* `)`

`TensorProductMon` returns a mon representing the tensor product of the mons in *list*.

```
gap> m1 := Mon( (1,2), [1, E(3), E(3)^2] );
Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
)
gap> m2 := Mon( (1,2), 3);
Mon( (1,2), 3 )
gap> TensorProductMon( [m1, m2] );
Mon(
  (1,5)(2,4)(3,6)(7,8),
  [ 1, 1, 1, E(3), E(3), E(3), E(3)^2, E(3)^2, E(3)^2 ]
)
```

1.21 CharPolyCyclesMon

`CharPolyCyclesMon(` *m* `)`

`CharPolyCyclesMon` returns the sorted list of the characteristic polynomials of the cycles of the mon m. All polynomials are written in a common polynomial ring. Applying `Product` to the result yields the characteristic polynomial of m.

```
gap> CharPolyCyclesMon( Mon( (1,2), 3 ) );
[ X(Rationals) - 1, X(Rationals)^2 - 1 ]
gap> CharPolyCyclesMon( Mon( (1,2), [1, E(3), E(3)^2] ) );
[ X(CF(3)) + (-E(3)^2), X(CF(3))^2 + (-E(3)) ]
```

1.22 AMats

The class **AMat** (**A**bstract **Mat**rices) is created to represent and calculate efficiently with structured matrices like e.g. $2 \cdot (A \oplus B)^C \otimes D \cdot E^2$, where A, B, C, D, E are matrices of compatible size/characteristic and $\oplus, \otimes$ denote the direct sum and tensor product (Kronecker product) resp. of matrices. The elements of the class **AMat** are called "amats" and implement a recursive datastructure to form expressions like the one above. Basic constructors for amats allow to create permutation matrices (see `AMatPerm`, 1.23), monomial matrices (see `AMatMon`, 1.24) and general matrices (see `AMatMat`, 1.25) in an efficient way (e.g. a permutation matrix is defined by a permutation, the degree and the characteristic). Higher constructors allow to construct direct sums (see `DirectSumAMat`, 1.40), tensor products (see `TensorProductAMat`, 1.41) etc. from given amats. Note that while building up a highly structured amat from other amats no computation is done beside checks for compatibility. To obtain the matrix represented by an amat the appropiate function has to be applied (e.g. `MatAMat`, 1.50).

Some remarks on the design of **AMat**: The class **AMat** is what is called a term algebra for expressions representing highly structured matrices over certain base fields. Amats are not necessarily square but can also be rectangular. Hence, if an amat must be invertible (e.g. when it shall conjugate another amat) this has to be proven by computation. To avoid many of these calculations the result (the inverse) is stored in the object and many functions accept a "hint". E.g. by supplying the hint "invertible" in the example above the explicit check for invertibility is suppressed. Using and passing correct hints is essential for efficient computation. A common problem in the design of non-trivial term algebras is the simplification strategy: Aggressive or conservative simplification? Our approach here is extremely conservative. This means even trivial subexpressions like $1 * A$ are not automatically simplified. This allows the user to write functions that return their result always in a fixed structure, e.g. the result is always a conjugated direct sum of tensor products even though the conjugation might be trivial. Finally, note that amats and normal matrices (i.e. lists of lists of field elements) do not mix – you have to convert explicitly with `AMatMat`, `MatAMat` etc. This greatly simplifies the amat module.

We define an amat recursively in Backus-Naur-Form as the disjoint union of the following cases.

amat ::=			
;	atomic cases		
	perm	;	"perm" (invertible)
|	*mon*	;	"mon" (invertible)
|	*mat*	;	"mat"
;	composed cases		
|	$scalar \cdot amat$	;	"scalarMultiple"
|	$amat \cdot \ldots \cdot amat$	;	"product"
|	$amat$ ^ int	;	"power"
|	$amat$ ^ $amat$	;	"conjugate"
|	$amat \oplus \ldots \oplus amat$	;	"directSum"
|	$amat \otimes \ldots \otimes amat$	;	"tensorProduct"
|	GaloisConjugate(*amat*, *aut*)	;	"galoisConjugate".

An amat A is a record with at least the following fields:

`isAMat`	:=	`true`
`operations`	:=	`AMatOps`
`type`	:	a string identifying the type of A
`dimensions`	:	size of the matrix represented (= [rows, columns])
`char`	:	characteristic of the base field

The cases as stated above are distinguished by the field `.type` of an amat. Depending on the type additional fields are mandatory as follows:

`type = "perm"`:
`element` defining permutation

`type = "mon"`:
`element` defining mon-object (see 1.2)

`type = "mat"`:
`element` defining matrix (list of lists of field elements)

`type = "scalarMultiple"`:
`element` the AMat multiplied
`scalar` the scalar

`type = "product"`:
`factors` list of AMats of compatible dimensions and the same characteristic

`type = "power"`:
`element` the square AMat to be raised to exponent
`exponent` the exponent (an integer)

`type = "conjugate"`:
`element` the square AMat to be conjugated
`conjugation` the conjugating invertible AMat

type = "directSum":

`summands`	List of AMats of the same characteristic

type = "tensorProduct":

`factors`	List of AMats of the same characteristic

type = "galoisConjugate":

`element`	the AMat to be Galois conjugated
`galoisAut`	the Galois automorphism

Note that there is an important difference between the *type of an amat* and the *type of the matrix being represented by the amat*: An amat can be of type "mat" but the matrix is in fact a permutation matrix. This distinction is refelcted in the naming of the functions: "XAMat" refers to the type of the amat, "XMat" to the type of the matrix being represented,

Here a short overview of the functions concerning amats. sections 1.23 – 1.43 are concerned with the construction of amats, sections 1.44 – 1.53 with the convertability and conversion of amats to permutations, mons and matrices, sections 1.54 – 1.65 contain functions for amats, e.g. computation of the determinant or simplification of amats.

The functions concerning amats are implemented in the file `"arep/lib/amat.g"`.

1.23 AMatPerm

`AMatPerm(` *p*, *d* `)`
`AMatPerm(` *p*, *d*, *char* `)`
`AMatPerm(` *p*, *d*, *field* `)`

`AMatPerm` returns an amat of type `"perm"` representing the $(d \times d)$ permutation matrix $[\delta_{i^p j} \mid i,j \in \{1,\ldots,d\}]$ corresponding to the permutation p. As optional parameter a characteristic *char* or a *field* can be supplied. The default characteristic is zero. The function will signal an error if the degree d is less than the largest moved point of p.

```
gap> AMatPerm( (1,2), 5 );
AMatPerm((1,2), 5)
gap> AMatPerm( (1,2,3), 5 , 3);
AMatPerm((1,2,3), 5, GF(3))
gap> A := AMatPerm( (1,2,3), 5 , Rationals);
AMatPerm((1,2,3), 5)
gap> A.type;
"perm"
```

1.24 AMatMon

`AMatMon(` *m* `)`

`AMatMon` returns an amat of type `"mon"` representing the monomial matrix given by the mon m. For the explanation of mons please refer to 1.2.

```
gap> AMatMon( Mon( (1,2), [1, E(3), E(3)^2] ) );
AMatMon( Mon(
```

```
    (1,2),
    [ 1, E(3), E(3)^2 ]
  ) )
gap> A := AMatMon( Mon( (1,2), 3) );
AMatMon( Mon( (1,2), 3 ) )
gap> A.type;
"mon"
```

1.25 AMatMat

`AMatMat(` *M* `)`
`AMatMat(` *M*, *hint* `)`

`AMatMat` returns an amat of type `"mat"` representing the matrix M. If the optional *hint* `"invertible"` is supplied then the field `.isInvertible` of the amat is set to `true` (without checking) indicating that the matrix represented is invertible.

```
gap> AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> A := AMatMat( [ [1,2], [3,4] ] , "invertible");
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ],
  "invertible"
)
gap> A.isInvertible;
true
```

1.26 IsAMat

`IsAMat(` *obj* `)`

`IsAMat` returns `true` if *obj*, which may be an object of arbitrary type, is an amat, and `false` otherwise.

```
gap> IsAMat( AMatPerm( (1,2,3), 3 ) );
true
gap> IsAMat( 1/2 );
false
```

1.27 IdentityPermAMat

`IdentityPermAMat(` *n* `)`
`IdentityPermAMat(` *n*, *char* `)`
`IdentityPermAMat(` *n*, *field* `)`

`IdentityPermAMat` returns an amat of type `"perm"` representing the $(n \times n)$ identity matrix. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero. Note that the same result can be obtained by using `AMatPerm`.

```
gap> IdentityPermAMat( 3 );
IdentityPermAMat(3)
gap> AMatPerm( ( ), 3);
IdentityPermAMat(3)
gap> IdentityPermAMat( 3 , GF(3) );
IdentityPermAMat(3, GF(3))
```

1.28 IdentityMonAMat

`IdentityMonAMat(` *n* `)`
`IdentityMonAMat(` *n*`,` *char* `)`
`IdentityMonAMat(` *n*`,` *field* `)`

`IdentityMonAMat` returns an amat of type `"mon"` representing the $(n \times n)$ identity matrix. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero. Note that the same result can be obtained by using `AMatMon`.

```
gap> IdentityMonAMat( 3 );
IdentityMonAMat(3)
gap> AMatMon( Mon( ( ), 3 ) );
IdentityMonAMat(3)
gap> IdentityMonAMat( 3, 3 );
IdentityMonAMat(3, GF(3))
```

1.29 IdentityMatAMat

`IdentityMatAMat(` *n* `)`
`IdentityMatAMat(` *n*`,` *char* `)`
`IdentityMatAMat(` *n*`,` *field* `)`

`IdentityMatAMat` returns an amat of type `"mat"` representing the $(n \times n)$ identity matrix. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero. Note that the same result can be obtained by using `AMatMat`.

```
gap> IdentityMatAMat( 3 );
IdentityMatAMat(3)
gap> AMatMat( [ [1, 0, 0], [0, 1, 0], [0, 0, 1] ]);
IdentityMatAMat(3)
gap> IdentityMatAMat( 3, GF(3) );
IdentityMatAMat(3, GF(3))
```

`IdentityMatAMat(` *dim* `)`
`IdentityMatAMat(` *dim*`,` *char* `)`
`IdentityMatAMat(` *dim*`,` *field* `)`

Let *dim* be a pair of positive integers. `IdentityMatAMat` returns an amat of type `"mat"` representing the rectangular identity matrix with *dim*`[1]` rows and *dim*`[2]` columns. A rectangular identity matrix has the entry 1 at the position (i, j) if $i = j$ and 0 else. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> IdentityMatAMat( [2, 3] );
IdentityMatAMat([ 2, 3 ])
gap> IdentityMatAMat( [2, 3], 3 );
IdentityMatAMat([ 2, 3 ], GF(3))
```

1.30 IdentityAMat

`IdentityAMat(` *dim* `)`
`IdentityAMat(` *dim*`,` *char* `)`
`IdentityAMat(` *dim*`,` *field* `)`

Let *dim* be a pair of positive integers. `IdentityAMat` returns an amat of type `"perm"` if *dim*`[1] =` *dim*`[2]` and an amat of type `"mat"` else, representing the identity matrix with *dim*`[1]` rows and *dim*`[2]` columns. A rectangular identity matrix has the entry 1 at the position (i, j) if $i = j$ and 0 else. Use this function if you do not know whether the matrix is square and you do not care about the type. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> IdentityAMat( [2, 2] );
IdentityPermAMat(2)
gap> IdentityAMat( [2, 3] );
IdentityMatAMat([ 2, 3 ])
```

1.31 AllOneAMat

`AllOneAMat(` *n* `)`
`AllOneAMat(` *n*`,` *char* `)`
`AllOneAMat(` *n*`,` *field* `)`

`AllOneAMat` returns an amat of type `"mat"` representing the $(n \times n)$ all-one matrix. An all-one matrix has the entry 1 at each position. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the all-one matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> AllOneAMat( 3 );
AllOneAMat(3)
gap> AllOneAMat( 3, 3);
AllOneAMat(3, GF(3))
```

`AllOneAMat(` *dim* `)`
`AllOneAMat(` *dim*`,` *char* `)`
`AllOneAMat(` *dim*`,` *field* `)`

Let *dim* a pair of positive integers. `AllOneAMat` returns an amat of type `"mat"` representing the rectangular all-one matrix with *dim*`[1]` rows and *dim*`[2]` columns. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the all-one matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> AllOneAMat( [3, 2] );
AllOneAMat([ 3, 2 ])
gap> AllOneAMat( [3, 2], GF(5) );
AllOneAMat([ 3, 2 ], GF(5))
```

1.32 NullAMat

`NullAMat(` *n* `)`
`NullAMat(` *n*, *char* `)`
`NullAMat(` *n*, *field* `)`

`NullAMat` returns an amat of type `"mat"` representing the $(n \times n)$ all-zero matrix. An all-zero matrix has the entry 0 at each position. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the all-zero matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> NullAMat( 3 );
NullAMat(3)
gap> NullAMat( 3, 3);
NullAMat(3, GF(3))
```

`NullAMat(` *dim* `)`
`NullAMat(` *dim*, *char* `)`
`NullAMat(` *dim*, *field* `)`

Let *dim* a pair of positive integers. `NullAMat` returns an amat of type `"mat"` representing the rectangular all-zero matrix with *dim*`[1]` rows and *dim*`[2]` columns. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the all-zero matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> NullAMat( [3, 2] );
NullAMat([ 3, 2 ])
gap> NullAMat( [3, 2], GF(5) );
NullAMat([ 3, 2 ], GF(5))
```

1.33 DiagonalAMat

`DiagonalAMat(` *list* `)`

Let *list* contain field elements of the same characteristic. `DiagonalAMat` returns an amat representing the diagonal matrix with diagonal entries in *list*. If all elements in *list* are $\neq 0$ the returned amat is of type `"mon"`, else of type `"directSum"` (see 1.22).

```
gap> DiagonalAMat( [2, 3] );
DiagonalAMat([ 2, 3 ])
gap> DiagonalAMat( [0, 2, 3] );
DirectSumAMat(
  NullAMat(1),
  AMatMat(
    [ [ 2 ] ]
  ),
  AMatMat(
    [ [ 3 ] ]
  )
)
```

1.34 DFTAMat

`DFTAMat(` *n* `)`
`DFTAMat(` *n*`,` *char* `)`
`DFTAMat(` *n*`,` *field* `)`

`DFTAMat` returns a special amat of type `"mat"` representing the matrix

$$\mathrm{DFT}_n = [\omega_n^{i \cdot j} \mid i, j \in \{0, \ldots, n-1\}],$$

with ω_n being a certain primitive n-th root of unity. DFT_n represents the Discrete Fourier Transform on n points (see 1.129). As optional parameter a characteristic *char* or a *field* can be supplied to obtain the DFT of arbitrary characteristic. The default characteristic is zero. Note that for characteristic p prime the DFT_n exists iff $\gcd(p, n) = 1$. For a given finite *field* the DFT_n exists iff $n \mid$ `Size`(F). If these conditions are violated an error is signaled. The choice of ω_n is `E(`n`)` if $char = 0$ and `Z(`q`)^((`q-1`)/`n`)` for $char = p$, q an appropiate p-power.

```
gap> DFTAMat(3);
DFTAMat(3)
gap> DFTAMat(3, 7);
DFTAMat(3, 7)
```

1.35 SORAMat

`SORAMat(` *n* `)`
`SORAMat(` *n*`,` *char* `)`
`SORAMat(` *n*`,` *field* `)`

`SORAMat` returns a special amat of type `"mat"` representing the matrix

$$\mathrm{SOR}_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & -1 & 0 & \cdots & 0 \\ 1 & 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 1 & 0 & 0 & & -1 \end{bmatrix}.$$

The SOR_n is the sparsest matrix that splits off the one-representation in a permutation representation. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the SOR of arbitrary characteristic. The default characteristic is zero.

```
gap> SORAMat( 4 );
SORAMat(4)
gap> SORAMat( 4, 7);
SORAMat(4, 7)
```

1.36 ScalarMultipleAMat

`ScalarMultipleAMat(` *s*`,` *A* `)` or *s* `*` *A*

Let s be a field element and A an amat. `ScalarMultipleAMat` returns an amat of type `"scalarMultiple"` representing the scalar multiple of s with A, which must have common characteristic otherwise an error is signaled. Note that s and A can be accessed in the fields `.scalar` resp. `.element` of the result.

```
gap> A := AMatPerm( (1,2,3), 4);
AMatPerm((1,2,3), 4)
gap> ScalarMultipleAMat( E(3), A );
E(3) * AMatPerm((1,2,3), 4)
gap> 2 * A;
2 * AMatPerm((1,2,3), 4)
```

1.37 Product and Quotient of AMats

A * *B*

Let A and B be amats. *A* * *B* returns an amat of type `"product"` representing the product of A and B, which must have compatible sizes and common characteristic otherwise an error is signaled. Note that the factors can be accessed in the field `.factors` of the result.

```
gap> A := AMatPerm( (1,2,3), 4);
AMatPerm((1,2,3), 4)
gap> B := AMatMat( [ [1, 2], [3, 4], [5, 6], [7, 8] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ] ]
)
gap> A * A;
AMatPerm((1,2,3), 4) *
AMatPerm((1,2,3), 4)
gap> C := A * B;
AMatPerm((1,2,3), 4) *
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ] ]
)
gap> C.type;
"product"
```

A / *B*

Let A and B be amats. *A* / *B* returns an amat of type `"product"` representing the quotient of A and B. The sizes and characteristics of A and B must be compatible, B must be square and invertible otherwise an error is signaled.

```
gap> A := AMatPerm( (1,2,3), 4);
AMatPerm((1,2,3), 4)
gap> B := DiagonalAMat( [1, E(3), 1, 3] );
DiagonalAMat([ 1, E(3), 1, 3 ])
gap> A / B;
AMatPerm((1,2,3), 4) *
DiagonalAMat([ 1, E(3), 1, 3 ]) ^ -1
```

1.38 PowerAMat

`PowerAMat(` *A*, *n* `)` or *A* ^ *n*
`PowerAMat(` *A*, *n*, *hint* `)`

Let A be an amat and n an integer. `PowerAMat` returns an amat of type `"power"` representing the power of A with n. A must be square otherwise an error is signaled. If n is negative then A is checked for invertibility if the hint `"invertible"` is not supplied. Note that A and n can be accessed in the fields `.element` resp. `.exponent` of the result.

```
gap> A := AMatPerm( (1,2,3), 4);
AMatPerm((1,2,3), 4)
gap> B := PowerAMat(A, 3);
AMatPerm((1,2,3), 4) ^ 3
gap> B ^ -2;
( AMatPerm((1,2,3), 4) ^ 3
) ^ -2
```

1.39 ConjugateAMat

`ConjugateAMat(` A`,` B `)` or A `^` B
`ConjugateAMat(` A`,` B`,` *hint* `)`

Let A and B be amats. `ConjugateAMat` returns an amat of type `"conjugate"` representing the conjugate of A with B (i.e. the matrix defined by $B^{-1} \cdot A \cdot B$). A and B must be square otherwise an error is signaled. B is checked for invertibility if the hint `"invertible"` is not supplied. Note that A and B can be accessed in the fields `.element` resp. `conjugation` of the result.

```
gap> A := AMatMon( Mon( (1,2), [1, E(4), -1] ) );
AMatMon( Mon(
  (1,2),
  [ 1, E(4), -1 ]
) )
gap> B := DFTAMat( 3 );
DFTAMat(3)
gap> ConjugateAMat( A, B, "invertible" );
ConjugateAMat(
  AMatMon( Mon(
    (1,2),
    [ 1, E(4), -1 ]
  ) ),
  DFTAMat(3)
)
gap> B ^ SORAMat( 3 );
ConjugateAMat(
  DFTAMat(3),
  SORAMat(3)
)
```

