RepnDecomp

Decompose representations of finite groups into irreducibles

1.2.1

2 March 2022

Kaashif Hymabaccus

Kaashif Hymabaccus
Email: `kaashif@kaashif.co.uk`
Homepage: `https://kaashif.co.uk`

Contents

1 Introduction **3**
- 1.1 Getting started with RepnDecomp . . . 3

2 Isomorphisms between representations **5**
- 2.1 Finding explicit isomorphisms . . . 5
- 2.2 Testing isomorphisms . . . 6

3 Algorithms for unitary representations **8**
- 3.1 Unitarising representations . . . 8
- 3.2 Decomposing unitary representations . . . 9

4 Miscellaneous useful functions **10**
- 4.1 Predicates for representations . . . 10
- 4.2 Efficient summing over groups . . . 10
- 4.3 Space-efficient representation of tensors of matrices . . . 11
- 4.4 Matrices and homomorphisms . . . 11
- 4.5 Representation theoretic functions . . . 12

5 Computing decompositions of representations **13**
- 5.1 Block diagonalizing . . . 13
- 5.2 Algorithms due to the authors . . . 13
- 5.3 Algorithms due to Serre . . . 15

6 Centralizer (commutant) rings **18**
- 6.1 Finding a basis for the centralizer . . . 18
- 6.2 Using the centralizer for computations . . . 19

Index **20**

Chapter 1

Introduction

1.1 Getting started with RepnDecomp

This package allows computations of various decompositions of a representation $\rho : G \to GL(V)$ where G is finite and V is a finite-dimensional $\mathbb{C}$-vector space.

1.1.1 Installation

To install this package, refer to the installation instructions in the README file in the source code. It is located here: `https://github.com/gap-packages/RepnDecomp/blob/master/README.md`.

1.1.2 Note on what is meant by a representation

Throughout this documentation, mathematical terminology is used e.g. representation. It is clear what is meant mathematically, but it is not entirely clear what is meant in terms of GAP types - what are you supposed to pass in when I say "pass in a representation". Occasionally I will not even mention what we are passing in and assume the reader knows that *rho* or ρ refers to a representation. A representation we can use is, in GAP, a homomorphism from a finite group to a matrix group where all matrices have coefficients in a cyclotomic field (`Cyclotomics` is the union of all such fields in GAP). You can check whether something you want to pass is suitable with the function `IsFiniteGroupLinearRepresentation` (4.1.1).

Here's an example of a representation *rho* in GAP:

Example
```
gap> G := SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
gap> images := List(GeneratorsOfGroup(G), g -> PermutationMat(g, 3));
[ [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ],
  [ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ] ]
gap> rho := GroupHomomorphismByImages(G, Group(images));
[ (1,2,3), (1,2) ] -> [ [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ],
  [ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ] ]
```

1.1.3 API Overview

The algorithms implemented can be divided into two groups: methods due to Serre from his book Linear Representations of Finite Groups, and original methods due to the authors of this package.

The default is to use the algorithms due to Serre. If you pass the option `method := "alternate"` to a function, it will use the alternate method. Passing the option `parallel` will try to compute in parallel as much as possible. See the individual functions for options you can pass.

The main functions implemented in this package are:

For decomposing representations into canonical and irreducible direct summands:

- `CanonicalDecomposition` (5.3.1)
- `IrreducibleDecomposition` (5.3.2)
- `IrreducibleDecompositionCollected` (5.3.3)

For block diagonalising representations:

- `BlockDiagonalBasisOfRepresentation` (5.1.1)
- `BlockDiagonalRepresentation` (5.1.2)

For computing centraliser rings:

- `CentralizerBlocksOfRepresentation` (6.1.1)
- `CentralizerOfRepresentation` (6.1.2)

For testing isomorphism and computing isomorphisms (intertwining operators) between representations:

- `LinearRepresentationIsomorphism` (2.1.1)
- `AreRepsIsomorphic` (2.2.1)
- `IsLinearRepresentationIsomorphism` (2.2.2)

For testing unitarity of representations and the unitarisation of representations:

- `UnitaryRepresentation` (3.1.1)
- `IsUnitaryRepresentation` (3.1.2)

Chapter 2

Isomorphisms between representations

2.1 Finding explicit isomorphisms

2.1.1 LinearRepresentationIsomorphism

▷ `LinearRepresentationIsomorphism(`*`rho, tau[, rho_cent_basis, tau_cent_basis]`*`)`

(function)

Returns: A matrix A or fail

Let $\rho : G \to GL(V)$ and $\tau : G \to GL(W)$. If there exists a linear map $A : V \to W$ such that for all $g \in G$, $\tau(g)A = A\rho(g)$, this function returns one such A. A is the isomorphism between the representations. If the representations are not isomorphic, then fail is returned.

