Contents

Chapter 1

Resolutions of the ground ring

::::::::::::::::::::::::
`TietzeReducedResolution(R)`

Inputs a $\mathbb{Z}G$-resolution R and returns a $\mathbb{Z}G$-resolution S which is obtained from R by applying "Tietze like operations" in each dimension. The hope is that S has fewer free generators than R.

::::::::::::::::::::::::
`ResolutionArithmeticGroup("PSL(4,Z)",n)`

Inputs a positive integer n and one of the following strings:

"SL(2,Z)" , "SL(3,Z)" , "PGL(3,Z[i])" , "PGL(3,Eisenstein_Integers)" , "PSL(4,Z)" , "PSL(4,Z)_b" , "PSL(4,Z)_c" , "PSL(4,Z)_d" , "Sp(4,Z)"

or one of the following strings

"SL(2,Z[sqrt(-2)])" , "SL(2,Z[sqrt(-7)])" , "SL(2,Z[sqrt(-11)])" , "SL(2,Z[sqrt(-19)])" , "SL(2,Z[sqrt(-43)])" , "SL(2,Z[sqrt(-67)])" , "SL(2,Z[sqrt(-163)])"

It returns n terms of a free ZG-resolution for the group G described by the string. (Subscripts _b , _c , _d denote alternative non-free ZG-resolutions for a given group G.)

Data for the first list of resolutions was provided provided by MATHIEU DUTOUR. Data for the second list was provided by ALEXANDER RAHM.

::::::::::::::::::::::::
`FreeGResolution(P,n)`
`FreeGResolution(P,n,p)`

Inputs a non-free ZG-resolution P with finite stabilizer groups, and a positive integer n. It returns a free ZG-resolution of length equal to the minimum of n and the length of P. If one requires only a mod p resolution then the prime p can be entered as an optional third argument.

The free resolution is returned without a contracting homotopy.

::::::::::::::::::::::::
`ResolutionGTree(P,n)`

Inputs a non-free ZG-resolution P of dimension 1 (i.e. a G-tree) with finite stabilizer groups, and a positive integer n. It returns a free ZG-resolution of length equal to n.

If P has a contracting homotopy then the free resolution is returned with a contracting homotopy.

This function was written by BUI ANH TUAN.

::::::::::::::::::::::::
`ResolutionSL2Z(p,n)`

Inputs positive integers m,n and returns n terms of a ZG-resolution for the group $G = SL(2,Z[1/m])$.

This function is joint work with BUI ANH TUAN.

::::::::::::::::::::::::
`ResolutionAbelianGroup(L,n)`
`ResolutionAbelianGroup(G,n)`

Inputs a list $L := [m_1, m_2, ..., m_d]$ of nonnegative integers, and a positive integer n. It returns n terms of a ZG-resolution for the abelian group $G = Z_L[1] + Z_L[2] + + Z_L[d]$.

If G is finite then the first argument can also be the abelian group G itself.

::::::::::::::::::::::::
`ResolutionAlmostCrystalGroup(G,n)`

Inputs a positive integer n and an almost crystallographic pcp group G. It returns n terms of a free ZG-resolution. (A group is almost crystallographic if it is nilpotent-by-finite and has no non-trivial finite normal subgroup. Such groups can be constructed using the ACLIB package.)

::::::::::::::::::::::::
`ResolutionAlmostCrystalQuotient(G,n,c)`

`ResolutionAlmostCrystalQuotient(G,n,c,false)`

An almost crystallographic group G is an extension of a finite group P by a nilpotent group T, and has no non-trivial finite normal subgroup. We define the relative lower central series by setting $T_1 = T$ and $T_{i+1} = [T_i, G]$.

This function inputs an almost crystallographic group G together with positive integers n and c. It returns n terms of a free ZQ-resolution R for the group $Q = G/T_c$.

In addition to the usual components, the resolution R has the component $R.quotientHomomorphism$ which gives the quotient homomorphism $G \longrightarrow Q$.

If a fourth optional variable is set equal to "false" then the function omits to test whether Q is finite and a "more canonical" resolution is constructed.

::::::::::::::::::::::::
`ResolutionArtinGroup(D,n)`

Inputs a Coxeter diagram D and an integer $n > 1$. It returns n terms of a free ZG-resolution R where G is the Artin monoid associated to D. It is conjectured that R is also a free resolution for the Artin group G. The conjecture is known to hold in certain cases.

$G = R.group$ is infinite and returned as a finitely presented group. The list $R.elts$ is a partial listing of the elements of G which grows as R is used. Initially $R.elts$ is empty and then, any time the boundary of a resolution generator is called, $R.elts$ is updated to include elements of G involved in the boundary.