1.40 DirectSumAMat

`DirectSumAMat(` A_1`, ...,` A_k `)`

`DirectSumAMat` returns an amat of type `"directSum"` representing the direct sum of the amats A_1, ..., A_k, which must have common characteristic otherwise an error is signaled. Note that the direct summands can be accessed in the field `.summands` of the result.

```
gap> A1 := AMatMat( [ [1, 2] ] );
AMatMat(
  [ [ 1, 2 ] ]
)
gap> A2 := DFTAMat( 2 );
DFTAMat(2)
gap> A3 := AMatPerm( (1,2), 2 );
AMatPerm((1,2), 2)
gap> DirectSumAMat( E(3) * A1, A2 ^ 2, A3 );
DirectSumAMat(
  E(3) * AMatMat( [ [ 1, 2 ] ] ),
  DFTAMat(2) ^ 2,
  AMatPerm((1,2), 2)
)
```

`DirectSumAMat(` *list* `)`

`DirectSumAMat` returns an amat of type `"directSum"` representing the direct sum of the amats in *list*. The amats must have common characteristic otherwise an error is signaled. The direct summands can be accessed in the field `.summands` of the result.

```
gap> A := DiagonalAMat( [ Z(3), Z(3)^2 ]);
DiagonalAMat([ Z(3), Z(3)^0 ])
gap> B := AMatPerm( (1,2), 3, 3);
AMatPerm((1,2), 3, GF(3))
gap> DirectSumAMat( [A, B] );
DirectSumAMat(
  DiagonalAMat([ Z(3), Z(3)^0 ]),
  AMatPerm((1,2), 3, GF(3))
)
```

1.41 TensorProductAMat

`TensorProductAMat(` A_1, ..., A_k `)`

`TensorProductAMat` returns an amat of type `"tensorProduct"` representing the tensor product (or Kronecker product) of the amats A_1, ..., A_k, which must have common characteristic otherwise an error is signaled. Note that the tensor factors can be accessed in the field `.factors` of the result.

```
gap> A := IdentityPermAMat( 2 );
IdentityPermAMat(2)
gap> B := AMatMat( [ [1, 2, 3], [4, 5, 6] ] );
AMatMat(
  [ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
)
gap> TensorProductAMat( A, B );
TensorProductAMat(
  IdentityPermAMat(2),
```

```
    AMatMat(
      [ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
    )
  )
```

`TensorProductAMat(` *list* `)`

`TensorPoductAMat` returns an amat of type `"tensorProduct"` representing the tensor product of the amats in *list*. The amats must have common characteristic otherwise an error is signaled. The tensor factors can be accessed in the field `.factors` of the result.

```
gap> A := AMatPerm( (1,2), 3 );
AMatPerm((1,2), 3)
gap> B := AMatMat( [ [1], [2] ]);
AMatMat(
  [ [ 1 ], [ 2 ] ]
)
gap> TensorProductAMat( [A ^ 2, 2 * B] );
TensorProductAMat(
  AMatPerm((1,2), 3) ^ 2,
  2 * AMatMat(
    [ [ 1 ], [ 2 ] ]
  )
)
```

1.42 GaloisConjugateAMat

`GaloisConjugateAMat(` A, k `)`
`GaloisConjugateAMat(` A, *aut* `)`

`GaloisConjugateAMat` returns an amat which represents a Galois conjugate of the amat A. The conjugating automorphism may either be a field automorphism *aut* or an integer k specifying the automorphism `x -> GaloisCyc(x,` k`)` in the case characteristic $= 0$ or `x -> x^(FrobeniusAut^`k`)` in the case characteristic $= p$ prime. Note that A and k/*aut* can be accessed in the fields `.element` resp. `.galoisAut` of the result.

```
gap> A := DiagonalAMat( [1, E(3)] );
DiagonalAMat([ 1, E(3) ])
gap> GaloisConjugateAMat( A, -1 );
GaloisConjugateAMat(
  DiagonalAMat([ 1, E(3) ]),
  -1
)
gap> aut := FrobeniusAutomorphism( GF(4) );
FrobeniusAutomorphism( GF(2^2) )
gap> B := AMatMon( Mon( (1,2), [ Z(2)^0, Z(2^2) ] ) );
AMatMon( Mon(
  (1,2),
  [ Z(2)^0, Z(2^2) ]
) )
gap> GaloisConjugateAMat( B, aut );
```

```
GaloisConjugateAMat(
  AMatMon( Mon(
    (1,2),
    [ Z(2)^0, Z(2^2) ]
  ) ),
  FrobeniusAutomorphism( GF(2^2) )
)
```

1.43 Comparison of AMats

A = *B*
A <> *B*

The equality operator `=` evaluates to `true` if the amats A and B are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the amats A and B are not equal and to `false` otherwise.

Two amats are equal iff they define the same matrix.

```
gap> A := DiagonalAMat( [E(3), 1] );
DiagonalAMat([ E(3), 1 ])
gap> B := A ^ 3;
DiagonalAMat([ E(3), 1 ]) ^ 3
gap> B = IdentityPermAMat( 2 );
true
```

A < *B*
A <= *B*
A >= *B*
A > *B*

The operators `<`, `<=`, `>=`, and `>` evaluate to `true` if the amat A is strictly less than, less than or equal to, greater than or equal to, and strictly greater than the amat B.

The ordering of amats is defined via the ordering of records.

1.44 Converting AMats

The following sections describe the functions for the convertability and conversion of amats to permutations, mons (see 1.2) and matrices.

The names of the conversion functions are chosen according to the usual GAP-convention: `ChalkCheese` makes chalk from cheese. The parts in the name (chalk, cheese) are

Perm	– a GAP-permutation, e.g. (1,2)
Mon	– a mon object, e.g. Mon((1,2), 2) (see 1.2)
Mat	– a GAP-matrix, e.g. [[1,2],[3,4]]
AMat	– an amat of any type
PermAMat	– an amat of type “perm”
MonAMat	– an amat of type “mon”
MatAMat	– an amat of type “mat”

1.45 IsIdentityMat

`IsIdentityMat(` *A* `)`

`IsIdentityMat` returns `true` if the matrix represented by the amat A is the identity matrix and `false` otherwise. Note that the name of the function is not `IsIdentityAMat` since A can be of any type but represents an identity matrix in the mathematical sense.

```
gap> IsIdentityMat(AMatPerm( (1,2), 3 ));
false
gap> A := DiagonalAMat( [Z(3), Z(3)] ) ^ 2;
DiagonalAMat([ Z(3), Z(3) ]) ^ 2
gap> IsIdentityMat(A);
true
```

1.46 IsPermMat

`IsPermMat(` *A* `)`

`IsPermMat` returns `true` if the matrix represented by the amat A is a permutation matrix and `false` otherwise. The name of the function is not `IsPermAMat` since A can be of any type but represents a permutation matrix in the mathematical sense. Note that `IsPermMat` sets and tests A`.isPermMat`.

```
gap> IsPermMat( AMatMon( Mon( (1,2), [1, -1] )));
false
gap> IsPermMat( DiagonalAMat( [Z(3), Z(9)] ) ^ 8);
true
```

1.47 IsMonMat

`IsMonMat(` *A* `)`

`IsMonMat` returns `true` if the matrix represented by the amat A is a monomial matrix (a matrix containing exactly one entry $\neq 0$ in every row and column) and `false` otherwise. The name of the function is not `IsMonAMat` since A can be of any type but represents a monomial matrix in the mathematical sense. Note that `IsMonMat` sets and tests A`.isMonMat`.

```
gap> IsMonMat( AMatPerm( (1,2), 3 ));
true
gap> IsMonMat( AMatPerm( (1,2,3), 3 ) ^ DFTAMat(3) );
true
```

1.48 PermAMat

`PermAMat(` *A* `)`

Let A be an amat. `PermAMat` returns the permutation represented by A if A is a permutation matrix (i.e. `IsPermMat(` *A* `) = true`) and `false` otherwise. Note that `PermAMat` sets and tests A`.perm`.

```
gap> PermAMat(AMatPerm( (1,2), 5 ));
(1,2)
```

```
gap> A := AMatMat( [ [Z(3)^0, Z(3)], [0*Z(3), Z(3)^0] ] );
AMatMat(
  [ [ Z(3)^0, Z(3) ], [ 0*Z(3), Z(3)^0 ] ]
)
gap> PermAMat(A);
false
gap> PermAMat(A ^ 3);
()
```

1.49 MonAMat

`MonAMat(` *A* `)`

Let A be an amat. `MonAMat` returns the mon (see 1.2) represented by A if A is a monomial matrix (i.e. `IsMonMat(` *A* `) = true`) and `false` otherwise. Note that `MonAMat` sets and tests A`.mon`.

```
gap> MonAMat(AMatPerm( (1,2,3), 5 ));
Mon( (1,2,3), 5 )
gap> MonAMat(AMatPerm( (1,2,3), 3 ) ^ DFTAMat(3) );
Mon( [ 1, E(3), E(3)^2 ] )
gap> MonAMat( AMatMat( [ [1, 2] ] ));
false
```

1.50 MatAMat

`MatAMat(` *A* `)`

`MatAMat` returns the matrix represented by the amat A. Note that `MatAMat` sets and tests A`.mat`.

```
gap> MatAMat( AMatPerm( (1,2), 3, 2 ));
[ [ 0*Z(2), Z(2)^0, 0*Z(2) ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> MatAMat(DFTAMat(3));
[ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ]
gap> A := IdentityPermAMat(2);
IdentityPermAMat(2)
gap> B := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> MatAMat(TensorProductAMat(A, B));
[ [ 1, 2, 0, 0 ], [ 3, 4, 0, 0 ], [ 0, 0, 1, 2 ], [ 0, 0, 3, 4 ] ]
```

1.51 PermAMatAMat

`PermAMatAMat(` *A* `)`

Let A be an amat. `PermAMatAMat` returns an amat of type `"perm"` equal to A if A is a permutation matrix (i.e. `IsPermMat(` *A* `) = true`) and `false` otherwise.

```
gap> PermAMatAMat(AMatMon(Mon( (1,2), 3 )));
AMatPerm((1,2), 3)
gap> PermAMatAMat(DiagonalAMat( [E(3), 1] ) ^ 3);
IdentityPermAMat(2)
gap> PermAMatAMat(AMatMat( [ [1,2] ] ));
false
```

1.52 MonAMatAMat

`MonAMatAMat(` A `)`

Let A be an amat. `MonAMatAMat` returns an amat of type `"mon"` equal to A if A is a monomial matrix (i.e. `IsMonMat(` A `) = true`) and `false` otherwise.

```
gap> MonAMat(AMatPerm( (1,2), 3 ));
Mon( (1,2), 3 )
gap> MonAMat(DFTAMat(3)^2);
Mon(
  (2,3),
  [ 3, 3, 3 ]
)
gap> MonAMat(AMatMat( [ [1, 2] ] ));
false
```

1.53 MatAMatAMat

`MatAMatAMat(` A `)`

`MatAMatAMat` returns an amat of type `"mat"` equal to A.

```
gap> A := AMatPerm( (1,2), 2 );
AMatPerm((1,2), 2)
gap> B := AMatMat( [ [1,2] ] );
AMatMat(
  [ [ 1, 2 ] ]
)
gap> MatAMatAMat(DirectSumAMat(A, B));
AMatMat(
  [ [ 0, 1, 0, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 1, 2 ] ]
)
```

1.54 Functions for AMats

The following sections describe useful functions for the calculation with amats (e.g. calculation of the inverse, determinant of an amat as well as simplifying amats). Most of these functions can take great advantage of the highly structured form of the amats.

1.55 InverseAMat

`InverseAMat(` A `)`

`InverseAMat` returns an amat representing the inverse of the amat A. If A is not invertible, an error is signaled. The function uses mathematical rules to invert the direct sum, tensor product etc. of matrices. Note that `InverseAMat` sets and tests A`.inverse`.

```
gap> A := AMatPerm( (1,2), 3);
AMatPerm((1,2), 3)
gap> B := AMatMat( [ [1,2], [3,4] ]);
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> C := DiagonalAMat( [ E(3), 1] );
DiagonalAMat([ E(3), 1 ])
gap> D := DirectSumAMat(A, TensorProductAMat(B, C));
DirectSumAMat(
  AMatPerm((1,2), 3),
  TensorProductAMat(
    AMatMat( [ [ 1, 2 ], [ 3, 4 ] ] ),
    DiagonalAMat([ E(3), 1 ])
  )
)
gap> InverseAMat(D);
DirectSumAMat(
  AMatPerm((1,2), 3),
  TensorProductAMat(
    AMatMat(
      [ [ -2, 1 ], [ 3/2, -1/2 ] ],
      "invertible"
    ),
    DiagonalAMat([ E(3)^2, 1 ])
  )
)
```

1.56 TransposedAMat

`TransposedAMat(` A `)`

`TransposedAMat` returns an amat representing the transpose of the amat A. The function uses mathematical rules to transpose the direct sum, tensor product etc. of matrices.

```
gap> A := AMatPerm( (1,2,3), 3);
AMatPerm((1,2,3), 3)
gap> B := AMatMat( [ [1, 2] ] );
AMatMat(
  [ [ 1, 2 ] ]
)
gap> TransposedAMat(TensorProductAMat(A, B));
TensorProductAMat(
  AMatPerm((1,3,2), 3),
  AMatMat(
    [ [ 1 ], [ 2 ] ]
```

```
    )
  )
```

1.57 DeterminantAMat

`DeterminantAMat(` A `)`

`DeterminantAMat` returns the determinant of the amat A. If A is not square an error is signaled. The function uses mathematical rules to calculate the determinant of the direct sum, tensor product etc. of matrices. Note that `DeterminantAMat` sets and tests A`.determinant`.

```
gap> A := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> B := AMatPerm( (1,2), 2 );
AMatPerm((1,2), 2)
gap> DeterminantAMat(TensorProductAMat(A, B));
4
```

1.58 TraceAMat

`TraceAMat(` A `)`

`TraceAMat` returns the trace of the amat A. If A is not square an error is signaled. The function uses mathematical rules to calculate the trace of direct sums, tensor product etc. of matrices. Note that `TraceAMat` sets and tests A`.trace`.

```
gap> A := DFTAMat(2);
DFTAMat(2)
gap> B := DiagonalAMat( [1, 2, 3] );
DiagonalAMat([ 1, 2, 3 ])
gap> TraceAMat(DirectSumAMat( A^2, B ));
10
```

1.59 RankAMat

`RankAMat(` A `)`

`RankAMat` returns the rank of the amat A. Note that `RankAMat` sets and tests A`.rank`.

```
gap> RankAMat(AllOneAMat(100));
1
gap> RankAMat(AMatPerm( (1,2), 10 ));
10
```

1.60 SimplifyAMat

`SimplifyAMat(` A `)`

`SimplifyAMat` returns a simplified amat representing the same matrix as the amat A. The simplification is performed recursively according to certain rules. E.g. the following simplifications are performed:

- If A represents a permutation matrix, monomial matrix then an amat of type "perm", "mon" resp. is returned.
- In a product resp. tensor product, trivial factors are omitted.
- Trivial conjugation is omitted.
- In a direct sum adjacent permutation/monomial matrices are put together.
- In a product adjacent permutation/monomial matrices are multiplied together.
- Successive scalars are multiplied together.
- Successive exponents are multiplied together, negative exponents are evaluated using `InverseAMat`.

Note that important information about the matrix is shifted to the simplification.

```
gap> A := IdentityPermAMat( 3 );
IdentityPermAMat(3)
gap> B := DiagonalAMat( [E(3), 1, 1] );
DiagonalAMat([ E(3), 1, 1 ])
gap> C := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> D := DirectSumAMat(A ^ -1, 1 * B * A, C);
DirectSumAMat(
  IdentityPermAMat(3) ^ -1,
  ( 1 * DiagonalAMat([ E(3), 1, 1 ])
  ) *
  IdentityPermAMat(3),
  AMatMat(
    [ [ 1, 2 ], [ 3, 4 ] ]
  )
)
gap> SimplifyAMat(D);
DirectSumAMat(
  IdentityPermAMat(3),
  DiagonalAMat([ E(3), 1, 1 ]),
  AMatMat(
    [ [ 1, 2 ], [ 3, 4 ] ]
  )
)
```

1.61 kbsAMat

`kbsAMat(` A_1`, ...,` A_k `)`

`kbsAMat` returns the joined kbs (conjugated blockstructure) of the amats A_1, ..., A_k. The amats must be square and of common size and characteristic otherwise an error is signaled. The joined kbs of a list of $(n \times n)$-matrices is a partition of $\{1, \ldots, n\}$ representing their common blockstructure. For an exact definition see 1.169.

```
gap> A := IdentityPermAMat(2);
IdentityPermAMat(2)
gap> B := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> kbsAMat(TensorProductAMat(A, B));
[ [ 1, 2 ], [ 3, 4 ] ]
gap> kbsAMat(AMatPerm( (1,3)(2,4), 5 ));
[ [ 1, 3 ], [ 2, 4 ], [ 5 ] ]
```

`kbsAMat(` *list* `)`

`kbsAMat` returns the joined kbs of the amats in *list* (see above).

1.62 kbsDecompositionAMat

`kbsDecompositionAMat(` A `)`

`kbsDecompositionAMat` decomposes the amat A into a conjugated (by an amat of type `"perm"`) direct sum of amats of type `"mat"` as far as possible. If A is not square an error is signaled. The decomposition is performed according to the kbs (see 1.169) of A which is a partition of $\{1, \ldots, n\}$ ($n =$ number of rows of A) describing the blockstructure of A.

```
gap> A := AMatMat( [[1,0,2,0], [0,1,0,2], [3,0,4,0], [0,3,0,4]] );
AMatMat(
  [ [ 1, 0, 2, 0 ], [ 0, 1, 0, 2 ], [ 3, 0, 4, 0 ], [ 0, 3, 0, 4 ] ]
)
gap> kbsDecompositionAMat(A);
ConjugateAMat(
  DirectSumAMat(
    AMatMat(
      [ [ 1, 2 ], [ 3, 4 ] ]
    ),
    AMatMat(
      [ [ 1, 2 ], [ 3, 4 ] ]
    )
  ),
  AMatPerm((2,3), 4)
)
```

1.63 AMatSparseMat

`AMatSparseMat(` M `)` `AMatSparseMat(` M`,` *match-blocks* `)`

Let M be a sparse matrix (i.e. containing entries $\neq 0$). `AMatSparseMat` returns an amat of the form $P_1 \cdot E_1 \cdot D \cdot E_2 \cdot P_2$ where (for $i = 1, 2$) P_i are amats of type `"perm"`, E_i are identity-amats (might be rectangular) and D is an amat of type "directSum". If *match-blocks* is true or not provided then, furthermore, the permutations p_1 and p_2 are chosen such that equivalent summands of D are equal and collected together by a tensor product. If *match-blocks* is false this is not done. The major part of the work is done by the function DirectSummandsPermutedMat (see 1.168). Use the function SimplifyAMat (see 1.60) for simplification of the result.