There are three methods that we can use to compute an isomorphism of linear representations, you can select one by passing options to the function.

- `use_kronecker`: Assumes the matrices are small enough that their Kronecker products can fit into memory. Uses `GroupSumBSGS` (4.2.1) and `KroneckerProduct` to compute an element of the fixed subspace of $\rho \otimes \tau^*$.

- `use_orbit_sum`: Finds an isomorphism by summing orbits of the the action of $\rho \otimes \tau^*$ on matrices. Note that orbits could be very large, so this could be as bad as summing over the whole group.

- The default, sums over the whole group to compute the projection onto the fixed subspace.

Example
```
gap> G := SymmetricGroup(4);;
gap> irreps := IrreducibleRepresentations(G);;
gap> # rho and tau are isomorphic - they just have a different block order
> rho := DirectSumOfRepresentations([irreps[1], irreps[3], irreps[3]]);;
gap> tau := DirectSumOfRepresentations([irreps[3], irreps[1], irreps[3]]);;
gap> # tau2 is just tau with a basis change - still isomorphic
> B := RandomInvertibleMat(5);;
gap> tau2 := ComposeHomFunction(tau, x -> B^-1 * x * B);;
gap> # using the default implementation
> M := LinearRepresentationIsomorphism(rho, tau);;
gap> IsLinearRepresentationIsomorphism(M, rho, tau);
true
gap> M := LinearRepresentationIsomorphism(tau, tau2);;
gap> IsLinearRepresentationIsomorphism(M, tau, tau2);
true
gap> # using the kronecker sum implementation
> M := LinearRepresentationIsomorphism(tau, tau2 : use_kronecker);;
gap> IsLinearRepresentationIsomorphism(M, tau, tau2);
true
gap> # using the orbit sum implementation
> M := LinearRepresentationIsomorphism(tau, tau2 : use_orbit_sum);;
gap> IsLinearRepresentationIsomorphism(M, tau, tau2);
true
gap> # two distinct irreps are not isomorphic
> M := LinearRepresentationIsomorphism(irreps[1], irreps[2]);
fail
```

2.1.2 LinearRepresentationIsomorphismSlow

▷ `LinearRepresentationIsomorphismSlow(`*`rho, tau`*`)` (function)

Returns: A matrix *A* or fail

Gives the same result as `LinearRepresentationIsomorphism` (2.1.1), but this function uses a simpler method which always involves summing over *G*, without using `GroupSumBSGS` (4.2.1). This might be useful in some cases if computing a good BSGS is difficult. However, for all cases that have been tested, it is slow (as the name suggests).

Example
```
gap> # Following on from the previous example
> M := LinearRepresentationIsomorphismSlow(rho, tau);;
gap> IsLinearRepresentationIsomorphism(M, rho, tau);
true
```

2.2 Testing isomorphisms

2.2.1 AreRepsIsomorphic

▷ `AreRepsIsomorphic(`*`rho, tau`*`)` (function)

Returns: true if *`rho`* and *`tau`* are isomorphic as representations, false otherwise.

Since representations of finite groups over $\mathbb{C}$ are determined by their characters, it is easy to check whether two representations are isomorphic by checking if they have the same character. We try to use characters wherever possible.