The contracting homotopy on R has not yet been implemented! Furthermore, the group G is currently returned only as a finitely presented group (without any method for solving the word problem).

::::::::::::::::::::::::
`ResolutionAsphericalPresentation(F,R,n)`

Inputs a free group F, a set R of words in F which constitute an aspherical presentation for a group G, and a positive integer n. (Asphericity can be a difficult property to verify. The function $IsAspherical(F,R)$ could be of help.)

The function returns n terms of a free ZG-resolution R which has generators in dimensions < 3 only. No contracting homotopy on R will be returned.

::::::::::::::::::::::::
`ResolutionBieberbachGroup( G )`
`ResolutionBieberbachGroup( G, v )`

Inputs a torsion free crystallographic group G, also known as a Bieberbach group, represented using AffineCrystGroupOnRight as in the GAP package Cryst. It also optionally inputs a choice of

vector v in the euclidean space R^n on which G acts freely. The function returns $n+1$ terms of the free ZG-resolution of Z arising as the cellular chain complex of the tesselation of R^n by the Dirichlet-Voronoi fundamental domain determined by v.

This function is part of the HAPcryst package written by MARC ROEDER and thus requires the HAPcryst package to be loaded.

The function requires the use of Polymake software.

::::::::::::::::::::::::
```
ResolutionCoxeterGroup(D,n)
```

Inputs a Coxeter diagram D and an integer $n > 1$. It returns k terms of a free ZG-resolution R where G is the Coxeter group associated to D. Here k is the maximum of n and the number of vertices in the Coxeter diagram. At present the implementation is only for finite Coxeter groups and the group G is returned as a permutation group. The contracting homotopy on R has not yet been implemented!

::::::::::::::::::::::::
```
ResolutionDirectProduct(R,S)
ResolutionDirectProduct(R,S,"internal")
```

Inputs a ZG-resolution R and ZH-resolution S. It outputs a ZD-resolution for the direct product $D = GxH$.

If G and H lie in a common group K, and if they commute and have trivial intersection, then an optional third variable "internal" can be used. This will force D to be the subgroup GH in K.

::::::::::::::::::::::::
```
ResolutionExtension(g,R,S)
ResolutionExtension(g,R, S,"TestFiniteness")
ResolutionExtension(g,R,S,"NoTest",GmapE)
```

Inputs a surjective group homomorphism $g : E \longrightarrow G$ with kernel N. It also inputs a ZN-resolution R and a ZG-resolution S. It returns a ZE-resolution. The groups E and G can be infinite.

If an optional fourth argument is set equal to "TestFiniteness" then the groups N and G will be tested to see if they are finite. If they are finite then some speed saving routines will be invoked.

If the homomorphism g is such that the GAP function $PreImagesElement(g,x)$ doesn't work, then a function $GmapE()$ should be included as a fifth input. For any x in G this function should return an element $GmapE(x)$ in E which gets mapped onto x by g.

The contracting homotopy on the ZE-resolution has not yet been fully implemented for infinite groups!

::::::::::::::::::::::::
```
ResolutionFiniteDirectProduct(R,S)
ResolutionFiniteDirectProduct(R,S, "internal")
```

Inputs a ZG-resolution R and ZH-resolution S where G and H are finite groups. It outputs a ZD-resolution for the direct product $D = G \times H$.

If G and H lie in a common group K, and if they commute and have trivial intersection, then an optional third variable "internal" can be used. This will force D to be the subgroup GH in K.

::::::::::::::::::::::::
```
ResolutionFiniteExtension(gensE,gensG,R,n)
ResolutionFiniteExtension(gensE,gensG,R,n,true)
ResolutionFiniteExtension(gensE,gensG,R,n,false,S)
```

Inputs: a set *gensE* of generators for a finite group E; a set *gensG* equal to the image of *gensE* in a quotient group G of E; a ZG-resolution R up to dimension at least n; a positive integer n. It uses the $TwistedTensorProduct()$ construction to return n terms of a ZE-resolution.

The function has an optional fourth argument which, when set equal to "true", invokes tietze reductions in the construction of a resolution for the kernel of $E \longrightarrow G$.

If a ZN-resolution S is available, where N is the kernel of the quotient $E \longrightarrow G$, then this can be incorporated into the computations using an optional fifth argument.

::::::::::::::::::::::::
```
ResolutionFiniteGroup(gens,n)
ResolutionFiniteGroup(gens,n,true)
ResolutionFiniteGroup(gens,n,false,p)
ResolutionFiniteGroup(gens,n,false,0,"extendible")
```

Inputs a set *gens* of generators for a finite group G and a positive integer n. It outputs n terms of a ZG-resolution.