For an explanation of the algorithm see [Egn97].

```
gap> M := [[0,0,0,0],[0,1,0,2],[0,0,3,0],[0,4,0,5]];;
gap> PrintArray(M);
[ [  0,  0,  0,  0 ],
  [  0,  1,  0,  2 ],
  [  0,  0,  3,  0 ],
  [  0,  4,  0,  5 ] ]
gap> AMatSparseMat(M);
AMatPerm((1,4,3), 4) *
IdentityMatAMat([ 4, 3 ]) *
DirectSumAMat(
  TensorProductAMat(
    IdentityPermAMat(1),
    AMatMat(
      [ [ 3 ] ]
    )
  ),
  TensorProductAMat(
    IdentityPermAMat(1),
    AMatMat(
      [ [ 1, 2 ], [ 4, 5 ] ]
    )
  )
) *
IdentityMatAMat([ 3, 4 ]) *
AMatPerm((1,3,4), 4)
```

1.64 SubmatrixAMat

SubmatrixAMat(A, *inds*)

Let A be an amat and *inds* a set of positive integers. SubmatrixAMat returns an amat of type "mat" representing the submatrix of A defined by extracting all entries with row and column index in *inds*.

```
gap> A := AMatPerm( (1,2), 2 );
AMatPerm((1,2), 2)
gap> B := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> SubmatrixAMat(TensorProductAMat(A, B), [2,3] );
AMatMat(
```

```
  [ [ 0, 3 ], [ 2, 0 ] ]
)
```

1.65 UpperBoundLinearComplexityAMat

`UpperBoundLinearComplexityAMat(` A `)`

`UpperBoundLinearComplexityAMat` returns an upper bound for the linear complexity of the amat A according to the complexity model L_∞ of Clausen/Baum, [CB93]. The linear complexity is a measure for the complexity of the matrix-vector multiplication of a given matrix with an arbitrary vector.

```
gap> UpperBoundLinearComplexityAMat(DFTAMat(2));
2
gap> UpperBoundLinearComplexityAMat(DiagonalAMat( [2, 3] ));
2
gap> A := AMatPerm( (1,2), 3);
AMatPerm((1,2), 3)
gap> B := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> UpperBoundLinearComplexityAMat(TensorProductAMat(A, B));
24
```

1.66 AReps

The class **ARep** (**A**bstract **Rep**resentations) is created to represent and calculate efficiently with structured matrix representations of finite groups up to equality, e.g. expressions like $(\phi \uparrow_T G)^M \otimes \psi$ where ϕ, ψ are representations and $\uparrow, \otimes$ denotes the induction resp. inner tensor product of representations. The implementation idea is the same as with the class **AMat** (see 1.22), i.e. a representation is a record containing the necessary information (e.g. degree, characteristic, list of images on the generators) to define a representation up to equality. The elements of **ARep** are called "areps" and are no group homomorphisms in the sense of GAP (which is the reason for the term "abstract" representation). Special care is taken of permutation and monomial representations, which can be represented very efficiently by storing a list of permutations or mons (see 1.2) instead of matrices as images on the generators.

Areps can represent representations of any finite group and any characteristic including modular (characteristic divides group size) representations, but most of the higher functions will only work in the non-modular case or even only in the case of characteristic zero. These restrictions are always indicated in the description of the respective function.

Basic constructors allow to create areps, e.g. by supplying the list of images on the generators (see `ARepByImages`, 1.73). Since GAP allows the manipulation of the generators given to construct a group, it is important for consistency to have a field with generators one can rely on. This is realized in the function `GroupWithGenerators`, 1.67.

Higher constructors allow to construct inductions (see `InductionARep`, 1.81), direct sums (see `DirectSumARep`, 1.77), inner tensor products (see `InnerTensorProductARep`, 1.78) etc. from given areps.

Some remarks on the design of **ARep**: The class **ARep** is a term algebra for matrix representations of finite groups (see also **AMat**, 1.22). The simplification strategy is extremely conservative, which means that even trivial expressions like `GaloisConjugate(`R`, `id`)` are only simplified upon explicit request. As in **AMat** we use the "hint"-concept extensively to suppress unnecessary expensive computations of little interest. The class **AMat** is used in **ARep** in three ways: 1. for images under areps, 2. for conjugating matrices (change of base of the underlying vector space) and 3. for elements of the intertwining space of two areps. Note that 3. requires non-invertible or even rectangular matrices to be represented. A special point that deserves mentioning is the way in which areps act as homomorphisms anf how they are defined. Areps are *no* GAP-homomorphisms. We simply did not manage to implement **ARep** as a term algebra *and* as GAP-homomorphisms in a relyable and efficient way which avoids maximal confusion. In addition, working with **ARep** usually involves many representations of the same group. This is supported in the most obvious way by fixing the list of generators used to create the group (see 1.67) and only varying the list of images. Although this strategy differs from the approach in GAP (which deliberately manipulates the generating list used to construct the group) it turned out to be very useful and efficient in the situation at hand.

We define an arep recursively in Backus-Naur-Form as the disjoint union of the following cases.

arep ::=		
; atomic cases		
perm	;	"perm"
| *mon*	;	"mon"
| *mat*	;	"mat"
; composed cases		
| *arep* ^ *arep*	;	"conjugate"
| $arep \oplus \ldots \oplus arep$	;	"directSum"
| $arep \otimes \ldots \otimes arep$	;	"innerTensorProduct"
| *arep* # ... # *arep*	;	"outerTensorProduct"
| $arep \downarrow subgrp$	;	"restriction"
| $arep \uparrow supgrp$, *transversal*	;	"induction"
| Extension(*arep*, *ext-character*)	;	"extension"
| GaloisConjugate(*arep*, *aut*)	;	"galoisConjugate"

An arep R is a record with the following fields mandatory to all types of areps.

`isARep`	:=	`true`
`operations`	:=	`AMatOps`
`char`	:	characteristic of the base field
`degree`	:	degree of the representation
`source`	:	the group being represented, which must contain the field `.theGenerators`, see 1.67
`type`	:	a string identifying the type of R

The cases as stated above are distinguished by the field `.type` of an arep R. Depending on the type additional fields are mandatory as follows.

`type = "perm"`:

`theImages`	list of permutations for the images of `source.theGenerators`

`type = "mon"`:

`theImages`	list of mons (see 1.2) for the images of `source.theGenerators`

`type = "mat"`:

`theImages`	list of matrices for the images of `source.theGenerators`

`type = "mat"`:

`rep`	an arep to be conjugated
`conjugation`	an amat (see 1.22) conjugating `rep`

`type = "directSum"`:

`summands`	list of areps of the same source and characteristic

`type = "innerTensorProduct"`:

`factors`	list of areps of the same characteristic

`type = "outerTensorProduct"`:

`factors`	list of areps of the same characteristic

`type = "restriction"`:

`rep`	an arep of a supergroup of `source`, the group `source` and `rep.source` have the same parent group

`type = "induction"`:

`rep`	an arep of a subgroup of `source`, the group `source` and `rep.source` have the same parent group
`transversal`	a right transversal of `Cosets(source, rep.source)`

`type = "galoisConjugate"`:

`rep`	an arep to be conjugated
`galoisAut`	the Galois automorphism

Note that most of the function concerning areps require calculation in the source group. Hence it is most useful to choose aggroups or permutation groups as sources if possible. Furthermore there is an important difference between the *type of an arep* and the *type of the representation being represented by the arep*: E.g. an arep can be of type "induction" but the representation is in fact a permutation representation. This distinction is reflected in the naming of the functions: "XARep" refers to the type of the arep, "XRep" to the type of the representation being represented,

Here a short overview of the function concerning areps. sections 1.67 – 1.83 are concerned with the construction of areps, sections 1.84 – 1.92 are concerned with the evaluation of an arep at a point, tests for equivalence and irreducibility, construction of an arep with given character etc., sections 1.94 – 1.99 deal with the conversion of areps to areps of type `"perm"`, `"mon"`, `"mat"`. Sections 1.100 – 1.123 provide function for the computation of the intertwining space of areps and a plenty of functions for monomial areps. The most important function here is `DecompositionMonRep` (see 1.123) performing the decomposition of a monomial arep including the computation of a highly structured decomposition matrix.

The basic functions concerning areps are implemented in the file `"arep/lib/arep.g"`, the higher functions in `"arep/lib/arepfcts.g"`.

For details on constructive representation theory and the theoretical background of the higher functions please refer to [Püs98].

1.67 GroupWithGenerators

`GroupWithGenerators(` *G* `)`

Let G be a group. `GroupWithGenerators` returns G with the field G`.theGenerators` being set to a fixed non-empty generating set of G. This function is created because GAP has the freedom to manipulate the generators given to construct a group. Based on the list G`.theGenerators` areps can be constructed, e.g. by the images on that list (`ARepByImages`, 1.73). If an arep for a group G is constructed with the field G`.theGenerators` unbound a warning is signaled and the field is set.

```
gap> G := Group( (1,2) );
Group( (1,2) )
gap> GroupWithGenerators(G);
Group( (1,2) )
gap> G.theGenerators;
[ (1,2) ]
gap> G := Group( () );
Group( () )
gap> GroupWithGenerators(G);
Group( () )
gap> G.theGenerators;
[ () ]
gap> G.generators;
[  ]
```

`GroupWithGenerators(` *list* `)`

`GroupWithGenerators` returns the group G generated by the elements in *list*. The field G`.theGenerators` is set to *list*. For the reason of this function see above.

```
gap> G := GroupWithGenerators( [ (), (1,2), (1,2,3) ] );
Group( (1,2), (1,2,3) )
gap> G.theGenerators;
[ (), (1,2), (1,2,3) ]
gap> G.generators;
[ (1,2), (1,2,3) ]
```

1.68 TrivialPermARep

`TrivialPermARep(` *G* `)`
`TrivialPermARep(` *G*`,` *d* `)`
`TrivialPermARep(` *G*`,` *d*`,` *char* `)`
`TrivialPermARep(` *G*`,` *d*`,` *field* `)`

`TrivialPermARep` returns an arep of type `"perm"` representing the one representation of the group G of degree d. The default degree is 1. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the one representation of arbitrary characteristic. The default characteristic is zero.

```
gap> G := GroupWithGenerators( [(1,2), (3,4)] );
Group( (1,2), (3,4) )
```

```
gap> TrivialPermARep(G, 2, 3);
TrivialPermARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 2, GF(3) )
gap> G := GroupWithGenerators( [(1,2), (3,4)] );
Group( (1,2), (3,4) )
gap> R := TrivialPermARep(G, 2, 3);
TrivialPermARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 2, GF(3) )
gap> R.degree;
2
gap> R.char;
3
```

1.69 TrivialMonARep

`TrivialMonARep(` *G* `)`
`TrivialMonARep(` *G*, *d* `)`
`TrivialMonARep(` *G*, *d*, *char* `)`
`TrivialMonARep(` *G*, *d*, *field* `)`

`TrivialMonARep` returns an arep of type `"mon"` representing the one representation of the group G of degree d. The default degree is 1. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the one representation of arbitrary characteristic. The default characteristic is zero.

```
gap> G := GroupWithGenerators( [(1,2), (3,4)] );
Group( (1,2), (3,4) )
gap> R := TrivialMonARep(G, 2);
TrivialMonARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 2 )
gap> R.theImages;
[ Mon( (), 2 ), Mon( (), 2 ) ]
```

1.70 TrivialMatARep

`TrivialMatARep(` *G* `)`
`TrivialMatARep(` *G*, *d* `)`
`TrivialMatARep(` *G*, *d*, *char* `)`
`TrivialMatARep(` *G*, *d*, *field* `)`

`TrivialMatARep` returns an arep of type `"mat"` representing the one representation of the group G of degree d. The default degree is 1. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the one representation of arbitrary characteristic. The default characteristic is zero.

```
gap> G := GroupWithGenerators( [(1,2), (3,4)] );
Group( (1,2), (3,4) )
gap> R := TrivialMatARep(G);
TrivialMatARep( GroupWithGenerators( [ (1,2), (3,4) ] ) )
gap> R.theImages;
[ [ [ 1 ] ], [ [ 1 ] ] ]
```

1.71 RegularARep

`RegularARep(` G `)`
`RegularARep(` G`,` *char* `)`
`RegularARep(` G`,` *field* `)`

`RegularARep` returns an arep of type `"induction"` representing the regular representation of G. The regular representation is defined (up to equality) by the induction $R = (1_{\mathsf{E}} \uparrow_T G)$ of the trivial representation (of degree one) of the trivial subgroup `E` of G with the transversal T being the ordered list of elements of G. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the regular representation of arbitrary characteristic. The default characteristic is zero.

```
gap> G := GroupWithGenerators(SymmetricGroup(3));
Group( (1,3), (2,3) )
gap> RegularARep(G);
RegularARep( GroupWithGenerators( [ (1,3), (2,3) ] ) )
gap> RegularARep(G, GF(2));
RegularARep( GroupWithGenerators( [ (1,3), (2,3) ] ), GF(2) )
```

1.72 NaturalARep

`NaturalARep(` G `)`
`NaturalARep(` G`,` d `)`
`NaturalARep(` G`,` d`,` *char* `)`
`NaturalARep(` G`,` d`,` *field* `)`

Let G be a mongroup or a matrix group (for mons see 1.2). `NaturalARep` returns an arep of type `"mon"` or `"mat"` resp. representing the representation given by G, which means that G is taken as a representation of itself.

For a permutation group G the desired degree d of the representation has to be supplied. The returned arep is of type `"perm"`. If d is smaller than the largest moved point of G an error is signaled. As optional parameter a characteristic *char* or a *field* can be supplied (if G is a permutation group). Note that a mongroup or a matrix group as source of an arep slows down most of the calculations with it.

```
gap> G := GroupWithGenerators( [ (1,2), (1,2,3) ] );
Group( (1,2), (1,2,3) )
gap> R := NaturalARep(G, 4);
NaturalARep( GroupWithGenerators( [ (1,2), (1,2,3) ] ), 4 )
gap> R.theImages;
[ (1,2), (1,2,3) ]
gap> R.degree;
4
gap> G := GroupWithGenerators( [ Mon( (1,2), [E(4), 1] ) ] );
Group( Mon(
  (1,2),
  [ E(4), 1 ]
) )
gap> NaturalARep(G);
```

```
NaturalARep(
  GroupWithGenerators( [ Mon(
      (1,2),
      [ E(4), 1 ]
    ) ] ) )
```

1.73 ARepByImages

ARepByImages(G, *list*)
ARepByImages(G, *list*, *hint*)

ARepByImages(G, *list*, d)
ARepByImages(G, *list*, d, *hint*)
ARepByImages(G, *list*, d, *char*)
ARepByImages(G, *list*, d, *field*)
ARepByImages(G, *list*, d, *char*, *hint*)
ARepByImages(G, *list*, d, *field*, *hint*)

`ARepByImages` allows to construct an arep of the group G by supplying the *list* of images on the list G`.theGenerators`.

Let *list* contain mons (see 1.2). `ARepByImages` returns an arep of type `"mon"` defined by mapping G`.theGenerators` elementwise onto *list*.

Let *list* contain matrices. `ARepByImages` returns an arep of type `"mat"` defined by mapping G`.theGenerators` elementwise onto *list*.

Let *list* contain permutations. `ARepByImages` returns an arep of type `"perm"` and degree d defined by mapping G`.theGenerators` elementwise onto *list*. If d is smaller than the largest moved point of G an error is signaled. As optional parameter a characteristic *char* or a *field* can be supplied to obtain an arep of arbitrary characteristic.

In all cases the *hint* `"hom"` or `"faithful"` can be supplied to indicate that the list of images does define a homomorphism or even a faithful homomorphism respectively. If no hint is supplied it is checked whether the list of images defines a homomorphism.

```
gap> G := GroupWithGenerators( [(1,2), (1,2,3)] );
Group( (1,2), (1,2,3) )
gap> ARepByImages(G, [ Mon( [-1] ), Mon( [1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2), (1,2,3) ] ),
  [ Mon( [ -1 ] ), Mon( (), 1 ) ],
  "hom"
)
gap> L := [ [ [Z(2), Z(2)], [0*Z(2), Z(2)] ], IdentityMat(2, GF(2)) ];
[ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] ]
gap> ARepByImages(G, L);
ARepByImages(
  GroupWithGenerators( [ (1,2), (1,2,3) ] ),
  [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ],
    [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ]
```

```
  ],
  GF(2),
  "hom"
)
gap> ARepByImages(G, [ (1,2), () ], 3);
ARepByImages(
  GroupWithGenerators( [ (1,2), (1,2,3) ] ),
  [ (1,2), () ],
  3, # degree
  "hom"
)
gap> ARepByImages(G, [ (1,2), () ], 3, "hom");
ARepByImages(
  GroupWithGenerators( [ (1,2), (1,2,3) ] ),
  [ (1,2), () ],
  3, # degree
  "hom"
)
```

1.74 ARepByHom

`ARepByHom(` *hom* `)`

`ARepByHom(` *hom*, *d* `)`
`ARepByHom(` *hom*, *d*, *char* `)`
`ARepByHom(` *hom*, *d*, *char* `)`

Let *hom* be a homomorphism of a group into a mongroup. `ARepByHom` returns an arep of type `"mon"` corresponding to *hom*.

Let *hom* be a homomorphism of a group into a matrix group. `ARepByHom` returns an arep of type `"mat"` corresponding to *hom*.

Let *hom* be a homomorphism of a group into a permutation group and d a positive integer. `ARepByHom` returns an arep of type `"perm"` and degree d corresponding to *hom*. If d is smaller than the largest moved point of *hom*`.range` an error is signaled. As optional parameter a characteristic *char* or a *field* can be supplied to obtain an arep of arbitrary characteristic.

```
gap> G := GroupWithGenerators(SymmetricGroup(4));
Group( (1,4), (2,4), (3,4) )
gap> phi := IdentityMapping(G);
IdentityMapping( Group( (1,4), (2,4), (3,4) ) )
gap> ARepByHom(phi, 4);
NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 )
gap> H := GroupWithGenerators( [ Mon( [-1] ) ] );
Group( Mon( [ -1 ] ) )
gap> psi :=
> GroupHomomorphismByImages(G, H, G.generators, [H.1, H.1, H.1]);
GroupHomomorphismByImages(
  Group( (1,4), (2,4), (3,4) ),
  Group( Mon( [ -1 ] ) ),
```

```
    [ (1,4), (2,4), (3,4) ],
    [ Mon( [ -1 ] ), Mon( [ -1 ] ), Mon( [ -1 ] ) ] )
  gap> ARepByHom(psi);
  ARepByImages(
    GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ),
    [ Mon( [ -1 ] ),
      Mon( [ -1 ] ),
      Mon( [ -1 ] )
    ],
    "hom"
  )
```

1.75 ARepByCharacter

`ARepByCharacter(` *chi* `)`

Let *chi* be a onedimensional character of a group. `ARepByCharacter` returns a onedimensional arep of type `"mon"` given by *chi*.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> L := Irr(G);
[ Character( Group( (1,2) ), [ 1, 1 ] ),
  Character( Group( (1,2) ), [ 1, -1 ] ) ]
gap> ARepByCharacter( L[2] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ Mon( [ -1 ] ) ],
  "hom"
)
```

1.76 ConjugateARep

`ConjugateARep(` R`,` A `)` or R `^` A
`ConjugateARep(` R`,` A`,` *hint* `)`

Let R be an arep and A an amat (see 1.22). `ConjugateARep` returns an arep of type `"conjugate"` representing the conjugated representation $R^A: \ x \mapsto A^{-1} \cdot R(x) \cdot A$. The amat is tested for invertibility if the optional *hint* `"invertible"` is not supplied. R and A must be compatible in size and characteristic otherwise an error is signaled. Note that R and A can be accessed in the fields `.rep` and `.conjugation` of the result.

```
gap> G := GroupWithGenerators(SymmetricGroup(4));
Group( (1,4), (2,4), (3,4) )
gap> R := NaturalARep(G, 4);
NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 )
gap> A := AMatPerm( (1,2,3,4), 4 );
AMatPerm((1,2,3,4), 4)
gap> R ^ A;
ConjugateARep(
```

```
    NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 ),
    AMatPerm((1,2,3,4), 4)
  )
```

1.77 DirectSumARep

`DirectSumARep(` R_1`, ...,` R_k `)`

`DirectSumARep` returns an arep of type `"directSum"` representing the direct sum $R_1 \oplus \ldots \oplus R_k$ of the areps R_1, ..., R_k, which must have common source and characteristic otherwise an error is signaled.

The direct sum $R = R_1 \oplus \ldots \oplus R_k$ of representations is defined as $x \mapsto R_1(x) \oplus \ldots \oplus R_k(x)$.

Note that the summands R_1, ..., R_k can be accessed in the field `.summands` of the result.