Example
```
gap> # Following on from the previous examples
> # Some isomorphic representations
> AreRepsIsomorphic(rho, tau);
true
gap> AreRepsIsomorphic(rho, tau2);
true
gap> # rho isn't iso to irreps[1] since rho is irreps[1] plus some other stuff
> AreRepsIsomorphic(rho, irreps[1]);
false
```

2.2.2 IsLinearRepresentationIsomorphism

▷ IsLinearRepresentationIsomorphism(*A, rho, tau*) (function)

Returns: true if *rho* and *tau* are isomorphic as as representations with the isomorphism given by the linear map *A*

This function tests if, for all $g \in G$, $A\rho(g) = \tau(g)A$. That is, true is returned iff A is the intertwining operator taking ρ to τ. that:

Example
```
gap> # We have already seen this function used heavily in previous examples.
> # If two representations are isomorphic, the following is always true:
> IsLinearRepresentationIsomorphism(LinearRepresentationIsomorphism(rho, tau), rho, tau);
true
gap> # Note: this test is sensitive to ordering:
> IsLinearRepresentationIsomorphism(LinearRepresentationIsomorphism(rho, tau), tau, rho);
false
```

Chapter 3

Algorithms for unitary representations

3.1 Unitarising representations

3.1.1 UnitaryRepresentation

▷ `UnitaryRepresentation(`*`rho`*`)` (function)

Returns: A record with fields basis_change and unitary_rep such that *`rho`* is isomorphic to unitary_rep, differing by a change of basis basis_change. Meaning if L is basis_change and ρ_u is the unitarised *`rho`*, then $\forall g \in G:\ L\rho_u(g)L^{-1} = \rho(g)$.

Unitarises the given representation quickly, summing over the group using a base and strong generating set and unitarising with `LDLDecomposition` (3.1.3).

Example
```
gap> G := SymmetricGroup(3);;
gap> irreps := IrreducibleRepresentations(G);;
gap> # It happens that we are given unitary irreps, so
> # rho is also unitary (its blocks are unitary)
> rho := DirectSumOfRepresentations([irreps[1], irreps[2]]);;
gap> IsUnitaryRepresentation(rho);
true
gap> # Arbitrary change of basis
> A := [ [ -1, 1 ], [ -2, -1 ] ];;
gap> tau := ComposeHomFunction(rho, x -> A^-1 * x * A);;
gap> # Not unitary, but still isomorphic to rho
> IsUnitaryRepresentation(tau);
false
gap> AreRepsIsomorphic(rho, tau);
true
gap> # Now we unitarise tau
> tau_u := UnitaryRepresentation(tau);;
gap> # We get a record with the unitarised rep:
> AreRepsIsomorphic(tau, tau_u.unitary_rep);
true
gap> AreRepsIsomorphic(rho, tau_u.unitary_rep);
true
gap> # The basis change is also in the record:
> ForAll(G, g -> tau_u.basis_change * Image(tau_u.unitary_rep, g) = Image(tau, g) * tau_u.basis_c
true
```

3.1.2 IsUnitaryRepresentation

▷ `IsUnitaryRepresentation(`*`rho`*`)` (function)

Returns: Whether *rho* is unitary, i.e. for all $g \in G$, $\rho(g^{-1}) = \rho(g)^*$ (where * denotes the conjugate transpose).

Example
```
gap> # TODO: this example
```

3.1.3 LDLDecomposition

▷ `LDLDecomposition(`*`A`*`)` (function)

Returns: a record with two fields, L and D such that $A = L\mathrm{diag}(D)L^*$. D is the $1 \times n$ vector which gives the diagonal matrix $\mathrm{diag}(D)$ (where *A* is an $n \times n$ matrix).

Example
```
gap> A := [ [ 3, 2*E(3)+E(3)^2, -3 ], [ E(3)+2*E(3)^2, -3, 3 ], [ -3, 3, -6 ] ];;
gap> # A is a conjugate symmetric matrix
> A = ConjugateTranspose@RepnDecomp(A);
true
gap> # Note that A is not symmetric - the LDL decomposition works for any
> # conjugate symmetric matrix.
> A = TransposedMat(A);
false
gap> decomp := LDLDecomposition(A);;
gap> # The LDL decomposition is such that A = LDL^*, D diagonal, and L lower triangular.
> A = decomp.L * DiagonalMat(decomp.D) * ConjugateTranspose@RepnDecomp(decomp.L);
true
gap> decomp.L[1][2] = 0 and decomp.L[1][3] = 0 and decomp.L[2][3] = 0;
true
```

3.2 Decomposing unitary representations

3.2.1 IrreducibleDecompositionDixon

▷ `IrreducibleDecompositionDixon(`*`rho`*`)` (function)

Returns: a list of irreps in the decomposition of *rho*

NOTE: this is not implemented yet. Assumes that *rho* is unitary and uses an algorithm due to Dixon to decompose it into unitary irreps.