```
gap> G := GroupWithGenerators( [(1,2,3,4), (1,3)] );
Group( (1,2,3,4), (1,3) )
gap> R1 := RegularARep(G);
RegularARep( GroupWithGenerators( [ (1,2,3,4), (1,3) ] ) )
gap> R2 := ARepByImages(G, [ [[1]], [[-1]] ]);
ARepByImages(
  GroupWithGenerators( [ (1,2,3,4), (1,3) ] ),
  [ [ [ 1 ] ], [ [ -1 ] ] ],
  "hom"
)
gap> DirectSumARep(R1, R2);
DirectSumARep(
  RegularARep( GroupWithGenerators( [ (1,2,3,4), (1,3) ] ) ),
  ARepByImages(
    GroupWithGenerators( [ (1,2,3,4), (1,3) ] ),
    [ [ [ 1 ] ], [ [ -1 ] ] ],
    "hom"
  )
)
```

`DirectSumARep(` *list* `)`

`DirectSumARep` returns an arep of type `"directSum"` representing the direct sum of the areps in *list* (see above).

1.78 InnerTensorProductARep

`InnerTensorProductARep(` R_1`, ...,` R_k `)`

`InnerTensorProductARep` returns an arep of type `"innerTensorProduct"` representing the inner tensor product $R = R_1 \otimes \ldots \otimes R_k$ of the areps R_1, ..., R_k, which must have common source and characteristic otherwise an error is signaled.

The inner tensor product $R = R_1 \otimes \ldots \otimes R_k$ of representations is defined as $x \mapsto R_1(x) \otimes \ldots \otimes R_k(x)$. Note that the inner tensor product yields a representation of the same source (in contrast to the outer tensor product, see 1.79).

Note that the tensor factors R_1, ..., R_k can be accessed in the field `.factors` of the result.

```
gap> G := GroupWithGenerators( [ (1,2), (3,4) ] );
Group( (1,2), (3,4) )
gap> R1 := ARepByImages(G, [ Mon( (1,2), 2 ), Mon( [-1, -1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2), (3,4) ] ),
  [ Mon( (1,2), 2 ), Mon( [ -1, -1 ] ) ],
  "hom"
)
gap> R2 := NaturalARep(G, 5);
NaturalARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 5 )
gap> InnerTensorProductARep(R1, R2);
InnerTensorProductARep(
  ARepByImages(
    GroupWithGenerators( [ (1,2), (3,4) ] ),
    [ Mon( (1,2), 2 ), Mon( [ -1, -1 ] ) ],
    "hom"
  ),
  NaturalARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 5 )
)
```

`InnerTensorProductARep(` *list* `)`

`InnerTensorProductARep` returns an arep of type `"innerTensorProduct"` representing the inner tensor product of the areps in *list* (see above).

1.79 OuterTensorProductARep

`OuterTensorProductARep(` R_1`, ...,` R_k `)`
`OuterTensorProductARep(` G`,` R_1`, ...,` R_k `)`

`OuterTensorProductARep` returns an arep of type `"outerTensorProduct"` representing the outer tensor product $R = R_1 \# \dots \# R_k$ of the areps R_1, ..., R_k, which must have common characteristic otherwise an error is signaled.

The outer tensor product $R = R_1 \# \dots \# R_k$ of representations is defined as $x \mapsto R_1(x) \otimes \dots \otimes R_k(x)$. Note that the outer tensor product of representations is a representation of the direct product of the sources (in contrast to the inner tensor product, see 1.78).

Using the first version `OuterTensorProductARep` returns an arep R with R`.source = DirectProduct(`R_1`.source, ...,` R_k`.source)` using the GAP function `DirectProduct`. In the second version the returned arep has as source the group G which must be the inner direct product $G = R_1$`.source` $\times \dots \times$ R_k`.source`. This property is not checked.

Note that the tensor factors R_1, ..., R_k can be accessed in the field `.factors` of the result.

```
gap> G1 := GroupWithGenerators(DihedralGroup(8));
Group( (1,2,3,4), (2,4) )
gap> G2 := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R1 := NaturalARep(G1, 4);
NaturalARep( GroupWithGenerators( [ (1,2,3,4), (2,4) ] ), 4 )
gap> R2 := ARepByImages(G2, [ [[-1]] ]);
```

```
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ [ [ -1 ] ] ],
  "hom"
)
gap> OuterTensorProductARep(R1, R2);
OuterTensorProductARep(
  NaturalARep( GroupWithGenerators( [ (1,2,3,4), (2,4) ] ), 4 ),
  ARepByImages(
    GroupWithGenerators( [ (1,2) ] ),
    [ [ [ -1 ] ] ],
    "hom"
  )
)
```

1.80 RestrictionARep

`RestrictionARep(` R`,` H `)`

`RestrictionARep` returns an arep of type `"restriction"` representing the restriction of the arep R to the subgroup H of R`.source`. Here, "subgroup" means, that all elements of H are contained in R`.source`.

The restriction $R \downarrow H$ of a representation R to a subgroup H is defined by $x \mapsto R(x),\ x \in H$.

Note that R can be accessed in the field `.rep` of the result.

```
gap> G := GroupWithGenerators(SymmetricGroup(4));
Group( (1,4), (2,4), (3,4) )
gap> H := GroupWithGenerators(AlternatingGroup(4));
Group( (1,2,4), (2,3,4) )
gap> R := NaturalARep(G, 4);
NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 )
gap> RestrictionARep(R, H);
RestrictionARep(
  NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 ),
  GroupWithGenerators( [ (1,2,4), (2,3,4) ] )
)
```

1.81 InductionARep

`InductionARep(` R`,` G `)`
`InductionARep(` R`,` G`,` T `)`

`InductionARep` returns an arep of type `"induction"` representing the induction of the arep R to the supergroup G with the transversal T of the residue classes R`.source`$\backslash G$. Here, "supergroup" means that all elements of R`.source` are contained in G. If no transversal T is supplied one is chosen by the function `RightTransversal`. If a transversal T is given it is not checked to be one.

The induction $R \uparrow_T G$ of a representation R of H to a supergroup G with transversal $T = \{t_1, \ldots, t_k\}$ of $H\backslash G$ is defined by $x \mapsto \left[\dot{R}\left(t_i \cdot x \cdot t_j^{-1}\right) \mid i, j \in \{1, \ldots, k\}\right]$, where $\dot{R}(y) = R(y)$ for $y \in H$ and 0 else.

Note that R and T can be accessed in the fields `.rep` and `.transversal` resp. of the result.

```
gap> G := GroupWithGenerators( [ (1,2,3,4), (1,2) ] );
Group( (1,2,3,4), (1,2) )
gap> H := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(H, [ [[Z(2), Z(2)], [0*Z(2), Z(2)]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ]
  ],
  "hom"
)
gap> R.name := "R";
"R"
gap> InductionARep(R, G);
InductionARep(
  R,
  GroupWithGenerators( [ (1,2,3,4), (1,2) ] ),
  [ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,4,3),
  (1,4), (1,4,2,3), (1,4)(2,3), (1,2,3), (1,2,3,4) ]
)
```

1.82 ExtensionARep

`ExtensionARep(` R, *chi* `)`

Let R be an irreducible arep of characteristic zero and *chi* a character of a supergroup of R`.source` which extends the character of R. `ExtensionARep` returns an arep of type `"extension"` representing an extension of R to *chi*`.source`. Here, "supergroup" means that all elements of R`.source` are contained in G. The extension is evaluated using Minkwitz's formula (see [Min96]).

Note that R and *chi* can be accessed in the fields `.rep` and `.character` of the result.

```
gap> G := GroupWithGenerators( [ (1,2,3,4), (1,2) ] );
Group( (1,2,3,4), (1,2) )
gap> H := GroupWithGenerators(AlternatingGroup(4));
Group( (1,2,4), (2,3,4) )
gap> G.name := "S4";
"S4"
gap> H.name := "A4";
"A4"
gap> R := ARepByImages(H, [ Mon( (1,2,3), [ 1, -1, -1 ] ),
> Mon( (1,2,3), 3 ) ] );
ARepByImages(
  A4,
  [ Mon( (1,2,3), [ 1, -1, -1 ] ),
    Mon( (1,2,3), 3 )
  ],
  "hom"
```

```
)
gap> L := Irr(G);
[ Character( Group( (1,2,3,4), (1,2) ), [ 1, 1, 1, 1, 1 ] ),
  Character( Group( (1,2,3,4), (1,2) ), [ 1, -1, 1, 1, -1 ] ),
  Character( Group( (1,2,3,4), (1,2) ), [ 2, 0, -1, 2, 0 ] ),
  Character( Group( (1,2,3,4), (1,2) ), [ 3, -1, 0, -1, 1 ] ),
  Character( Group( (1,2,3,4), (1,2) ), [ 3, 1, 0, -1, -1 ] ) ]
gap> ExtensionARep(R, L[4]);
ExtensionARep(
  ARepByImages(
    A4,
    [ Mon(
        (1,2,3),
        [ 1, -1, -1 ]
      ),
      Mon( (1,2,3), 3 )
    ],
    "hom"
  ),
  Character( Group( (1,2,3,4), (1,2) ), [ 3, -1, 0, -1, 1 ] )
)
```

1.83 GaloisConjugateARep

`GaloisConjugateARep(` R, *aut* `)`
`GaloisConjugateARep(` R, k `)`

`GaloisConjugateARep` returns an arep of type `"galoisConjugate"` representing the Galois conjugate of the arep A. The conjugating automorphism may either be a field automorphism *aut* or an integer k specifying the automorphism `x -> GaloisCyc(x,` k`)` in the case characteristic $= 0$ or `x -> x^(FrobeniusAut^`k`)` in the case characteristic $= p$ prime.

The Galois conjugate of a representation R with a field automorphism *aut* is defined by $x \mapsto R(x)^{aut}$.

Note that R and *aut* can be accessed in the fields `.rep` and `.galoisAut` resp. of the result.

```
gap> G := GroupWithGenerators( [ (1,2,3) ] );
Group( (1,2,3) )
gap> R := ARepByImages(G, [ [[E(3)]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2,3) ] ),
  [ [ [ E(3) ] ]
  ],
  "hom"
)
gap> GaloisConjugateARep(R, -1);
GaloisConjugateARep(
  ARepByImages(
    GroupWithGenerators( [ (1,2,3) ] ),
    [ [ [ E(3) ] ]
```

```
      ],
      "hom"
    ),
    -1
  )
```

1.84 Basic Functions for AReps

The following sections describe basic functions for areps like e.g. testing irreducibility and equivalence, evaluating an arep at a group element, computing kernel and character, and constructing an arep with given character.

1.85 Comparison of AReps

R_1 = R_2
R_1 <> R_2

The equality operator = evaluates to `true` if the areps R_1 and R_2 are equal and to `false` otherwise. The inequality operator <> evaluates to `true` if the amats R_1 and R_2 are not equal and to `false` otherwise.

Two areps are equal iff they define the same representation. This means that first the sources have to be equal, i.e. R_1`.source` = R_2`.source` and second the images are pointwise equal.

R_1 < R_2
R_1 <= R_2
R_1 >= R_2
R_1 > R_2

The operators <, <=, >=, and > evaluate to `true` if the arep R_1 is strictly less than, less than or equal to, greater than or equal to, and strictly greater than the arep R_2.

The ordering of areps is defined via the ordering of records.

1.86 ImageARep

`ImageARep(` x, R `)` or x ^ R

Let R be an arep and x a group element of R`.source`. `ImageARep` returns the image of x under R as an amat (see 1.22). For conversion of amats see 1.48 – 1.50.

```
gap> G := GroupWithGenerators(SolvableGroup(8, 5));
Q8
gap> R := RegularARep(G);
RegularARep( Q8 )
gap> x := Random(G);
c
gap> ImageARep(x, R);
TensorProductAMat(
  AMatPerm((1,2)(3,4)(5,6)(7,8), 8),
  IdentityPermAMat(1)
```

```
) *
DirectSumAMat(
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1)
)
gap> PermAMat(last);
(1,2)(3,4)(5,6)(7,8)
```

`ImageARep(` *list*`,` R `)`

`ImageARep` returns the list of images of the group elements in *list* under the arep R (see above). The images are amats (see 1.22). For conversion of amats see 1.48 – 1.50.

1.87 IsEquivalentARep

`IsEquivalentARep(` R_1`,` R_2 `)`

Let R_1 and R_2 be two areps with Maschke condition, i.e. `Size(` R_i`.source ) mod` R_i`.char` $\neq 0$, $i = 1, 2$. `IsEquivalentARep` returns `true` if the areps R_1 and R_2 define equivalent representations and `false` otherwise. Two representations (with Maschke condition) are equivalent iff they have the same character. R_1 and R_2 must have identical source (i.e. `IsIdentical(`R_1`,` R_2`) = true`) and characteristic otherwise an error is signaled.

```
gap> G := GroupWithGenerators( [ (1,2,3) ] );
Group( (1,2,3) )
gap> R1 := NaturalARep(G, 3);
NaturalARep( GroupWithGenerators( [ (1,2,3) ] ), 3 )
gap> R2 := RegularARep(G);
RegularARep( GroupWithGenerators( [ (1,2,3) ] ) )
gap> IsEquivalentARep(R1, R2);
true
```

1.88 CharacterARep

`CharacterARep(` R `)`

`CharacterARep` returns the character of the arep R. Since GAP only provides characters of characteristic zero, `CharacterARep` only works in this case and will signal an error otherwise. Note that `CharacterARep` sets and tests R`.character`.

```
gap> G := GroupWithGenerators( [ (1,2), (3,4) ] );
Group( (1,2), (3,4) )
gap> CharacterARep(RegularARep(G));
Character( Group( (1,2), (3,4) ), [ 4, 0, 0, 0 ] )
```

1.89 IsIrreducibleARep

`IsIrreducibleARep(` R `)`

Let R an arep of characteristic zero. `IsIrreducibleARep` returns `true` if R represents an irreducible arep and `false` otherwise. To determine irreducibility the character is used, which is the reason for the condition characteristic $= 0$ (see 1.88). Note that `IsIrreducibleARep` sets and tests R`.isIrreducible`.

```
gap> G := GroupWithGenerators(SolvableGroup(12, 5));
A4
gap> L := Irr(G);
[ Character( A4, [ 1, 1, 1, 1 ] ),
  Character( A4, [ 1, 1, E(3), E(3)^2 ] ),
  Character( A4, [ 1, 1, E(3)^2, E(3) ] ),
  Character( A4, [ 3, -1, 0, 0 ] ) ]
gap> R := ARepByCharacter(L[2]);
ARepByImages(
  A4,
  [ Mon( [ E(3) ] ),
    Mon( (), 1 ),
    Mon( (), 1 )
  ],
  "hom"
)
gap> IsIrreducibleARep(R);
true
gap> IsIrreducibleARep(RegularARep(G));
false
```

1.90 KernelARep

`KernelARep(` R `)`

`KernelARep` returns the kernel of the arep R. Note that `KernelARep` sets and tests R`.kernel`.

```
gap> G := GroupWithGenerators(SymmetricGroup(3));
Group( (1,3), (2,3) )
gap> R := ARepByImages(G, [ [[-1]], [[-1]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,3), (2,3) ] ),
  [ [ [ -1 ] ],
    [ [ -1 ] ]
  ],
  "hom"
)
gap> KernelARep(R);
Subgroup( Group( (1,3), (2,3) ), [ (1,3,2) ] )
```

1.91 IsFaithfulARep

`IsFaithfulARep(` *R* `)`

`IsFaithfulARep` returns `true` if the arep R represents a faithful representation and `false` otherwise. Note that `IsFaithfulARep` sets and tests R`.isFaithful`.

```
gap> G := GroupWithGenerators(SolvableGroup(16, 7));
Q8x2
gap> IsFaithfulARep(TrivialPermARep(G));
false
gap> IsFaithfulARep(RegularARep(G));
true
```

1.92 ARepWithCharacter

`ARepWithCharacter(` *chi* `)`

`ARepWithCharacter` constructs an arep with character *chi*. The group *chi*`.source` must be solvable otherwise an error is signaled. Note that the function returns a monomial arep if this is possible.

Attention: `ARepWithCharacter` only works in GAP 3.4.4 after bugfix 9!

```
gap> G := GroupWithGenerators(SolvableGroup(8, 5));
Q8
gap> L := Irr(G);
[ Character( Q8, [ 1, 1, 1, 1, 1 ] ),
  Character( Q8, [ 1, 1, -1, 1, -1 ] ),
  Character( Q8, [ 1, 1, 1, -1, -1 ] ),
  Character( Q8, [ 1, 1, -1, -1, 1 ] ),
  Character( Q8, [ 2, -2, 0, 0, 0 ] ) ]
gap> MonARepARep(ARepWithCharacter(L[5]));
ARepByImages(
  Q8,
  [ Mon(
      (1,2),
      [ -1, 1 ]
    ),
    Mon( [ E(4), -E(4) ] ),
    Mon( [ -1, -1 ] )
  ],
  "hom"
)
```

1.93 GeneralFourierTransform

`GeneralFourierTransform(` *G* `)`

`GeneralFourierTransform` returns an amat representing a Fourier transform over the complex numbers for the solvable group G. For an explanation of Fourier transforms see [CB93].

In order to obtain a *fast* Fourier transform for G apply the function `DecompositionMonRep` to any regular representation of G.

Attention: `GeneralFourierTransform` only works in GAP 3.4.4 after bugfix 9!

```
gap> G := SymmetricGroup(3);
Group( (1,3), (2,3) )
gap> GeneralFourierTransform(G);
AMatMat(
  [ [ 1, 1, 1, 1, 1, 1 ], [ 1, -1, -1, 1, 1, -1 ],
  [ 1, 0, 0, E(3), E(3)^2, 0 ], [ 0, 1, E(3)^2, 0, 0, E(3) ],
  [ 0, 1, E(3), 0, 0, E(3)^2 ], [ 1, 0, 0, E(3)^2, E(3), 0 ] ],
  "invertible"
) ^ -1
```

1.94 Converting AReps

The following sections describe functions for convertibility and conversion of arbitrary areps to areps of type `"perm"`, `"mon"`, and `"mat"`. As in **AMat** (see 1.22) the naming of the functions follows the usual GAP-convention: `ChalkCheese` makes chalk from cheese. The parts in the name (chalk, cheese) are:

ARep	– an arep of any type
PermARep	– an arep of type “perm”
MonARep	– an arep of type “mon”
MatARep	– an arep of type “mat”

1.95 IsPermRep

`IsPermRep(` R `)`

`IsPermRep` returns `true` if R represents a permutation representation and `false` otherwise. Note that the name of this function is not `IsPermARep` since R can be an arep of any type but represents a permutation representation in the mathematical sense (every image is a permutation matrix). Note that `IsPermRep` sets and tests R`.isPermRep`.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ Mon( [1, -1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ Mon( [ 1, -1 ] )
  ],
  "hom"
)
gap> IsPermRep(ConjugateARep(R, DFTAMat(2)));
true
```

1.96 IsMonRep

`IsMonRep(` R `)`

`IsMonRep` returns `true` if R represents a monomial representation and `false` otherwise. Note that the name of this function is not `IsMonARep` since R can be an arep of any type but represents a monomial representation in the mathematical sense (every image is a monomial matrix). Note that `IsMonRep` sets and tests R`.isMonRep`.

```
gap> G := GroupWithGenerators(SolvableGroup(8, 5));
Q8
gap> R := RegularARep(G);
RegularARep( Q8 )
gap> IsMonRep(InnerTensorProductARep(R, R));
true
```

1.97 PermARepARep

`PermARepARep(` R `)`

`PermARepARep` returns an arep of type `"perm"` representing the same representation as the arep R if possible. Otherwise `false` is returned. Note that `PermARepARep` sets and tests R`.permARep`.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ Mon( [1, -1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ Mon( [ 1, -1 ] )
  ],
  "hom"
)
gap> PermARepARep(ConjugateARep(R, DFTAMat(2)));
NaturalARep( GroupWithGenerators( [ (1,2) ] ), 2 )
gap> PermARepARep(R);
false
```

1.98 MonARepARep

`MonARepARep(` R `)`

`MonARepARep` returns an arep of type `"mon"` representing the same representation as the arep R if possible. Otherwise `false` is returned. Note that `MonARepARep` sets and tests R`.monARep`.

```
gap> G := GroupWithGenerators( [ (1,2,3), (1,2) ] );
Group( (1,2,3), (1,2) )
gap> R1 := ARepByImages(G, [ [[1]], [[-1]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2,3), (1,2) ] ),
  [ [ [ 1 ] ],
```

```
    [ [ -1 ] ]
  ],
  "hom"
)
gap> R2 := NaturalARep(G, 4);
NaturalARep( GroupWithGenerators( [ (1,2,3), (1,2) ] ), 4 )
gap> MonARepARep(InnerTensorProductARep(R1, R2));
ARepByImages(
  GroupWithGenerators( [ (1,2,3), (1,2) ] ),
  [ Mon( (1,2,3), 4 ),
    Mon(
      (1,2),
      [ -1, -1, -1, -1 ]
    )
  ],
  "hom"
)
```

1.99 MatARepARep

`MatARepARep(` *R* `)`

`MatARepARep` returns an arep of type `"mat"` representing the same representation as the arep *R*. Note that `MatARepARep` sets and tests *R*`.matARep`.

```
gap> G := GroupWithGenerators( [ (1,2), (3,4) ] );
Group( (1,2), (3,4) )
gap> MatARepARep(RegularARep(G, 3));
ARepByImages(
  GroupWithGenerators( [ (1,2), (3,4) ] ),
  [ [ [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ],
      [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ] ],
    [ [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
      [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ],
      [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ] ]
  ],
  "hom"
)
```

1.100 Higher Functions for AReps

The following sections describe functions allowing the structural manipulation of, mainly monomial, areps. The idea is to convert a given arep into a mathematical equal (not only equivalent!) arep having more structure. Examples are: converting a transitive monomial arep into a conjugated induction (see 1.111), converting an induction into a conjugated double induction (see 1.112), changing the transversal of an induction (see 1.115), decomposing a transitive monomial arep into a conjugated outer tensor product (see 1.116) and last but not least decomposing a monomial arep into a conjugated sum of irreducibles (see 1.123). The latter is one of the most interesting functions of the package AREP.

1.101 IsRestrictedCharacter

`IsRestrictedCharacter(` *chi*`,` *chisub* `)`

`IsRestrictedCharacter` returns `true` if the character *chisub* is a restriction of the character *chi* to *chisub*`.source` and `false` otherwise. All elements of *chisub*`.source` must be contained in *chi*`.source` otherwise an error is signaled.

```
gap> G := SymmetricGroup(3); G.name := "S3";
Group( (1,3), (2,3) )
"S3"
gap> H := CyclicGroup(3); H.name := "Z3";
Group( (1,2,3) )
"Z3"
gap> L1 := Irr(G);
[ Character( S3, [ 1, 1, 1 ] ), Character( S3, [ 1, -1, 1 ] ),
  Character( S3, [ 2, 0, -1 ] ) ]
gap> L2 := Irr(H);
[ Character( Z3, [ 1, 1, 1 ] ), Character( Z3, [ 1, E(3), E(3)^2 ] ),
  Character( Z3, [ 1, E(3)^2, E(3) ] ) ]
gap> IsRestrictedCharacter(L1[2], L2[1]);
true
```

1.102 AllExtendingCharacters

`AllExtendingCharacters(` *chi*`,` G `)`

`AllExtendingCharacters` returns the list of all characters of G extending *chi*. All elements of *chi*`.source` must be contained in G otherwise an error is signaled.

```
gap> H := AlternatingGroup(4); H.name := "A4";
Group( (1,2,4), (2,3,4) )
"A4"
gap> G := SymmetricGroup(4); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> L := Irr(H);
[ Character( A4, [ 1, 1, 1, 1 ] ),
  Character( A4, [ 1, 1, E(3)^2, E(3) ] ),
  Character( A4, [ 1, 1, E(3), E(3)^2 ] ),
  Character( A4, [ 3, -1, 0, 0 ] ) ]
gap> AllExtendingCharacters(L[4], G);
[ Character( S4, [ 3, -1, -1, 0, 1 ] ),
  Character( S4, [ 3, 1, -1, 0, -1 ] ) ]
```

1.103 OneExtendingCharacter

`OneExtendingCharacter(` *chi*`,` G `)`

`OneExtendingCharacter` returns one character of G extending *chi* if possible or returns false otherwise. All elements of *chi*`.source` must be contained in G otherwise an error is signaled.

```
gap> H := Group( (1,3)(2,4) ); H.name := "Z2";
Group( (1,3)(2,4) )
"Z2"
gap> G := Group( (1,2,3,4) ); G.name := "Z4";
Group( (1,2,3,4) )
"Z4"
gap> L := Irr(H);
[ Character( Z2, [ 1, 1 ] ), Character( Z2, [ 1, -1 ] ) ]
gap> OneExtendingCharacter(L[2], G);
Character( Z4, [ 1, E(4), -1, -E(4) ] )
```

1.104 IntertwiningSpaceARep

`IntertwiningSpaceARep(` R_1`,` R_2 `)`

`IntertwiningSpaceARep` returns a list of amats (see 1.22) representing a base of the intertwining space $\mathrm{Int}(R_1, R_2)$ of the areps R_1 and R_2, which must have common source and characteristic otherwise an error is signaled.

The intertwining space $\mathrm{Int}(R_1, R_2)$ of two representations R_1 and R_2 of a group G of the same characteristic is the vector space of matrices $\{M \mid R_1(x) \cdot M = M \cdot R_2(x), \text{ for all } x \in G\}$.

```
gap> G := GroupWithGenerators( [ (1,2,3) ] );
Group( (1,2,3) )
gap> R1 := NaturalARep(G, 3);
NaturalARep( GroupWithGenerators( [ (1,2,3) ] ), 3 )
gap> R2 := ARepByImages(G, [ Mon( [1, E(3), E(3)^2] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2,3) ] ),
  [ Mon( [ 1, E(3), E(3)^2 ] )
  ],
  "hom"
)
gap> IntertwiningSpaceARep(R1, R2);
[ AMatMat( [ [ 1, 0, 0 ], [ 1, 0, 0 ], [ 1, 0, 0 ] ] ),
  AMatMat( [ [ 0, 1, 0 ], [ 0, E(3), 0 ], [ 0, E(3)^2, 0 ] ] ),
  AMatMat( [ [ 0, 0, 1 ], [ 0, 0, E(3)^2 ], [ 0, 0, E(3) ] ] ) ]
```

1.105 IntertwiningNumberARep

`IntertwiningNumberARep(` R_1`,` R_2 `)`

`IntertwiningNumberARep` returns the intertwining number of the areps R_1 and R_2. The Maschke condition must hold for both R_1 and R_2, otherwise an error is signaled. R_1 and R_2 must have identical source (i.e. IsIdentical(R_1, R_2) = `true`) and characteristic otherwise an error is signaled.

The intertwining number of two representations R_1 and R_2 (with Maschke condition) is the dimension of the intertwining space or the scalar product of the characters.

```
gap> G := GroupWithGenerators(SolvableGroup(64, 12));
2^3xD8
gap> R := RegularARep(G);
RegularARep( 2^3xD8 )
gap> IntertwiningNumberARep(R, R);
64
```

1.106 UnderlyingPermRep

`UnderlyingPermRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `UnderlyingPermRep` returns an arep of type `"perm"` representing the underlying permutation representation of R.

The underlying permutation representation of a monomial representation R is obtained by replacing all entries $\neq 0$ in the images $R(x)$, $x \in G$ by 1.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ [[0, 2], [1/2, 0]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ [ [ 0, 2 ], [ 1/2, 0 ] ]
  ],
  "hom"
)
gap> UnderlyingPermARep(R);
NaturalARep( GroupWithGenerators( [ (1,2) ] ), 2 )
```

1.107 IsTransitiveMonRep

`IsTransitiveMonRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `IsTransitiveMonRep` returns `true` if R is transitive and `false` otherwise. Note that `IsTransitiveMonRep` sets and tests R`.isTransitive`.

A monomial representation is transitive iff the underlying permutation representation is.

```
gap> G := GroupWithGenerators( [ (1,2), (3,4) ] );
Group( (1,2), (3,4) )
gap> IsTransitiveMonRep(NaturalARep(G, 4));
false
gap> IsTransitiveMonRep(RegularARep(G));
true
```

1.108 IsPrimitiveMonRep

`IsPrimitiveMonRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `IsPrimitiveMonRep` returns `true` if R is primitive and `false` otherwise.

A monomial representation is primitive iff the underlying permutation representation is.

```
gap> G := GroupWithGenerators(SymmetricGroup(4)); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> H := GroupWithGenerators(SymmetricGroup(3)); H.name := "S3";
Group( (1,3), (2,3) )
"S3"
gap> L := Irr(H);
[ Character( S3, [ 1, 1, 1 ] ), Character( S3, [ 1, -1, 1 ] ),
  Character( S3, [ 2, 0, -1 ] ) ]
gap> R := ARepByCharacter(L[2]);
ARepByImages(
  S3,
  [ Mon( [ -1 ] ),
    Mon( [ -1 ] )
  ],
  "hom"
)
gap> IsPrimitiveMonRep(InductionARep(R, G));
true
```

1.109 TransitivityDegreeMonRep

`TransitivityDegreeMonRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `TransitivityDegreeMonRep` returns the degree of transitivity of R as an integer. Note that `TransitivityDegreeMonRep` sets and tests R`.transitivity`.

The degree of transitivity of a monomial representation is defined as the degree of transitivity of the underlying permutation representation.

```
gap> G := GroupWithGenerators(AlternatingGroup(5));
Group( (1,2,5), (2,3,5), (3,4,5) )
gap> TransitivityDegreeMonRep(NaturalARep(G, 5));
3
```

1.110 OrbitDecompositionMonRep

`OrbitDecompositionMonRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `OrbitDecompositionMonRep` returns an arep equal to R with structure $(R_1 \oplus \ldots \oplus R_k)^P$ where $R_i,\ i = 1, \ldots, k$ are transitive areps of type `"mon"` and P is an amat of type `"perm"` (for amats see 1.22).

```
gap> G := GroupWithGenerators( [ (1,2,3,4) ] ); G.name := "Z4";
Group( (1,2,3,4) )
"Z4"
```

```
gap> R := ARepByImages(G, [ Mon( (1,2)(3,4), [1,-1,1,1,-1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2,3,4) ] ),
  [ Mon( (1,2)(3,4), [ 1, -1, 1, 1, -1 ] ) ],
  "hom"
)
gap> OrbitDecompositionMonRep(R);
ConjugateARep(
  DirectSumARep(
    ARepByImages(
      Z4,
      [ Mon( (1,2), [ 1, -1 ] ) ],
      "hom"
    ),
    ARepByImages(
      Z4,
      [ Mon( (1,2), 2 ) ],
      "hom"
    ),
    ARepByImages(
      Z4,
      [ Mon( [ -1 ] ) ],
      "hom"
    )
  ),
  IdentityPermAMat(5)
)
```

1.111 TransitiveToInductionMonRep

`TransitiveToInductionMonRep(` R `)`
`TransitiveToInductionMonRep(` R, i `)`

Let R be a transitive monomial arep of a group G. `TransitiveToInductionMonRep` returns an arep equal to R with structure $R = (L \uparrow_T G)^D$. L is an arep of degree one of the stabilizer H of the point i and T a transversal of $H\backslash G$. The default for i is R`.degree`. D is a diagonal amat (see 1.22) of type `"mon"`. Note that `TransitiveToInductionMonRep` sets and tests the field R`.induction` if $i = R$`.degree`.

```
gap> G := GroupWithGenerators(DihedralGroup(8));
Group( (1,2,3,4), (2,4) )
gap> R := ARepByImages(G, [ Mon( [E(4), E(4)^-1] ), Mon( (1,2), 2 ) ]);
ARepByImages(
  GroupWithGenerators( [ (1,2,3,4), (2,4) ] ),
  [ Mon( [ E(4), -E(4) ] ), Mon( (1,2), 2 ) ],
  "hom"
)
gap> TransitiveToInductionMonRep(R);
ConjugateARep(
```

```
    InductionARep(
      ARepByImages(
        GroupWithGenerators( [ (1,2,3,4) ] ),
        [ Mon( [ -E(4) ] ) ],
        "hom"
      ),
      GroupWithGenerators( [ (1,2,3,4), (2,4) ] ),
      [ (2,4), () ]
    ),
    IdentityMonAMat(2)
  )
```

1.112 InsertedInductionARep

`InsertedInductionARep(` R, H `)`

Let R be an arep of type `"induction"`, i.e. $R = L \uparrow_T G$ where L is an arep of $U \leq G$ and $U \leq H \leq G$. `InsertedInductionARep` returns an arep equal to R with structure $((L \uparrow_{T_1} H) \uparrow_{T_2} G)^M$ where M is an amat (see 1.22) with a structure similar to R. If R`.rep` is of degree 1 then M is an amat of type `"mon"`.

```
gap> G := GroupWithGenerators(SymmetricGroup(4)); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> H := GroupWithGenerators(AlternatingGroup(4)); H.name := "A4";
Group( (1,2,4), (2,3,4) )
"A4"
gap> U := GroupWithGenerators(CyclicGroup(3)); U.name := "Z3";
Group( (1,2,3) )
"Z3"
gap> R := ARepByImages(U, [ [[E(3)]] ] );
ARepByImages(
  Z3,
  [ [ [ E(3) ] ]
  ],
  "hom"
)
gap> InsertedInductionARep(InductionARep(R, G), H);
ConjugateARep(
  InductionARep(
    InductionARep(
      ARepByImages(
        Z3,
        [ [ [ E(3) ] ] ],
        "hom"
      ),
      A4,
      [ (), (2,3,4), (2,4,3), (1,4)(2,3) ]
    ),
```

```
    S4,
    [ (), (3,4) ]
  ),
  AMatMon( Mon(
    (2,4,8,7,3,5),
    [ 1, 1, 1, 1, 1, 1, E(3)^2, 1 ]
  ) )
)
```

1.113 ConjugationPermReps

`ConjugationPermReps(` R_1`,` R_2 `)`

Let R_1 and R_2 be permutation representations (i.e. `IsPermRep(` R_i `) = true`, $i = 1, 2$). `ConjugationPermReps` returns an amat A (see 1.22) of type `"perm"` such that ${R_1}^A = R_2$. R_1 and R_2 must have common source and characteristic otherwise an error is signaled.

```
gap> G := GroupWithGenerators( [ (1,2,3) ] );
Group( (1,2,3) )
gap> R1 := NaturalARep(G, 3);
NaturalARep( GroupWithGenerators( [ (1,2,3) ] ), 3 )
gap> R2 := ARepByImages(G, [ (1,3,2) ], 3);
ARepByImages(
  GroupWithGenerators( [ (1,2,3) ] ),
  [ (1,3,2)
  ],
  3, # degree
  "hom"
)
gap> A := ConjugationPermReps(R1, R2);
AMatPerm((2,3), 3)
gap> R1 ^ A = R2;
true
```

1.114 ConjugationTransitiveMonReps

`ConjugationTransitiveMonReps(` R_1`,` R_2 `)`

Let R_1 and R_2 be transitive monomial representations. `ConjugationTransitiveMonReps` returns an amat A (see 1.22) of type `"mon"` such that ${R_1}^A = R_2$ if possible and `false` otherwise. R_1 and R_2 must have common source otherwise an error is signaled.

Note that a conjugating monomial matrix exists iff R_1 and R_2 are induced from inner conjugated representations of degree one (see [Püs98]).

```
gap> G := GroupWithGenerators( [ (1,2,3), (1,2) ] );
Group( (1,2,3), (1,2) )
gap> R1 := ARepByImages(G, [ Mon( [E(3), E(3)^2] ), Mon( (1,2), 2 ) ]);
ARepByImages(
  GroupWithGenerators( [ (1,2,3), (1,2) ] ),
  [ Mon( [ E(3), E(3)^2 ] ),
```

```
    Mon( (1,2), 2 )
  ],
  "hom"
)
gap> R2 := ARepByImages(G, [ Mon( [E(3)^2, E(3)] ), Mon( (1,2), 2 ) ]);
ARepByImages(
  GroupWithGenerators( [ (1,2,3), (1,2) ] ),
  [ Mon( [ E(3)^2, E(3) ] ),
    Mon( (1,2), 2 )
  ],
  "hom"
)
gap> ConjugationTransitiveMonReps(R1, R2);
AMatMon( Mon( (1,2), 2 ) )
```

1.115 TransversalChangeInductionARep

`TransversalChangeInductionARep(` *R*, *T* `)`
`TransversalChangeInductionARep(` *R*, *T*, *hint* `)`

Let R be an arep of type `"induction"`, i.e. $R = L \uparrow_S G$ and T another transversal of L`.source`$\backslash G$. `TransversalChangeInductionARep` returns an arep equal to R with structure $(L \uparrow_T G)^M$ where M is an amat (see 1.22). M is of type `"mon"` if L is of degree 1 else M has a structure similar to R. The *hint* `"isTransversal"` suppresses checking T to be a right transversal.

```
gap> G := GroupWithGenerators(SymmetricGroup(4)); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> H := GroupWithGenerators(SymmetricGroup(3)); H.name := "S3";
Group( (1,3), (2,3) )
"S3"
gap> R := ARepByImages(H, [ [[-1]], [[-1]] ], "hom" );
ARepByImages(
  S3,
  [ [ [ -1 ] ], [ [ -1 ] ] ],
  "hom"
)
gap> RG := InductionARep(R, G);
InductionARep(
  ARepByImages(
    S3,
    [ [ [ -1 ] ], [ [ -1 ] ] ],
    "hom"
  ),
  S4,
  [ (), (3,4), (2,4), (1,4) ]
)
gap> T := [(1,2,3,4), (2,3,4), (3,4), ()];;
```

```
gap> TransversalChangeInductionARep(RG, T);
ConjugateARep(
  InductionARep(
    ARepByImages(
      S3,
      [ [ [ -1 ] ], [ [ -1 ] ] ],
      "hom"
    ),
    S4,
    [ (1,2,3,4), (2,3,4), (3,4), () ]
  ),
  AMatMon( Mon( (1,4)(2,3), [ 1, 1, -1, 1 ] ) )
)
gap> last = RG;
true
```

1.116 OuterTensorProductDecompositionMonRep

`OuterTensorProductDecompositionMonRep(` R `)`

Let R be a transitive monomial arep. `OuterTensorProductDecompositionMonRep` returns an arep equal to R with structure $(R_1 \# \ldots \# R_k)^M$. The R_i are areps of type `"mon"`, M is an amat of type mon.