Chapter 4

Miscellaneous useful functions

4.1 Predicates for representations

4.1.1 IsFiniteGroupLinearRepresentation (for IsGroupHomomorphism)

▷ `IsFiniteGroupLinearRepresentation(`*`rho`*`)` (attribute)

Returns: true or false

Tells you if *`rho`* is a linear representation of a finite group. The algorithms implemented in this package work on these homomorphisms only.

4.1.2 IsFiniteGroupPermutationRepresentation (for IsGroupHomomorphism)

▷ `IsFiniteGroupPermutationRepresentation(`*`rho`*`)` (attribute)

Returns: true or false

Tells you if *`rho`* is a homomorphism from a finite group to a permutation group.

4.2 Efficient summing over groups

4.2.1 GroupSumBSGS

▷ `GroupSumBSGS(`*`G, summand`*`)` (function)

Returns: $\sum_{g \in G} \mathrm{summand}(g)$

Uses a basic stabiliser chain for G to compute the sum described above. This trick requires *`summand`* to be a function (in the GAP sense) that defines a monoid homomorphism (in the mathematical sense). The computation of the stabiliser chain assumes *`G`* is a group. More precisely, if we have the basic stabiliser chain:

$$\{1\} = G_1 \leq \ldots \leq G_n = G$$

We traverse the chain from G_1 to G_n, using the previous sum G_{i-1} to build the sum G_i. We do this by using the fact that (writing f for *`summand`*)

$$\sum_{g \in G_i} f(g) = \sum_{r_j} \left(\sum_{h \in G_{i-1}} f(h) \right) f(r_j)$$

where the r_j are right coset representatives of G_{i-1} in G_i. The condition on *`summand`* is satisfied if, for example, it is a linear representation of a group *`G`*.

4.3 Space-efficient representation of tensors of matrices

Suppose we have representations of G, ρ and τ, with degree n and m. If we would like to construct the tensor product representation of G, $\rho \otimes \tau$, the usual way to do it would be to take the Kronecker product of the matrices. This means we now have to store very large $nm \times nm$ matrices for each generator of G. This can be avoided by storing the tensor of matrices as pairs, essentially storing $A \otimes B$ as a pair (A,B) and implementing group operations on top of these, along with some representation-theoretic functions. It is only possible to guarantee an economical representation for pure tensors, i.e. matrices of the form $A \otimes B$. These are closed under group operations, so it is natural to define a group structure.

4.3.1 IsTensorProductOfMatricesObj (for IsMultiplicativeElementWithInverse)

▷ `IsTensorProductOfMatricesObj(`*arg*`)` (filter)

Returns: `true` or `false`

Position i in this representation stores the matrix A_i in the tensor product $A_1 \otimes A_2$.

4.3.2 IsTensorProductPairRep (for IsPositionalObjectRep)

▷ `IsTensorProductPairRep(`*arg*`)` (filter)

Returns: `true` or `false`

Position 1 stores the full Kronecker product of the matrices, this is very space inefficient and supposed to be used as a last resort.

4.3.3 IsTensorProductKroneckerRep (for IsPositionalObjectRep)

▷ `IsTensorProductKroneckerRep(`*arg*`)` (filter)

Returns: `true` or `false`

More convenient constructor for a tensor product (automatically handles family)

4.3.4 TensorProductOfMatrices

▷ `TensorProductOfMatrices(`*arg*`)` (function)

This uses the multiplicativity of characters when taking tensor products to avoid having to compute the trace of a big matrix.

4.3.5 CharacterOfTensorProductOfRepresentations

▷ `CharacterOfTensorProductOfRepresentations(`*arg*`)` (function)

4.4 Matrices and homomorphisms

4.4.1 ComposeHomFunction

▷ `ComposeHomFunction(`*hom, func*`)` (function)

Returns: Homomorphism g given by g(x) = func(hom(x)).

This is mainly for convenience, since it handles all GAP accounting issues regarding the range, ByImages vs ByFunction, etc.

4.5 Representation theoretic functions

4.5.1 TensorProductRepLists

▷ `TensorProductRepLists(list1, list2)` (function)

Returns: All possible tensor products given by $\rho \otimes \tau$ where $\rho : G \to \mathrm{GL}(V)$ is taken from `list1` and $\tau : H \to \mathrm{GL}(W)$ is taken from `list2`. The result is then a list of representations of $G \times H$.