For a definition of the outer tensor product of representations see 1.79. For an explanation of the algorithm see [Püs98].

```
gap> G := GroupWithGenerators(SolvableGroup(48, 16));
2x4xS3
gap> R := RegularARep(G, 2);
RegularARep( 2x4xS3, GF(2) )
gap> OuterTensorProductDecompositionMonRep(R);
ConjugateARep(
  OuterTensorProductARep(
    2x4xS3,
    ARepByImages(
      GroupWithGenerators( [ c ] ),
      [ Mon( (1,2), 2, GF(2) ) ],
      "hom"
    ),
    ARepByImages(
      GroupWithGenerators( [ d, e ] ),
      [ Mon( (1,3,2,4), 4, GF(2) ),
        Mon( (1,2)(3,4), 4, GF(2) )
      ],
      "hom"
    ),
    ARepByImages(
      GroupWithGenerators( [ a*e, b ] ),
      [ Mon( (1,4)(2,6)(3,5), 6, GF(2) ),
```

```
      Mon( (1,2,3)(4,5,6), 6, GF(2) )
    ],
    "hom"
  )
 ),
 AMatMon( Mon( ( 2, 9,18,44,16,28,30,46,31, 6,42,48,47,39,23,35,37, 7)
( 3,17,36,45,24,43, 8,10,25, 5,34,29,38,15,19, 4,26,13)
(11,33,22,27,21,20,12,41,40,32,14), 48, GF(2) ) )
)
gap> last = R;
true
```

1.117 InnerConjugationARep

`InnerConjugationARep(` R, G, t `)`

Let R be an arep with source $H \leq G$ and $t \in G$. `InnerConjugationARep` returns an arep of type `"perm"` or `"mon"` or `"mat"`, the most specific possible, representing the inner conjugate R^t of R with t.

The inner conjugate R^t is a representation of H^t defined by $x \mapsto R(t \cdot x \cdot t^{-1})$.

```
gap> G := GroupWithGenerators(SymmetricGroup(4));
Group( (1,4), (2,4), (3,4) )
gap> H := GroupWithGenerators(SymmetricGroup(3));
Group( (1,3), (2,3) )
gap> R := NaturalARep(H, 3);
NaturalARep( GroupWithGenerators( [ (1,3), (2,3) ] ), 3 )
gap> InnerConjugationARep(R, G, (1,2,3,4));
ARepByImages(
  GroupWithGenerators( [ (2,4), (3,4) ] ),
  [ (1,3), (2,3) ],
  3, # degree
  "hom"
)
```

1.118 RestrictionInductionARep

`RestrictionInductionARep(` R, K `)`

Let R be an arep of type `"induction"`, i.e. $R = L \uparrow_T G$ where L is an arep of $H \leq G$ of degree 1 and $K \leq G$ a subgroup. `RestrictionInductionARep` returns an arep equal to the restriction $R \downarrow K$ with structure $\left(\bigoplus_{i=1}^{k}((L^{s_i} \downarrow (H^{s_i} \cap K)) \uparrow_{T_i} K)\right)^M$. $S = \{s_1, \ldots, s_k\}$ is a transversal of the double cosets $H \backslash G / K$, L^{s_i} denotes the inner conjugate of R with s_i, and M is an amat (see 1.22) of type `"mon"`.

Note that this decomposition is based on a refined version of Mackey's subgroup theorem (see [Püs98]).

```
gap> G := GroupWithGenerators(SymmetricGroup(4)); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
```

```
"S4"
gap> H := GroupWithGenerators( [ (1,2) ] ); H.name := "Z2";
Group( (1,2) )
"Z2"
gap> K := GroupWithGenerators( [ (1,2,3) ] ); K.name := "Z3";
Group( (1,2,3) )
"Z3"
gap> L := ARepByImages(H, [ Mon( [-1] ) ] );
ARepByImages(
  Z2,
  [ Mon( [ -1 ] )
  ],
  "hom"
)
gap> RestrictionInductionARep(InductionARep(L, G), K);
ConjugateARep(
  DirectSumARep(
    RegularARep( GroupWithGenerators( [ (1,2,3) ] ) ),
    RegularARep( GroupWithGenerators( [ (1,2,3) ] ) ),
    RegularARep( GroupWithGenerators( [ (1,2,3) ] ) ),
    RegularARep( GroupWithGenerators( [ (1,2,3) ] ) )
  ),
  AMatMon( Mon(
    ( 2,12, 4, 6, 9, 5, 8,10),
    [ 1, 1, -1, -1, 1, 1, -1, -1, -1, -1, 1, 1 ]
  ) )
)
```

1.119 kbsARep

`kbsARep(` R `)`

`kbsARep` returns the kbs (conjugated blockstructure) of the arep R. The kbs of a representation is a partition of the set $\{1, \ldots, R.\mathtt{degree}\}$ representing the blockstructure of R. For an exact definition see 1.169.

Note that for a monomial representation the kbs is exactly the list of orbits.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ (2,3) ], 4);
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ (2,3) ],
  4, # degree
  "hom"
)
gap> kbsARep(R);
[ [ 1 ], [ 2, 3 ], [ 4 ] ]
```

1.120 RestrictionToSubmoduleARep

`RestrictionToSubmoduleARep(` *R*, *list* `)`
`RestrictionToSubmoduleARep(` *R*, *list*, *hint* `)`

Let R be an arep and *list* a subset of `[1..`R`.degree]`. `RestrictionToSubmoduleARep` returns an arep of type `"perm"` or `"mon"` or `"mat"`, the most specific possible, representing the restriction of R to the submodule generated by the base vectors given through *list*. The optional *hint* `"hom"` avoids the check for homomorphism.

Note that the restriction to the submodule given by *list* defines a homomorphism iff *list* is a union of lists in the kbs of R (see 1.119).

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ (2,4) ], 4);
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ (2,4) ],
  4, # degree
  "hom"
)
gap> RestrictionToSubmoduleARep(R, [2,4]);
NaturalARep( GroupWithGenerators( [ (1,2) ] ), 2 )
```

1.121 kbsDecompositionARep

`kbsDecompositionARep(` *R* `)`

`kbsDecompositionARep` returns an arep equal to R with structure $(R_1 \oplus \ldots \oplus R_k)^P$ where P is an amat (see 1.22) of type `"perm"`and all R_i have trivial kbs (see 1.119).

Note that for a monomial arep `kbsDecompositionARep` performs exactly the same as the function `OrbitDecompositionMonRep` (see 1.110).

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G,
> [ [[Z(2), Z(2), 0*Z(2), 0*Z(2)], [0*Z(2), Z(2), 0*Z(2), 0*Z(2)],
> [0*Z(2), 0*Z(2), Z(2), Z(2)], [0*Z(2), 0*Z(2), 0*Z(2), Z(2)]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ [ [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ]
  ],
  "hom"
)
gap> kbsDecompositionARep(R);
ConjugateARep(
  DirectSumARep(
```

```
    ARepByImages(
      GroupWithGenerators( [ (1,2) ] ),
      [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] ],
      "hom"
    ),
    ARepByImages(
      GroupWithGenerators( [ (1,2) ] ),
      [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] ],
      "hom"
    )
  ),
  IdentityPermAMat(4, GF(2))
)
```

1.122 ExtensionOnedimensionalAbelianRep

`ExtensionOnedimensionalAbelianRep(` R, G `)`

Let R be an arep of the subgroup $H \leq G$ and let $G/\mathrm{kernel}(R)$ be an abelian factor group. `ExtensionOnedimensionalAbelianRep` returns an arep of type `"mon"` and degree 1 extending R to G. For the extension the smallest possible extension field is chosen.

```
gap> G := GroupWithGenerators(CyclicGroup(8));
Group( (1,2,3,4,5,6,7,8) )
gap> H := GroupWithGenerators( [ G.1^2 ] );
Group( (1,3,5,7)(2,4,6,8) )
gap> R := ARepByImages(H, [ [[-1]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,3,5,7)(2,4,6,8) ] ),
  [ [ [ -1 ] ]
  ],
  "hom"
)
gap> ExtensionOnedimensionalAbelianRep(R, G);
ARepByImages(
  GroupWithGenerators( [ (1,2,3,4,5,6,7,8) ] ),
  [ Mon( [ E(4) ] )
  ],
  "hom"
)
```

1.123 DecompositionMonRep

`DecompositionMonRep(` R `)`
`DecompositionMonRep(` R, *hint* `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `)` yields `true`). `DecompositionMonRep` returns an arep equal to R with structure $(R_1 \oplus \ldots \oplus R_k)^{A^{-1}}$ where all R_i are irreducible and A^{-1} is a highly structured amat (see 1.22). A is a decomposition matrix for R and can be accessed in the field `.conjugation.element` of the result. The list of the R_i can be accessed in the field `.rep.summands` of the result. Note that any R_i is monomial if this is possible. If the *hint* `"noOuter"` is supplied, the decomposition of R is performed without any decomposition into an outer tensor product which may speed up the function. The function only works for characteristic zero otherwise an error is signaled. At least the following types of monomial areps can be decomposed: monomial representations of solvable groups, double transitive permutation representations, primitive permutation representations with solvable socle. If `DecompositionMonRep` is not able to decompose R then `false` is returned. The performance of `DecompositionMonRep` depends on the size of the group represented as well as on the degree of R. E.g. the decomposition of a regular representation of a group of size 96 takes less than half a minute (CPU-time on a SUN Ultra-Sparc 150 MHz) if the source group is an ag group.

Note that in the case that R is a regular representation of the solvable group G the structured decomposition matrix A computed by `DecompositionMonRep` represents a fast Fourier transform for G. Hence, `DecompositionMonRep` is able to compute a fast Fourier transform for any solvable group.

The algorithm is a major result of [Püs98] where a thorough explanation can be found.

Set `InfoLatticeDec := Print` to obtain information on the recursive decomposition of R.

An important application of this function is the automatic generation of fast algorithms for discrete signal transforms which is realized in 1.147. (see [Min93], [Egn97], [Püs98]).

```
gap> G := GroupWithGenerators(SolvableGroup(8, 5));
Q8
gap> R := RegularARep(G);
RegularARep( Q8 )
gap> DecompositionMonRep(R);
ConjugateARep(
  DirectSumARep(
    TrivialMonARep( Q8 ),
    ARepByImages(
      Q8,
      [ Mon( [ -1 ] ), Mon( [ -1 ] ), Mon( (), 1 ) ],
      "hom"
    ),
    ARepByImages(
      Q8,
      [ Mon( [ -1 ] ), Mon( (), 1 ), Mon( (), 1 ) ],
      "hom"
    ),
    ARepByImages(
      Q8,
      [ Mon( (), 1 ), Mon( [ -1 ] ), Mon( (), 1 ) ],
      "hom"
    ),
    ARepByImages(
      Q8,
      [ Mon( (1,2), [ -1, 1 ] ),
        Mon( [ E(4), -E(4) ] ),
```

```
          Mon( [ -1, -1 ] )
        ],
        "hom"
      ),
      ARepByImages(
        Q8,
        [ Mon( (1,2), [ -1, 1 ] ),
          Mon( [ E(4), -E(4) ] ),
          Mon( [ -1, -1 ] )
        ],
        "hom"
      )
    ),
    ( AMatPerm((7,8), 8) *
      TensorProductAMat(
        IdentityPermAMat(2),
        AMatPerm((2,3), 4) *
        TensorProductAMat(
          DFTAMat(2),
          IdentityPermAMat(2)
        ) *
        DiagonalAMat([ 1, 1, 1, E(4) ]) *
        TensorProductAMat(
          IdentityPermAMat(2),
          DFTAMat(2)
        ) *
        AMatPerm((2,3), 4)
      ) *
      AMatMon( Mon(
        (2,5,3)(4,8,7),
        [ 1, 1, 1, 1, 1, 1, -1, 1 ]
      ) ) *
      DirectSumAMat(
        TensorProductAMat(
          DFTAMat(2),
          IdentityPermAMat(2)
        ),
        IdentityPermAMat(4)
      ) *
      AMatPerm((2,4), 8)
    ) ^ -1
  )
  gap>  last = R;
  true
```

1.124 Symmetry of Matrices

The following sections describe functions for the computation of symmetry of a given matrix.

A symmetry of a matrix is a pair (R_1, R_2) of representations of the same group G with the property $R_1(x) \cdot M = M \cdot R_2(x)$ for all $x \in G$. This definition corresponds to the definition of the intertwining space of R_1, R_2 (see 1.104). The origin of this definition is due to Minkwitz (see [Min95], [Min93]) and was generalized to the definition above by the authors of this package.

Restrictions on the representations R_1, R_2 yield special types of symmetry. We consider the following three types:

- Perm-Irred symmetry: R_1 is a permutation representation, R_2 is a conjugated (by a permutation) direct sum of irreducible representations
- Perm-Perm symmetry: both R_1 and R_2 are permutation representations
- Mon-Mon symmetry: both R_1 and R_2 are monomial representations

There are two implementations for the search algorithm for Perm-Perm-Symmetry. One is entirely in GAP by S. Egner, the other uses the external C-program `desauto` bei J. Leon which is distributed with the GUAVA package. By default the GAP code is run. In order to use the much faster method of J. Leon based on partitions (see [Leo91]) you should set `UseLeon := true` and make sure that an executable version of `desauto` is placed in `$GAP/pkg/arep/bin`. The implementation of Leon requires the matrix to have ≤ 256 different entries. If this condition is violated the GAP implementation is run.

A matrix with symmetry of one of the types above contains structure in a sense and can be decomposed into a product of highly structured sparse matrices (see 1.147).

For details on the concept and computation of symmetry see [Egn97] and [Püs98].

The following functions are implemented in the file `"arep/lib/symmetry.g"` based on functions from `"arep/lib/permperm.g"`, `"arep/lib/monmon.g"`, `"arep/lib/permblk.g"` and `"arep/lib/permmat.g"`.

1.125 PermPermSymmetry

`PermPermSymmetry(` M `)`

Let M be a matrix or an amat (see 1.22). `PermPermSymmetry` returns a pair (R_1, R_2) of areps of type `"perm"` (see 1.66) of the same group G representing the perm-perm symmetry of M, i.e. $R_1(x) \cdot M = M \cdot R_2(x)$ for all $x \in G$. The returned symmetry is maximal in the sense that for every pair (p_1, p_2) of permutations satisfying $p_1 \cdot M = M \cdot p_2$ there is an x with $p_1 = R_1(x)$ and $p_2 = R_2(x)$.

To use the much faster implementation of J. Leon set `UseLeon := true` as explained in 1.124.

Set `InfoPermSym1 := true` to obtain information about the search.

For the algorithm see [Leo91] resp. [Egn97].

```
gap> M := DFT(5);;
gap> PrintArray(M);
[ [       1,       1,       1,       1,       1 ],
  [       1,    E(5),  E(5)^2,  E(5)^3,  E(5)^4 ],
```

```
  [       1,  E(5)^2,  E(5)^4,    E(5),  E(5)^3 ],
  [       1,  E(5)^3,    E(5),  E(5)^4,  E(5)^2 ],
  [       1,  E(5)^4,  E(5)^3,  E(5)^2,    E(5) ] ]
gap> L := PermPermSymmetry(M);
[ ARepByImages(
      GroupWithGenerators( [ g1, g2 ] ),
      [ (2,3,5,4),
        (2,5)(3,4)
      ],
      5, # degree
      "hom"
    ), ARepByImages(
      GroupWithGenerators( [ g1, g2 ] ),
      [ (2,4,5,3),
        (2,5)(3,4)
      ],
      5, # degree
      "hom"
    ) ]
gap> L[1]^AMatMat(M) = L[2];
true
```

1.126 MonMonSymmetry

`MonMonSymmetry(` M `)`

Let M be a matrix or an amat (see 1.22) of characteristic zero. `MonMonSymmetry` returns a pair (R_1, R_2) of areps of type `"mon"` (see 1.66) of the same group G representing a mon-mon symmetry of M, i.e. $R_1(x) \cdot M = M \cdot R_2(x)$ for all $x \in G$.

The non-zero entries in the matrices $R_1(x), R_2(x)$ are all roots of unity of a certain order d. This order is given by the lcm of all quotients of non-zero entries of M with equal absolute value. The returned symmetry is maximal in the sense that for every pair (m_1, m_2) of monomial matrices containing only dth roots of unity (and 0) and satisfying $m_1 \cdot M = M \cdot m_2$ there is an x with $m_1 = R_1(x)$ and $m_2 = R_2(x)$.

`MonMonSymmetry` uses the function `PermPermSymmetry`. Hence you can accelerate the function using the faster implementation of J. Leon by setting `UseLeon := true` as explained in 1.124.

For an explanation of the algorithm see [Püs98].

```
gap> M := DFT(5);;
gap> PrintArray(M);
[ [       1,       1,       1,       1,       1 ],
  [       1,    E(5),  E(5)^2,  E(5)^3,  E(5)^4 ],
  [       1,  E(5)^2,  E(5)^4,    E(5),  E(5)^3 ],
  [       1,  E(5)^3,    E(5),  E(5)^4,  E(5)^2 ],
  [       1,  E(5)^4,  E(5)^3,  E(5)^2,    E(5) ] ]
gap> L := MonMonSymmetry(M);
[ ARepByImages(
```

```
      GroupWithGenerators( [ g1, g2, g3, g4, g5 ] ),
      [ Mon(
          (2,3,5,4),
          [ 1, E(5)^3, E(5), E(5)^4, E(5)^2 ]
        ),
        Mon(
          (2,5)(3,4),
          [ 1, E(5)^2, E(5)^4, E(5), E(5)^3 ]
        ),
        Mon(
          (1,2,3,4,5),
          [ E(5), E(5), E(5), E(5), E(5) ]
        ),
        Mon( [ E(5), 1, E(5)^4, E(5)^3, E(5)^2 ] ),
        Mon( [ 1, E(5), E(5)^2, E(5)^3, E(5)^4 ] )
      ],
      "hom"
    ), ARepByImages(
      GroupWithGenerators( [ g1, g2, g3, g4, g5 ] ),
      [ Mon( (1,3,4,2), 5 ),
        Mon( (1,4)(2,3), 5 ),
        Mon( [ E(5), E(5)^2, E(5)^3, E(5)^4, 1 ] ),
        Mon(
          (1,2,3,4,5),
          [ E(5), E(5), E(5), E(5), E(5) ]
        ),
        Mon( (1,5,4,3,2), 5 )
      ],
      "hom"
    ) ]
gap> L[1]^AMatMat(M) = L[2];
true
```

1.127 PermIrredSymmetry

`PermIrredSymmetry(` *M* `)`
`PermIrredSymmetry(` *M* `,` *maxblocksize* `)`

Let M be a matrix or an amat (see 1.22) of characteristic zero. `PermIrredSymmetry` returns a list of pairs (R_1, R_2) of areps (see 1.66) of the same group G representing a perm-irred symmetry of M, i.e. $R_1(x) \cdot M = M \cdot R_2(x)$ for all $x \in G$ and R_1 is a permutation representation and R_2 a conjugated (by a permutation) direct sum of irreducible representations. If *maxblocksize* is supplied exactly those perm-irred symmetries are returned where R_2 contains at least one irreducible of degree $\leq$ *maxblocksize*. The default for *maxblocksize* is 2.

Refer to [Egn97] to understand how the search is done and how to interpret the result.

Note that the perm-irred symmetry is not symmetric. Hence it is possible that a matrix M admits a perm-irred symmetry but its transpose not.

The perm-irred symmetry is a special case of a perm-block symmetry. The perm-block symmetries admitted by a fixed matrix M can be described by two lattices which are in a certain way related to each other (semi-order preserving). To explore this structure (described in [Egn97]) you should refer to `PermBlockSym` and `DisplayPermBlockSym` in the file `"arep/lib/permblk.g"`.