4.5.2 DirectSumOfRepresentations

▷ `DirectSumOfRepresentations(list)` (function)

Returns: Direct sum of the list of representations `list`

4.5.3 DegreeOfRepresentation

▷ `DegreeOfRepresentation(rho)` (function)

Returns: Degree of the representation `rho`. That is, $\mathrm{Tr}(\rho(e_G))$, where e_G is the identity of the group G that `rho` has as domain.

4.5.4 PermToLinearRep

▷ `PermToLinearRep(rho)` (function)

Returns: Linear representation ρ isomorphic to the permutation representation `rho`.

4.5.5 IsOrthonormalSet

▷ `IsOrthonormalSet(S, prod)` (function)

Returns: Whether `S` is an orthonormal set with respect to the inner product `prod`.

Chapter 5

Computing decompositions of representations

5.1 Block diagonalizing

Given a representation $\rho : G \to GL(V)$, it is often desirable to find a basis for V that block diagonalizes each $\rho(g)$ with the block sizes being as small as possible. This speeds up matrix algebra operations, since they can now be done block-wise.

5.1.1 BlockDiagonalBasisOfRepresentation

▷ `BlockDiagonalBasisOfRepresentation(`*rho*`)` (function)

Returns: Basis for V that block diagonalizes ρ.

Let G have irreducible representations ρ_i, with dimension d_i and multiplicity m_i. The basis returned by this operation gives each $\rho(g)$ as a block diagonal matrix which has m_i blocks of size $d_i \times d_i$ for each i.

5.1.2 BlockDiagonalRepresentation

▷ `BlockDiagonalRepresentation(`*rho*`)` (function)

Returns: Representation of G isomorphic to ρ where the images $\rho(g)$ are block diagonalized.

This is just a convenience operation that uses `BlockDiagonalBasisOfRepresentation` (5.1.1) to calculate the basis change matrix and applies it to put ρ into the block diagonalised form.

5.2 Algorithms due to the authors

5.2.1 REPN_ComputeUsingMyMethod (for IsGroupHomomorphism)

▷ `REPN_ComputeUsingMyMethod(`*rho*`)` (attribute)

Returns: A record in the same format as `REPN_ComputeUsingSerre` (5.3.4)

Computes the same values as `REPN_ComputeUsingSerre` (5.3.4), taking the same options. The heavy lifting of this method is done by `LinearRepresentationIsomorphism` (2.1.1), where there are some further options that can be passed to influence algorithms used.

```
                                    Example
gap> G := SymmetricGroup(4);;
gap> irreps := IrreducibleRepresentations(G);;
gap> rho := DirectSumOfRepresentations([irreps[4], irreps[5]]);;
gap> # Jumble rho up a bit so it's not so easy for the library.
> A := [ [ 3, -3, 2, -4, 0, 0 ], [ 4, 0, 1, -5, 1, 0 ], [ -3, -1, -2, 4, -1, -2 ],
>        [ 4, -4, -1, 5, -3, -1 ], [ 3, -2, 1, 0, 0, 0 ], [ 4, 2, 4, -1, -2, 1 ] ];;
gap> rho := ComposeHomFunction(rho, B -> A^-1 * B * A);;
gap> # We've constructed rho from two degree 3 irreps, so there are a few
> # things we can check for correctness:
> decomposition := REPN_ComputeUsingMyMethod(rho);;
gap> # Two distinct irreps, so the centralizer has dimension 2
> Length(decomposition.centralizer_basis) = 2;
true
gap> # Two distinct irreps i.e. two invariant subspaces
> Length(decomposition.decomposition) = 2;
true
gap> # All subspaces are dimension 3
> ForAll(decomposition.decomposition, Vs -> Length(Vs) = 1 and Dimension(Vs[1]) = 3);
true
gap> # And finally, check that the block diagonalized representation
> # computed is actually isomorphic to rho:
> AreRepsIsomorphic(rho, decomposition.diagonal_rep);
true
```

5.2.2 REPN_ComputeUsingMyMethodCanonical (for IsGroupHomomorphism)

▷ REPN_ComputeUsingMyMethodCanonical(*rho*) (attribute)

Returns: A record in the same format as REPN_ComputeUsingMyMethod (5.2.1).