```
gap> M := DFT(4);
[ [ 1, 1, 1, 1 ], [ 1, E(4), -1, -E(4) ], [ 1, -1, 1, -1 ],
  [ 1, -E(4), -1, E(4) ] ]
gap> PermIrredSymmetry(M);
[ [ NaturalARep( G2, 4 ), ConjugateARep(
          DirectSumARep(
            TrivialMatARep( G2 ),
            ARepByImages(
              G2,
              [ [ [ -1 ] ],
                [ [ E(4) ] ]
              ],
              "hom"
            ),
            ARepByImages(
              G2,
              [ [ [ 1 ] ],
                [ [ -1 ] ]
              ],
              "hom"
            ),
            ARepByImages(
              G2,
              [ [ [ -1 ] ],
                [ [ -E(4) ] ]
              ],
              "hom"
            )
          ),
          IdentityPermAMat(4)
        ) ], [ NaturalARep( G3, 4 ), ConjugateARep(
          DirectSumARep(
            TrivialMatARep( G3 ),
            ARepByImages(
              G3,
              [ [ [ 0, -E(4) ], [ E(4), 0 ] ],
                [ [ 0, 1 ], [ 1, 0 ] ],
                [ [ 0, -1 ], [ -1, 0 ] ]
              ],
              "hom"
            ),
            ARepByImages(
```

```
        G3,
        [ [ [ -1 ] ],
          [ [ 1 ] ],
          [ [ 1 ] ]
        ],
        "hom"
      )
    ),
    AMatPerm((3,4), 4)
  ) ], [ NaturalARep( G1, 4 ), ConjugateARep(
    DirectSumARep(
      TrivialMatARep( G1 ),
      ARepByImages(
        G1,
        [ [ [ 1/2, -1/2+1/2*E(4), 1/2*E(4) ],
    [ -1/2-1/2*E(4), 0, -1/2+1/2*E(4) ],
    [ -1/2*E(4), -1/2-1/2*E(4), 1/2 ] ],
          [ [ 0, 0, 1 ], [ 0, 1, 0 ], [ 1, 0, 0 ] ],
          [ [ 1/2, 1/2+1/2*E(4), -1/2*E(4) ],
    [ 1/2-1/2*E(4), 0, 1/2+1/2*E(4) ],
    [ 1/2*E(4), 1/2-1/2*E(4), 1/2 ] ]
        ],
        "hom"
      )
    ),
    IdentityPermAMat(4)
  ) ] ]
```

1.128 Discrete Signal Transforms

The following sections describe functions for the construction of many well known signal transforms in matrix form, as e.g. the discrete Fourier transform, several discrete cosine transforms etc. For the definition of the mentioned signal transforms see [ER82], [Mal92], [Mer96].

The functions for discrete signal transforms are implemented in `"arep/lib/transf.g"`.

1.129 DiscreteFourierTransform

`DiscreteFourierTransform(` *r* `)`
`DiscreteFourierTransform(` *n* `)`
`DiscreteFourierTransform(` *n*`,` *char* `)`

shortcut: `DFT`

`DiscreteFourierTransform` or `DFT` returns the discrete Fourier transform from a given root of unity r or the size n and the characteristic *char* (see [CB93]). The default for *char* is zero. Note that the DFT on n points and characteristic *char* exists iff n and *char* are coprime. If this condition is violenced an error is signaled.

The DFT_n of size n is defined as $\mathrm{DFT}_n = [\omega_n^{k\ell} \mid k, \ell \in \{0, \ldots, n-1\}]$, ω_n a primitive nth root of unity.

```
gap> DFT(Z(3));
[ [ Z(3)^0, Z(3)^0 ], [ Z(3)^0, Z(3) ] ]
gap> DFT(4);
[ [ 1, 1, 1, 1 ], [ 1, E(4), -1, -E(4) ], [ 1, -1, 1, -1 ],
  [ 1, -E(4), -1, E(4) ] ]
```

1.130 InverseDiscreteFourierTransform

`InverseDiscreteFourierTransform(` *r* `)`
`InverseDiscreteFourierTransform(` *n* `)`
`InverseDiscreteFourierTransform(` *n*`,` *char* `)`

shortcut: `InvDFT`

`InverseDiscreteFourierTransform` or `InvDFT` returns the inverse of the discrete Fourier transform from a given root of unity r or the size n and the characteristic *char* (see 1.129). The default for *char* is zero.

```
gap> InvDFT(3);
[ [ 1/3, 1/3, 1/3 ], [ 1/3, 1/3*E(3)^2, 1/3*E(3) ],
  [ 1/3, 1/3*E(3), 1/3*E(3)^2 ] ]
```

1.131 DiscreteHartleyTransform

`DiscreteHartleyTransform(` *n* `)`

shortcut: `DHT`

`DiscreteHartleyTransform` or `DHT` returns the discrete Hartley transform on n points.

The DHT_n of size n is defined by $\mathrm{DHT}_n = [1/\sqrt{n} \cdot (\cos(2\pi k\ell/n) + \sin(2\pi k\ell/n)) \mid k, \ell \in \{0, \ldots, n-1\}]$.

```
gap> DHT(4);
[ [ 1/2, 1/2, 1/2, 1/2 ], [ 1/2, 1/2, -1/2, -1/2 ],
  [ 1/2, -1/2, 1/2, -1/2 ], [ 1/2, -1/2, -1/2, 1/2 ] ]
```

1.132 InverseDiscreteHartleyTransform

`InverseDiscreteHartleyTransform(` *n* `)`

shortcut: `InvDHT`

`InverseDiscreteHartleyTransform` or `InvDHT` returns the inverse of the discrete Hartley transform on n points. Since the DHT is self inverse the result is exactly the same as from `DHT` above.

```
gap> InvDHT(4);
[ [ 1/2, 1/2, 1/2, 1/2 ], [ 1/2, 1/2, -1/2, -1/2 ],
  [ 1/2, -1/2, 1/2, -1/2 ], [ 1/2, -1/2, -1/2, 1/2 ] ]
```

1.133 DiscreteCosineTransform

`DiscreteCosineTransform(` n `)`

shortcut: `DCT`

`DiscreteCosineTransform` returns the standard cosine transform (type II) on n points.

The DCT_n of size n is defined by $\mathrm{DCT}_n = [\sqrt{2/n} \cdot c_k \cdot (\cos(k(\ell+1/2)\pi/n) \mid k, \ell \in \{0, \dots, n-1\}]$, $c_k = 1/\sqrt{2}$ for $k = 0$ and $c_k = 1$ else.

```
gap> DCT(3);
[ [ 1/3*E(12)^7-1/3*E(12)^11, 1/3*E(12)^7-1/3*E(12)^11,
      1/3*E(12)^7-1/3*E(12)^11 ],
  [ -1/2*E(8)+1/2*E(8)^3, 0, 1/2*E(8)-1/2*E(8)^3 ],
  [ -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19,
      1/3*E(24)-1/3*E(24)^11-1/3*E(24)^17+1/3*E(24)^19,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ] ]
```

1.134 InverseDiscreteCosineTransform

`InverseDiscreteCosineTransform(` n `)`

shortcut: `InvDCT`

`InverseDiscreteCosineTransform` returns the inverse of the standard cosine transform (type II) on n points. Since the DCT is orthogonal, the result is the transpose of the DCT, which is exactly the discrete cosine transform of type III.

```
[ [ 1/3*E(12)^7-1/3*E(12)^11, -1/2*E(8)+1/2*E(8)^3,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, 0,
      1/3*E(24)-1/3*E(24)^11-1/3*E(24)^17+1/3*E(24)^19 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, 1/2*E(8)-1/2*E(8)^3,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ] ]
```

1.135 DiscreteCosineTransformIV

`DiscreteCosineTransformIV(` n `)`

shortcut: `DCT_IV`

`DiscreteCosineTransformIV` returns the cosine transform of type IV on n points.

The $\mathrm{DCT\_IV}_n$ of size n is defined by $\mathrm{DCT\_IV}_n = [\sqrt{2/n} \cdot (\cos((k + 1/2)(\ell + 1/2)\pi/n) \mid k, \ell \in \{0, \dots, n-1\}]$.

```
[ [ 1/2*E(12)^4+1/6*E(12)^7+1/2*E(12)^8-1/6*E(12)^11,
      1/3*E(12)^7-1/3*E(12)^11,
      1/2*E(12)^4-1/6*E(12)^7+1/2*E(12)^8+1/6*E(12)^11 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, -1/3*E(12)^7+1/3*E(12)^11,
      -1/3*E(12)^7+1/3*E(12)^11 ],
  [ 1/2*E(12)^4-1/6*E(12)^7+1/2*E(12)^8+1/6*E(12)^11,
      -1/3*E(12)^7+1/3*E(12)^11,
      1/2*E(12)^4+1/6*E(12)^7+1/2*E(12)^8-1/6*E(12)^11 ] ]
```

1.136 InverseDiscreteCosineTransformIV

`InverseDiscreteCosineTransformIV(` n `)`

shortcut: `InvDCT_IV`

`InverseDiscreteCosineTransformIV` returns the inverse of the cosine transform of type IV on n points. Since the DCT_IV is orthogonal, the result is the transpose of the DCT_IV.

```
[ [ 1/3*E(12)^7-1/3*E(12)^11, -1/2*E(8)+1/2*E(8)^3,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, 0,
      1/3*E(24)-1/3*E(24)^11-1/3*E(24)^17+1/3*E(24)^19 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, 1/2*E(8)-1/2*E(8)^3,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ] ]
```

1.137 DiscreteCosineTransformI

`DiscreteCosineTransformI(` n `)`

shortcut: `DCT_I`

`DiscreteCosineTransformI` returns the cosine transform of type I on $n+1$ points.

The $\mathrm{DCT\_I}_n$ of size $n+1$ is defined by $\mathrm{DCT\_I}_n = [\sqrt{2/n} \cdot c_k \cdot c_\ell \cdot (\cos(k\ell\pi/n) \mid k,\ell \in \{0,\ldots,n\}]$, $c_k = 1/\sqrt{2}$ for $k=0$ and $c_k = 1$ else.

```
[ [ 1/2, 1/2*E(8)-1/2*E(8)^3, 1/2 ],
  [ 1/2*E(8)-1/2*E(8)^3, 0, -1/2*E(8)+1/2*E(8)^3 ],
  [ 1/2, -1/2*E(8)+1/2*E(8)^3, 1/2 ] ]
```

1.138 InverseDiscreteCosineTransformI

`InverseDiscreteCosineTransformI(` n `)`

shortcut: `InvDCT_I`

`InverseDiscreteCosineTransformI` returns the inverse of the cosine transform of type I on n points. Since the DCT_I is orthogonal, the result is the transpose of the DCT_I.

```
[ [ 1/2, 1/2*E(8)-1/2*E(8)^3, 1/2 ],
  [ 1/2*E(8)-1/2*E(8)^3, 0, -1/2*E(8)+1/2*E(8)^3 ],
  [ 1/2, -1/2*E(8)+1/2*E(8)^3, 1/2 ] ]
```

1.139 WalshHadamardTransform

`WalshHadamardTransform(` n `)`

shortcut: `WHT`

`WalshHadamardTransform` returns the Walsh-Hadamard transform on n points.

Let $n = \prod_{i=1}^{k} p_i^{\nu_i}$ be the prime factor decomposition of n. Then the WHT_n is defined by $\mathrm{WHT}_n = \bigotimes_{i=1}^{k} \mathrm{DFT}_{p_i}^{\otimes \nu_i}$.

```
gap> WHT(4);
[ [ 1, 1, 1, 1 ], [ 1, -1, 1, -1 ],
  [ 1, 1, -1, -1 ], [ 1, -1, -1, 1 ] ]
```

1.140 InverseWalshHadamardTransform

`InverseWalshHadamardTransform(` n `)`

shortcut: `InvWHT`

`InverseWalshHadamardTransform` returns the inverse of the Walsh-Hadamard transform on n points.

```
gap> InvWHT(4);
[ [ 1/4, 1/4, 1/4, 1/4 ], [ 1/4, -1/4, 1/4, -1/4 ],
  [ 1/4, 1/4, -1/4, -1/4 ], [ 1/4, -1/4, -1/4, 1/4 ] ]
```

1.141 SlantTransform

`SlantTransform(` n `)`

shortcut: `ST`

`SlantTransform` returns the Slant transform on n points, which must be a power of 2, $n = 2^k$

For a definition of the Slant transform see [ER82], 10.9.

```
gap> ST(4);
[ [ 1/2, 1/2, 1/2, 1/2 ],
  [ 3/10*E(5)-3/10*E(5)^2-3/10*E(5)^3+3/10*E(5)^4,
    1/10*E(5)-1/10*E(5)^2-1/10*E(5)^3+1/10*E(5)^4,
      -1/10*E(5)+1/10*E(5)^2+1/10*E(5)^3-1/10*E(5)^4,
      -3/10*E(5)+3/10*E(5)^2+3/10*E(5)^3-3/10*E(5)^4 ],
  [ 1/2, -1/2, -1/2, 1/2 ],
  [ 1/10*E(5)-1/10*E(5)^2-1/10*E(5)^3+1/10*E(5)^4,
    -3/10*E(5)+3/10*E(5)^2+3/10*E(5)^3-3/10*E(5)^4,
      3/10*E(5)-3/10*E(5)^2-3/10*E(5)^3+3/10*E(5)^4,
      -1/10*E(5)+1/10*E(5)^2+1/10*E(5)^3-1/10*E(5)^4 ] ]
```

1.142 InverseSlantTransform

`InverseSlantTransform(` n `)`

shortcut: `InvST`

`InverseSlantTransform` returns the inverse of the Slant transform on n points, which must be a power of 2, $n = 2^k$. Since ST is orthogonal, this is exactly the transpose of the ST.

```
gap> InvST(4);
[ [ 1/2, 3/10*E(5)-3/10*E(5)^2-3/10*E(5)^3+3/10*E(5)^4, 1/2,
      1/10*E(5)-1/10*E(5)^2-1/10*E(5)^3+1/10*E(5)^4 ],
  [ 1/2, 1/10*E(5)-1/10*E(5)^2-1/10*E(5)^3+1/10*E(5)^4, -1/2,
      -3/10*E(5)+3/10*E(5)^2+3/10*E(5)^3-3/10*E(5)^4 ],
  [ 1/2, -1/10*E(5)+1/10*E(5)^2+1/10*E(5)^3-1/10*E(5)^4, -1/2,
      3/10*E(5)-3/10*E(5)^2-3/10*E(5)^3+3/10*E(5)^4 ],
  [ 1/2, -3/10*E(5)+3/10*E(5)^2+3/10*E(5)^3-3/10*E(5)^4, 1/2,
      -1/10*E(5)+1/10*E(5)^2+1/10*E(5)^3-1/10*E(5)^4 ] ]
```

1.143 HaarTransform

`HaarTransform(` n `)`

shortcut: `HT`

`HaarTransform` returns the Haar transform on n points, which must be a power of 2, $n = 2^k$.

For a definition of the Haar transform see [ER82], 10.10.

```
gap> HT(4);
[ [ 1/4, 1/4, 1/4, 1/4 ], [ 1/4, 1/4, -1/4, -1/4 ],
  [ 1/4*E(8)-1/4*E(8)^3, -1/4*E(8)+1/4*E(8)^3, 0, 0 ],
  [ 0, 0, 1/4*E(8)-1/4*E(8)^3, -1/4*E(8)+1/4*E(8)^3 ] ]
```

1.144 InverseHaarTransform

`InverseHaarTransform(` n `)`

shortcut: `InvHT`

`InverseHaarTransform` returns the inverse of the Haar transform on n points, which must be a power of 2, $n = 2^k$.

The inverse is exactly n times the transpose of HT.

```
gap> InvHT(4);
[ [ 1, 1, E(8)-E(8)^3, 0 ], [ 1, 1, -E(8)+E(8)^3, 0 ],
  [ 1, -1, 0, E(8)-E(8)^3 ], [ 1, -1, 0, -E(8)+E(8)^3 ] ]
```

1.145 RationalizedHaarTransform

`RationalizedHaarTransform(` n `)`

shortcut: `RHT`

`RationalizedHaarTransform` returns the rationalized Haar transform on n points, which must be a power of 2, $n = 2^k$.

For a definition of the rationalized Haar transform see [ER82], 10.11.

```
gap> RHT(4);
[ [ 1, 1, 1, 1 ], [ 1, 1, -1, -1 ],
  [ 1, -1, 0, 0 ], [ 0, 0, 1, -1 ] ]
```

1.146 InverseRationalizedHaarTransform

`InverseRationalizedHaarTransform(` n `)`

shortcut: `InvRHT`

`InverseRationalizedHaarTransform` returns the inverse of the rationalized Haar transform on n points, which must be a power of 2, $n = 2^k$.

```
gap> InvRHT(4);
[ [ 1/4, 1/4, 1/2, 0 ], [ 1/4, 1/4, -1/2, 0 ],
  [ 1/4, -1/4, 0, 1/2 ], [ 1/4, -1/4, 0, -1/2 ] ]
```

1.147 Matrix Decomposition

The decomposition of a matrix M with symmetry is a striking application of constructive representation theory and was the original motivation to create the package AREP. Here, decomposition means that M is decomposed into a product of highly structured sparse matrices. Applied to matrices corresponding to discrete signal transforms such a decomposition may represent a fast algorithm for the signal transform.

For the definition of symmetry see 1.124.

The idea of decomposing a matrix with symmetry is due to Minkwitz [Min95], [Min93] and was further developed by the authors of this package. See [Egn97], chapter 1 or [Püs98], chapter 3 for a thorough explanation of the method.

The following three functions correspond to the three types of symmetry considered in this package (see 1.124). The functions are implemented in the file `"arep/lib/algogen.g"`.

1.148 MatrixDecompositionByPermPermSymmetry

`MatrixDecompositionByPermPermSymmetry(` M `)`

Let M be a matrix or an amat (see 1.22). `MatrixDecompositionByPermPermSymmetry` returns a highly structured amat of type `"product"` with all factors being sparse which represents the matrix M. The returned amat can be viewed as a fast algorithm for the multiplication with M.

The function uses the perm-perm symmetry (see 1.125) to decompose the matrix (see 1.147) and can hence be accelerated by setting `UseLeon := true` as described in 1.124.

The following examples show that `MatrixDecompositionByPermPermSymmetry` discovers automatically the method of Rader (see [Rad68]) for a discrete Fourier transform of prime degree as well as the well-known decomposition of circulant matrices.