Performs the same computation as REPN_ComputeUsingMyMethod (5.2.1), but first splits the representation into canonical summands using CanonicalDecomposition (5.3.1). This might reduce the size of the matrices we need to work with significantly, so could be much faster.

If the option parallel is given, the decomposition of canonical summands into irreps is done in parallel, which could be much faster.

```
                                    Example
gap> # This is the same example as before, but splits into canonical
> # summands internally. It gives exactly the same results, up to
> # isomorphism.
> other_decomposition := REPN_ComputeUsingMyMethodCanonical(rho);;
gap> Length(other_decomposition.centralizer_basis) = 2;
true
gap> Length(other_decomposition.decomposition) = 2;
true
gap> ForAll(other_decomposition.decomposition, Vs -> Length(Vs) = 1 and Dimension(Vs[1]) = 3);
true
gap> AreRepsIsomorphic(rho, other_decomposition.diagonal_rep);
true
```

5.3 Algorithms due to Serre

Note: all computation in this section is actually done in the function `REPN_ComputeUsingSerre` (5.3.4), the other functions are wrappers around it.

5.3.1 CanonicalDecomposition

▷ `CanonicalDecomposition(rho)` (function)

Returns: List of vector spaces V_i, each G-invariant and a direct sum of isomorphic irreducibles. That is, for each i, $V_i \cong \oplus_j W_i$ (as representations) where W_i is an irreducible G-invariant vector space.

Computes the canonical decomposition of V into $\oplus_i V_i$ using the formulas for projections $V \to V_i$ due to Serre. You can pass in the option `irreps` with a list of irreps of G, and this will be used instead of computing a complete list ourselves. If you already know which irreps will appear in ρ, for instance, this will save time.

Example
```
gap> # This is the trivial group
> G := Group(());;
gap> # The trivial group has only one representation per degree, so a
> # degree d representation decomposes into a single canonical summand
> # containing the whole space
> rho := FuncToHom@RepnDecomp(G, g -> IdentityMat(3));;
gap> canonical_summands_G := CanonicalDecomposition(rho);
[ ( Cyclotomics^3 ) ]
gap> # More interesting example, S_3
> H := SymmetricGroup(3);;
gap> # The standard representation: a permutation to the corresponding
> # permutation matrix.
> tau := FuncToHom@RepnDecomp(H, h -> PermutationMat(h, 3));;
gap> # Two canonical summands corresponding to the degree 2 and
> # trivial irreps (in that order)
> List(CanonicalDecomposition(tau), Dimension);
[ 2, 1 ]
```

5.3.2 IrreducibleDecomposition

▷ `IrreducibleDecomposition(rho)` (function)

Returns: List of vector spaces W_j such that $V = \oplus_j W_j$ and each W_j is an irreducible G-invariant vector space.

Computes the decomposition of V into irreducible subprepresentations.

Example
```
gap> # The trivial group has 1 irrep of degree 1, so rho decomposes into 3
> # lines.
> irred_decomp_G := IrreducibleDecomposition(rho);
[ rec( basis := [ [ 1, 0, 0 ] ] ), rec( basis := [ [ 0, 1, 0 ] ] ),
  rec( basis := [ [ 0, 0, 1 ] ] ) ]
gap> # The spaces are returned in this format - explicitly keeping the
> # basis - since this basis block diagonalises rho into the irreps,
> # which are the smallest possible blocks. This is more obvious with
> # H.
> irred_decomp_H := IrreducibleDecomposition(tau);
```

```
[ rec( basis := [ [ 1, 1, 1 ] ] ),
  rec( basis := [ [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ] ) ]
gap> # Using the basis vectors given there block diagonalises tau into
> # the two blocks corresponding to the two irreps:
> nice_basis := [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ];;
gap> tau_diag := ComposeHomFunction(tau, X -> nice_basis^-1 * X * nice_basis);
[ (1,2,3), (1,2) ] -> [ [ [ 1, 0, 0 ], [ 0, E(3), 0 ], [ 0, 0, E(3)^2 ] ],
  [ [ 1, 0, 0 ], [ 0, 0, E(3)^2 ], [ 0, E(3), 0 ] ] ]
```

5.3.3 IrreducibleDecompositionCollected

▷ `IrreducibleDecompositionCollected(`*`rho`*`)` (function)

Returns: List of lists V_i of vector spaces V_{ij} such that $V = \bigoplus_i \bigoplus_j V_{ij}$ and $V_{ik} \cong V_{il}$ for all i, k and l (as representations).

Computes the decomposition of V into irreducible subrepresentations, grouping together the isomorphic subrepresentations.

5.3.4 REPN_ComputeUsingSerre (for IsGroupHomomorphism)

▷ `REPN_ComputeUsingSerre(`*`rho`*`)` (attribute)

Returns: A record, in the format described below

This function does all of the computation and (since it is an attribute) saves the results. Doing all of the calculations at the same time ensures consistency when it comes to irrep ordering, block ordering and basis ordering. There is no canonical ordering of irreps, so this is crucial.

`irreps` is the complete list of irreps involved in the direct sum decomposition of *`rho`*, this can be given in case the default (running Dixon's algorithm) is too expensive, or e.g. you don't want representations over Cyclotomics.

The return value of this function is a record with fields:

- `basis`: The basis that block diagonalises ρ, see `BlockDiagonalBasisOfRepresentation` (5.1.1).
- `diagonal_rep`: ρ, block diagonalised with the basis above. See `BlockDiagonalRepresentation` (5.1.2)
- `decomposition`: The irreducible G-invariant subspaces, collected according to isomorphism, see `IrreducibleDecompositionCollected` (5.3.3)
- `centralizer_basis`: An orthonormal basis for the centralizer ring of ρ, written in block form. See `CentralizerBlocksOfRepresentation` (6.1.1)

Pass the option `parallel` for the computations per-irrep to be done in parallel.

Pass the option `irreps` with the complete list of irreps of ρ to avoid recomputing this list (could be very expensive)

Example
```
gap> # Does the same thing we have done in the examples above, but all in
> # one step, with as many subcomputations reused as possible
> REPN_ComputeUsingSerre(tau);
rec( basis := [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ],
  centralizer_basis := [ [ [ [ 1 ] ], [ [ 0, 0 ], [ 0, 0 ] ] ],
      [ [ [ 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ] ],
  decomposition := [ [ rec( basis := [ [ 1, 1, 1 ] ] ) ], [  ],
      [ rec( basis := [ [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ] ) ] ],
  diagonal_rep := [ (1,2,3), (1,2) ] ->
    [ [ [ 1, 0, 0 ], [ 0, E(3), 0 ], [ 0, 0, E(3)^2 ] ],
      [ [ 1, 0, 0 ], [ 0, 0, E(3)^2 ], [ 0, E(3), 0 ] ] ] )
gap> # You can also do the computation in parallel:
> REPN_ComputeUsingSerre(tau : parallel);;
gap> # Or specify the irreps if you have already computed them:
> irreps_H := IrreducibleRepresentations(H);;
gap> REPN_ComputeUsingSerre(tau : irreps := irreps_H);;
```

Chapter 6

Centralizer (commutant) rings

6.1 Finding a basis for the centralizer

6.1.1 CentralizerBlocksOfRepresentation

▷ `CentralizerBlocksOfRepresentation(rho)` (function)

Returns: List of vector space generators for the centralizer ring of $\rho(G)$, written in the basis given by `BlockDiagonalBasisOfRepresentation` (5.1.1). The matrices are given as a list of blocks.

Let G have irreducible representations ρ_i with multiplicities m_i. The centralizer has dimension $\sum_i m_i^2$ as a $\mathbb{C}$-vector space. This function gives the minimal number of generators required.

Example
```
gap> G := DihedralGroup(8);;
gap> irreps := IrreducibleRepresentations(G);;
gap> # rho is the sum of two isomorphic degree 1 irreps, and a degree
> # 2 irrep.
> rho := DirectSumOfRepresentations([irreps[4], irreps[4], irreps[5]]);;
gap> # Compute a basis for the centralizer (in blocks)
> cent_basis_blocks := CentralizerBlocksOfRepresentation(rho);;
gap> # Verify that the dimension is the sum of the multiplicities squared,
> # in this case 2^2 + 1 = 5.
> Length(cent_basis_blocks) = 5;
true
```

6.1.2 CentralizerOfRepresentation

▷ `CentralizerOfRepresentation(arg)` (function)

Returns: List of vector space generators for the centralizer ring of $\rho(G)$.