```
gap> M := DFT(5);;
gap> PrintArray(M);
[ [       1,       1,       1,       1,       1 ],
  [       1,    E(5),  E(5)^2,  E(5)^3,  E(5)^4 ],
  [       1,  E(5)^2,  E(5)^4,    E(5),  E(5)^3 ],
  [       1,  E(5)^3,    E(5),  E(5)^4,  E(5)^2 ],
  [       1,  E(5)^4,  E(5)^3,  E(5)^2,    E(5) ] ]
gap> MatrixDecompositionByPermPermSymmetry(M);
AMatPerm((4,5), 5) *
DirectSumAMat(
  IdentityPermAMat(1),
  TensorProductAMat(
    DFTAMat(2),
    IdentityPermAMat(2)
  ) *
  DiagonalAMat([ 1, 1, 1, E(4) ]) *
  TensorProductAMat(
    IdentityPermAMat(2),
    DFTAMat(2)
```

```
    ) *
    AMatPerm((2,3), 4)
  ) *
  AMatPerm((1,4,2,5,3), 5) *
  DirectSumAMat(
    DiagonalAMat([ E(20)^4-E(20)^13-E(20)^16+E(20)^17,
    E(5)-E(5)^2-E(5)^3+E(5)^4, E(20)^4+E(20)^13-E(20)^16-E(20)^17 ]),
    AMatMat(
      [ [ 1, 4 ], [ 1, -1 ] ]
    )
  ) *
  AMatPerm((1,3,5,2,4), 5) *
  DirectSumAMat(
    IdentityPermAMat(1),
    AMatPerm((2,3), 4) *
    TensorProductAMat(
      IdentityPermAMat(2),
      DiagonalAMat([ 1/2, 1/2 ]) *
      DFTAMat(2)
    ) *
    DiagonalAMat([ 1, 1, 1, -E(4) ]) *
    TensorProductAMat(
      DiagonalAMat([ 1/2, 1/2 ]) *
      DFTAMat(2),
      IdentityPermAMat(2)
    )
  ) *
  AMatPerm((3,4,5), 5)

  gap> M := [[1, 2, 3], [3, 1, 2], [2, 3, 1]];;
  gap> PrintArray(M);
  [ [  1,  2,  3 ],
    [  3,  1,  2 ],
    [  2,  3,  1 ] ]
  gap> MatrixDecompositionByPermPermSymmetry(M);
  DFTAMat(3) *
  AMatMon( Mon(
    (2,3),
    [ 2, 2/3*E(3)+1/3*E(3)^2, 1/3*E(3)+2/3*E(3)^2 ]
  ) ) *
  DFTAMat(3)
```

1.149 MatrixDecompositionByMonMonSymmetry

`MatrixDecompositionByMonMonSymmetry(` M `)`

Let M be a matrix or an amat (see 1.22). `MatrixDecompositionByMonMonSymmetry` returns a highly structured amat of type `"product"` with all factors being sparse which represents the matrix M. The returned amat can be viewed as a fast algorithm for the multiplication with M.

The function uses the mon-mon symmetry (see 1.126) to decompose the matrix (see 1.147) and can hence be accelerated by setting `UseLeon := true` as described in 1.124.

The following example show that `MatrixDecompositionByMonMonSymmetry` is able to find automatically a decomposition of the discrete cosine transform of type IV (see 1.135).

```
gap> M := DCT_IV(8);;
gap> MatrixDecompositionByMonMonSymmetry(M);
AMatMon( Mon(
  (3,4,7,6,8,5),
  [ E(4), E(16)^5, E(8)^3, -E(16)^7, 1, -E(16), E(8), -E(16)^3 ]
) ) *
TensorProductAMat(
  DFTAMat(2),
  IdentityPermAMat(4)
) *
DiagonalAMat([ 1, 1, 1, 1, 1, E(8), E(4), E(8)^3 ]) *
TensorProductAMat(
  IdentityPermAMat(2),
  DFTAMat(2),
  IdentityPermAMat(2)
) *
DiagonalAMat([ 1, 1, 1, E(4), 1, 1, 1, E(4) ]) *
TensorProductAMat(
  IdentityPermAMat(4),
  DFTAMat(2)
) *
DiagonalAMat([ -E(64), -E(64), E(64)^9, -E(64)^9, E(64)^23, -E(64)^23,
  E(64)^31, E(64)^31 ]) *
TensorProductAMat(
  IdentityPermAMat(4),
  DiagonalAMat([ 1/2, 1/2 ]) *
  DFTAMat(2)
) *
DiagonalAMat([ 1, 1, 1, -E(4), 1, 1, 1, -E(4) ]) *
TensorProductAMat(
  IdentityPermAMat(2),
  DiagonalAMat([ 1/2, 1/2 ]) *
  DFTAMat(2),
  IdentityPermAMat(2)
) *
DiagonalAMat([ 1, 1, 1, 1, 1, -E(8)^3, -E(4), -E(8) ]) *
TensorProductAMat(
  DiagonalAMat([ 1/2, 1/2 ]) *
  DFTAMat(2),
  IdentityPermAMat(4)
) *
```

```
AMatMon( Mon(
  (2,6,3,4,7,5,8),
  [ E(4), E(16)^5, -E(16)^7, E(8), E(8)^3, -E(16)^3, -E(16), 1 ]
) )
```

1.150 MatrixDecompositionByPermIrredSymmetry

`MatrixDecompositionByPermIrredSymmetry(` *M* `)`
`MatrixDecompositionByPermIrredSymmetry(` *M* , *maxblocksize* `)`

Let M be a matrix or an amat (see 1.22). `MatrixDecompositionByPermIrredSymmetry` returns a highly structured amat of type `"product"` with all factors being sparse which represents the matrix M. The returned amat can be viewed as a fast algorithm for the multiplication with M.

The function uses the perm-irred symmetry (see 1.127) to decompose the matrix (see 1.147).

If *maxblocksize* is supplied only those perm-irred symmetries with all irreducibles having degree less than *maxblocksize* are considered. The default for *maxblocksize* is 2.

Note that the perm-irred symmetry is not symmetric. Hence it is possible that a matrix M decomposes but its transpose not.

The following examples show that `MatrixDecompositionByPermIrredSymmetry` discovers automatically the Cooley-Tukey decomposition (see [CT65]) of a discrete Fourier transform as well as a decomposition of the transposed discrete cosine transform of type II (see 1.133).

```
gap> M := DFT(4);
[ [ 1, 1, 1, 1 ], [ 1, E(4), -1, -E(4) ], [ 1, -1, 1, -1 ],
  [ 1, -E(4), -1, E(4) ] ]
gap> MatrixDecompositionByPermIrredSymmetry(M);
TensorProductAMat(
  DFTAMat(2),
  IdentityPermAMat(2)
) *
DiagonalAMat([ 1, 1, 1, E(4) ]) *
TensorProductAMat(
  IdentityPermAMat(2),
  DFTAMat(2)
) *
AMatPerm((2,3), 4)

gap> M := TransposedMat(DCT(8));;
gap> MatrixDecompositionByPermIrredSymmetry(M);
AMatPerm((1,2,6,7,5,3,8), 8) *
TensorProductAMat(
  IdentityPermAMat(2),
  AMatPerm((3,4), 4) *
  TensorProductAMat(
    IdentityPermAMat(2),
    DFTAMat(2)
  ) *
```

```
      AMatPerm((2,3), 4) *
      DirectSumAMat(
        DFTAMat(2),
        IdentityPermAMat(2)
      )
    ) *
    AMatPerm((2,7,5,4,3)(6,8), 8) *
    DirectSumAMat(
      IdentityPermAMat(3),
      DirectSumAMat(
        IdentityPermAMat(1),
        AMatMat(
          [ [ -1/2*E(8)+1/2*E(8)^3, 1/2*E(8)-1/2*E(8)^3 ],
            [  1/2*E(8)-1/2*E(8)^3, 1/2*E(8)-1/2*E(8)^3 ] ],
          "invertible"
        )
      ),
      IdentityPermAMat(2)
    ) *
    DirectSumAMat(
      TensorProductAMat(
        DFTAMat(2),
        IdentityPermAMat(3)
      ),
      IdentityPermAMat(2)
    ) *
    AMatPerm((2,7,3,8,4), 8) *
    DirectSumAMat(
      DiagonalAMat([ 1/4*E(8)-1/4*E(8)^3, 1/4*E(8)-1/4*E(8)^3 ]),
      AMatMat(
        [ [ 1/4*E(16)-1/4*E(16)^7, 1/4*E(16)^3-1/4*E(16)^5 ],
          [ 1/4*E(16)^3-1/4*E(16)^5, -1/4*E(16)+1/4*E(16)^7 ] ]
      ),
      AMatMat(
        [ [ -1/4*E(32)+1/4*E(32)^15, -1/4*E(32)^7+1/4*E(32)^9 ],
          [ 1/4*E(32)^7-1/4*E(32)^9, -1/4*E(32)+1/4*E(32)^15 ] ]
      ),
      AMatMat(
        [ [ -1/4*E(32)^3+1/4*E(32)^13, -1/4*E(32)^5+1/4*E(32)^11 ],
          [ -1/4*E(32)^5+1/4*E(32)^11, 1/4*E(32)^3-1/4*E(32)^13 ] ]
      )
    ) *
    AMatPerm((2,5)(4,7)(6,8), 8)
```

1.151 Complex Numbers

The next sections describe basic functions for the calculation with complex numbers which are represented as cyclotomics, e.g. computation of the complex conjugate or certain sine and cosine expressions.

The following functions are implemented in the file `"arep/lib/complex.g"`.

1.152 ImaginaryUnit

`ImaginaryUnit( )`

`ImaginaryUnit` returns `E(4)`.

```
gap> ImaginaryUnit();
E(4)
```

1.153 Re

`Re(` *z* `)`

`Re` returns the real part of the cyclotomic z.

```
gap> z := E(3) + E(4);
E(12)^4-E(12)^7-E(12)^11
gap> Re(z);
-1/2
```

`Re(` *list* `)`

`Re` returns the list of the real parts of the cyclotomics in *list*.

1.154 Im

`Im(` *z* `)`

`Im` returns the imaginary part of the cyclotomic z.

```
gap> z := E(3) + E(4);
E(12)^4-E(12)^7-E(12)^11
gap> Im(z);
-E(12)^4-1/2*E(12)^7-E(12)^8+1/2*E(12)^11
```

`Im(` *list* `)`

`Im` returns the list of the imaginary parts of the cyclotomics in *list*.

1.155 AbsSqr

`AbsSqr(` *z* `)`

`AbsSqr` returns the squared absolute value of the cyclotomic z.

```
gap> AbsSqr(z);
-2*E(12)^4-E(12)^7-2*E(12)^8+E(12)^11
```

`AbsSqr(` *list* `)`

`AbsSqr` returns the list of the squared absolute values of the cyclotomics in *list*.

1.156 Sqrt

`Sqrt(` r `)`

`Sqrt` returns the square root of the rational number r.

```
gap> Sqrt(1/3);
1/3*E(12)^7-1/3*E(12)^11
```

1.157 ExpIPi

`ExpIPi(` r `)`

Let r be a rational number. `ExpIPi` returns $e^{\pi i r}$.

```
gap> ExpIPi(1/5);
-E(5)^3
```

1.158 CosPi

`CosPi(` r `)`

Let r be a rational number. `CosPi(` r `)` returns $\cos(\pi r)$.

```
gap> CosPi(1/5);
-1/2*E(5)^2-1/2*E(5)^3
```

1.159 SinPi

`SinPi(` r `)`

Let r be a rational number. `SinPi(` r `)` returns $\sin(\pi r)$.

```
gap> SinPi(1/5);
 -1/2*E(20)^13+1/2*E(20)^17
```

1.160 TanPi

`TanPi(` r `)`

Let r be a rational number. `TanPi(` r `)` returns $\tan(\pi r)$.

```
gap> TanPi(1/5);
E(20)-E(20)^9+E(20)^13-E(20)^17
```

1.161 Functions for Matrices and Permutations

The following sections describe basic functions for matrices and permutations, like forming the tensor product (Kronecker product) or direct sum and determination of the blockstructure of a matrix.

The following functions are implemented in the files `"arep/lib/permblk.g"` (`kbs`, see 1.169), `"arep/lib/summands.g"` (`DirectSummandsPermutedMat`, see 1.168) and the file `"arep/lib/tools.g"` (the other functions).

1.162 DiagonalMat

`DiagonalMat(` *list* `)`

Let *list* contain field elements of common characteristic. `DiagonalMat` returns the diagonal matrix with *list* as diagonal.

```
gap> DiagonalMat([Z(2), Z(2)]);
[ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ]
```

1.163 DirectSumMat

`DirectSumMat(` M_1, ..., M_k `)`

`DirectSumMat` returns the direct sum of the matrices $M_1, ..., M_k$.

```
gap> DirectSumMat( [[1]], [[1,2], [3,4]], [[5]] );
[ [ 1, 0, 0, 0 ], [ 0, 1, 2, 0 ], [ 0, 3, 4, 0 ], [ 0, 0, 0, 5 ] ]
```

`DirectSumMat(` *list* `)`

`DirectSumMat` returns the direct sum of the matrices in *list*.

1.164 TensorProductMat

`TensorProductMat(` M_1, ..., M_k `)`

`TensorProductMat` returns the tensor product of the matrices $M_1, ..., M_k$.

```
gap> TensorProductMat( [[1]], [[1,2], [3,4]], [[5,6], [7,8]] );
[ [ 5, 6, 10, 12 ], [ 7, 8, 14, 16 ],
  [ 15, 18, 20, 24 ], [ 21, 24, 28, 32 ] ]
```

`TensorProductMat(` *list* `)`

`TensorProductMat` returns the tensor product of the matrices in *list*.

1.165 MatPerm

`MatPerm(` *p*, *d* `)` `MatPerm(` *p*, *d*, *char* `)`

`MatPerm` returns the permutation matrix of degree d corresponding to the permutation p in characteristic *char*. The default characteristic is 0. If d is less than the largest moved point of p an error is signaled.

We use the following convention to create a permutation matrix from a permutation p with degree d $[\delta_{i^p j} \mid i, j \in \{1, \ldots, d\}]$.

```
gap> MatPerm( (1,2,3), 4 );
[ [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, 1 ] ]
```

1.166 PermMat

`PermMat(` M `)`

`PermMat` returns the permutation represented by the matrix M and returns false otherwise. For the convention see 1.165.

```
gap> PermMat( [[0,0,1], [1,0,0], [0,1,0]] );
(1,3,2)
```

1.167 PermutedMat

`PermutedMat(` p_1 `,` M `,` p_2 `)`

Let p_1, p_2 be permutations and M a matrix with r rows and c columns. `PermutedMat` returns `MatPerm(` p_1 `,` r `)` $\cdot M \cdot$ `MatPerm(` p_2 `,` c `)` (see 1.165). The largest moved point of p_1 and p_2 must not exceed r resp. c otherwise an error is signaled.

```
gap> PermutedMat( (1,2), [[1,2,3], [4,5,6], [7,8,9]], (1,2,3) );
[ [ 6, 4, 5 ], [ 3, 1, 2 ], [ 9, 7, 8 ] ]
```

1.168 DirectSummandsPermutedMat

`DirectSummandsPermutedMat(` M `)`
`DirectSummandsPermutedMat(` M `,` *match-blocks* `)`

Let M be a matrix. `DirectSummandsPermutedMat` returns the list `[` p_1 `, [` M_1 `, ...,` M_k `],` p_2 `]` where p_1, p_2 are permutations and M_i, $i = 1, \ldots, k$, are matrices with the property $M =$ `PermutedMat`$(p_1,$ `DirectSumMat`$(M_1, ..., M_k), p_2)$ (see 1.167, 1.163). If *match-blocks* is `true` or not provided then the permutations p_1 and p_2 are chosen such that equivalent M_i are equal and occur next to each other. If *match-blocks* is `false` this is not done.

For an explanation of the algorithm see [Egn97].

```
gap> M := [ [ 0, 0, 0, 2, 0, 1], [ 3, 1, 0, 0, 0, 0],
 > [ 0, 0, 1, 0, 2, 0], [ 1, 2, 0, 0, 0, 0],
 > [ 0, 0, 0, 1, 0, 3], [ 0, 0, 3, 0, 1, 0] ];;
gap> PrintArray(M);
[ [  0,  0,  0,  2,  0,  1 ],
  [  3,  1,  0,  0,  0,  0 ],
  [  0,  0,  1,  0,  2,  0 ],
  [  1,  2,  0,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  3 ],
  [  0,  0,  3,  0,  1,  0 ] ]
gap> DirectSummandsPermutedMat(M);
[ (2,4,3,5),
  [ [ [ 2, 1 ], [ 1, 3 ] ],
    [ [ 2, 1 ], [ 1, 3 ] ],
    [ [ 2, 1 ], [ 1, 3 ] ] ],
  (1,4)(2,6,3) ]
```

1.169 kbs

`kbs(` M `)`

Let M be a square matrix of degree n. `kbs` (konjugierte Blockstruktur = conjugated block structure) returns the partition $\mathrm{kbs}(M) = \{1, \ldots, n\}/R^*$ where R is the reflexive, symmetric, transitive closure of the relation R defined by $(i, j) \in R \Leftrightarrow M[i][j] \neq 0$.

For an investigation of the kbs of a matrix see [Egn97].

```
gap> M := [[1,0,1,0], [0,2,0,3], [1,0,3,0], [0,4,0,1]];
[ [ 1, 0, 1, 0 ], [ 0, 2, 0, 3 ], [ 1, 0, 3, 0 ], [ 0, 4, 0, 1 ] ]
```

```
gap> PrintArray(M);
[ [  1,  0,  1,  0 ],
  [  0,  2,  0,  3 ],
  [  1,  0,  3,  0 ],
  [  0,  4,  0,  1 ] ]
gap> kbs(M);
[ [ 1, 3 ], [ 2, 4 ] ]
```

`kbs(` *list* `)`

`kbs` returns the joined kbs of the matrices in *list*. The matrices in *list* must have common size otherwise an error is signaled.

1.170 DirectSumPerm

`DirectSumPerm(` *list1* , *list2* `)`

Let *list2* contain permutations and *list1* be of the same length and contain degrees equal or larger than the corresponding largest moved points. `DirectSumPerm` returns the direct sum of the permutations defined via the direct sum of the corresponding matrices.

```
gap> DirectSumPerm( [3, 3], [(1,2), (1,2,3)] );
(1,2)(4,5,6)
```

1.171 TensorProductPerm

`TensorProductPerm(` *list1* , *list2* `)`

Let *list2* contain permutations and *list1* be of the same length and contain degrees equal or larger than the corresponding largest moved points. `TensorProductPerm` returns the tensor product (Kronecker product) of the permutations defined via the tensor product of the corresponding matrices.

```
gap> TensorProductPerm( [3, 3], [(1,2), (1,2,3)] );
(1,5,3,4,2,6)(7,8,9)
```

1.172 MovedPointsPerm

`MovedPointsPerm(` *p* `)`

`MovedPointsPerm` returns the set of the moved points of the permutation *p*.

```
gap> MovedPointsPerm( (1,7)(2,3,8) );
[ 1, 2, 3, 7, 8 ]
```

1.173 NrMovedPointsPerm

`MovedPointsPerm(` *p* `)`

`MovedPointsPerm` returns the number of the moved points of the permutation *p*.

```
gap> NrMovedPointsPerm( (1,7)(2,3,8) );
 5
```

Bibliography

[CB93] M. Clausen and U. Baum. *Fast Fourier Transforms.* BI-Wissenschaftsverlag, Mannheim, 1993.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine computation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[Egn97] S. Egner. *Zur Algorithmischen Zerlegungstheorie linearer Transformationen mit Symmetrie.* PhD thesis, Universität Karlsruhe, 1997.

[ER82] Douglas F. Elliott and K. Ramamohan Rao. *Fast Transforms — Algorithms, Analyses, Applications.* Academic Press, 1982.

[Leo91] S. Jeffrey Leon. Permutation Group Algorithms Based on Partitions, I: Theory and Algorithms . *Journal of Symbolic Computation*, 12:533–583, 1991.

[Mal92] Henrique S. Malvar. *Signal Processing with Lapped Transforms.* Artech House, 1992.

[Mer96] A. Mertins. *Signaltheorie.* Teubner Verlag, 1996.

[Min93] T. Minkwitz. *Algorithmensynthese für lineare Systeme mit Symmetrie.* PhD thesis, Universität Karlsruhe, 1993.

[Min95] T. Minkwitz. Algorithms Explained by Symmetry. *Lecture Notes on Computer Science*, 900:157–167, 1995.

[Min96] T. Minkwitz. Extension of Irreducible Representations. *Applicable Algebra in Engineering, Communication and Computing*, 7:391–399, 1996.

[Püs98] M. Püschel. *Konstruktive Darstellungstheorie und Algorithmengenerierung.* PhD thesis, Universität Karlsruhe, 1998.

[Rad68] Charles M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56:1107–1108, 1968.

Index

Index