This gives the same result as `CentralizerBlocksOfRepresentation` (6.1.1), but with the matrices given in their entirety: not as lists of blocks, but as full matrices.

Example
```
gap> # This is the actual basis for the centralizer.
> cent_basis := CentralizerOfRepresentation(rho);;
gap> # All matrices in the span should commute with the image of rho.
> ForAll(G, g -> ForAll(cent_basis, M -> Image(rho, g)*M = M*Image(rho,g)));
true
```

6.2 Using the centralizer for computations

6.2.1 ClassSumCentralizer

▷ `ClassSumCentralizer(rho, class, cent_basis)` (function)

Returns: $\sum_{s \in t^G} \rho(s)$, where t is a representative of the conjugacy class `class` of G.

We require that `rho` is unitary. Uses the given orthonormal basis (with respect to the inner product $\langle A, B \rangle = \mathrm{Trace}(AB^*)$) for the centralizer ring of `rho` to calculate the sum of the conjugacy class `class` quickly, i.e. without summing over the class.

NOTE: Orthonormality of `cent_basis` and unitarity of `rho` are checked. See `ClassSumCentralizerNC` (6.2.2) for a version of this function without checks. The checks are not very expensive, so it is recommended you use the function with checks.

Example
```
gap> # Now we have a basis for the centralizer, we can sum a conjugacy class
> # of G.
> class := List(ConjugacyClasses(G)[3]);;
gap> # We can do the computation naively, with no centralizer basis given:
> sum1 := ClassSumCentralizer(rho, class, fail);;
gap> # Before summing with th centralizer basis given, we need to
> # orthonormalize it. It's already orthogonal, but not normal:
> orth_basis := OrthonormalBasis@RepnDecomp(cent_basis);;
gap> IsOrthonormalSet(orth_basis, InnerProduct@RepnDecomp);
true
gap> # And with the centralizer given, should be more efficient in certain
> # cases (small degree, low multiplicities, but very large group)
> sum2 := ClassSumCentralizer(rho, class, orth_basis);;
gap> # Should be the same:
> sum1 = sum2;
true
```

6.2.2 ClassSumCentralizerNC

▷ `ClassSumCentralizerNC(rho, class, cent_basis)` (function)

The same as `ClassSumCentralizer` (6.2.1), but does not check the basis for orthonormality or the representation for unitarity.

Example
```
gap> # The very same as the above, but with no checks on orthonormality.
> sum3 := ClassSumCentralizerNC(rho, class, orth_basis);;
gap> sum1 = sum3;
true
```

Index

AreRepsIsomorphic, 6

BlockDiagonalBasisOfRepresentation, 13
BlockDiagonalRepresentation, 13

CanonicalDecomposition, 15
CentralizerBlocksOfRepresentation, 18
CentralizerOfRepresentation, 18
CharacterOfTensorProductOf-Representations, 11
ClassSumCentralizer, 19
ClassSumCentralizerNC, 19
ComposeHomFunction, 11

DegreeOfRepresentation, 12
DirectSumOfRepresentations, 12

GroupSumBSGS, 10

IrreducibleDecomposition, 15
IrreducibleDecompositionCollected, 16
IrreducibleDecompositionDixon, 9
IsFiniteGroupLinearRepresentation
  for IsGroupHomomorphism, 10
IsFiniteGroupPermutationRepresentation
  for IsGroupHomomorphism, 10
IsLinearRepresentationIsomorphism, 7
IsOrthonormalSet, 12
IsTensorProductKroneckerRep
  for IsPositionalObjectRep, 11
IsTensorProductOfMatricesObj
  for IsMultiplicativeElementWithInverse, 11
IsTensorProductPairRep
  for IsPositionalObjectRep, 11
IsUnitaryRepresentation, 9

LDLDecomposition, 9
LinearRepresentationIsomorphism, 5
LinearRepresentationIsomorphismSlow, 6

PermToLinearRep, 12

REPN_ComputeUsingMyMethod
  for IsGroupHomomorphism, 13
REPN_ComputeUsingMyMethodCanonical
  for IsGroupHomomorphism, 14
REPN_ComputeUsingSerre
  for IsGroupHomomorphism, 16

TensorProductOfMatrices, 11
TensorProductRepLists, 12

UnitaryRepresentation, 8