The function has an optional third argument which, when set equal to *true*, invokes tietze reductions in the construction of the resolution.

The function has an optional fourth argument which, when set equal to a prime p, records the fact that the resolution will only be used for mod p calculations. This could speed up subsequent constructions.

The function has an optional fifth argument which, when set equal to "extendible", returns a resolution whose length can be increased using the command R!.extend() .

::::::::::::::::::::::::
```
ResolutionFiniteSubgroup(R,K)
ResolutionFiniteSubgroup(R,gensG,gensK)
```

Inputs a ZG-resolution for a finite group G and a subgroup K of index $|G:K|$. It returns a free ZK-resolution whose ZK-rank is $|G:K|$ times the ZG-rank in each dimension.

Generating sets *gensG*, *gensK* for G and K can also be input to the function (though the method does not depend on a choice of generators).

This ZK-resolution is not reduced. ie. it has more than one generator in dimension 0.

::::::::::::::::::::::::
```
ResolutionGraphOfGroups(D,n)
ResolutionGraphOfGroups(D,n,L)
```

Inputs a graph of groups D and a positive integer n. It returns n terms of a free ZG-resolution for the fundamental group G of D.

An optional third argument $L = [R_1, \ldots, R_t]$ can be used to list (in any order) free resolutions for some/all of the vertex and edge groups in D. If for some vertex or edge group no resolution is listed in L then the function *ResolutionFiniteGroup*() will be used to try to construct the resolution.

The ZG-resolution is usually not reduced. i.e. it has more than one generator in dimension 0.

The contracting homotopy on the ZG-resolution has not yet been implemented! Furthermore, the group G is currently returned only as a finitely presented group (without any method for solving the word problem).

::::::::::::::::::::::::
```
ResolutionNilpotentGroup(G,n)
ResolutionNilpotentGroup(G,n,"TestFiniteness")
```

Inputs a nilpotent group G and positive integer n. It returns n terms of a free ZG-resolution. The resolution is computed using a divide-and-conquer technique involving the lower central series.

This function can be applied to infinite groups G. For finite groups the function *ResolutionNormalSeries*() probably gives better results.

If an optional third argument is set equal to "TestFiniteness" then the groups N and G will be tested to see if they are finite. If they are finite then some speed saving routines will be invoked.

The contracting homotopy on the ZE-resolution has not yet been fully implemented for infinite groups.

::::::::::::::::::::::::
```
ResolutionNormalSeries(L,n)
ResolutionNormalSeries(L,n,true)
ResolutionNormalSeries(L,n,false,p)
```

Inputs a positive integer n and a list $L = [L_1, ..., L_k]$ of normal subgroups L_i of a finite group G satisfying $G = L_1 > L2 > \ldots > L_k$. Alternatively, $L = [gensL_1, ...gensL_k]$ can be a list of generating sets

for the L_i (and these particular generators will be used in the construction of resolutions). It returns a ZG-resolution by repeatedly using the function *ResolutionFiniteExtension*().

The function has an optional third argument which, if set equal to true, invokes tietze reductions in the construction of resolutions.

The function has an optional fourth argument which, if set equal to p > 0, produces a resolution which is only valid for mod p calculations.

::::::::::::::::::::::::
`ResolutionPrimePowerGroup(P,n)`
`ResolutionPrimePowerGroup(G,n,p)`

Inputs a p-group P and integer n>0. It uses GAP's standard linear algebra functions over the field F of p elements to construct a free FP-resolution for mod p calculations only. The resolution is minimal - meaning that the number of generators of R_n equals the rank of $H_n(P,F)$.

The function can also be used to obtain a free non-minimal FG-resolution of a small group G of non-prime-power order. In this case the prime p must be entered as the third input variable. (In the non-prime-power case the algorithm is naive and not very good.)

::::::::::::::::::::::::
`ResolutionSmallFpGroup(G,n)`
`ResolutionSmallFpGroup(G,n,p)`

Inputs a small finitely presented group G and an integer n>0. It returns n terms of a ZG-resolution which, in dimensions 1 and 2, corresponds to the given presentation for G. The method returns no contracting homotopy for the resolution.

The function has an optional fourth argument which, when set equal to a prime p, records the fact that the resolution will only be used for mod p calculations. This could speed up subsequent constructions.

This function was written by Irina Kholodna.

::::::::::::::::::::::::
`ResolutionSubgroup(R,K)`

Inputs a ZG-resolution for an (infinite) group G and a subgroup K of finite index $|G:K|$. It returns a free ZK-resolution whose ZK-rank is $|G:K|$ times the ZG-rank in each dimension.

If G is finite then the function *ResolutionFiniteSubgroup*(R,G,K) will probably work better. In particular, resolutions from this function probably won't work with the function *EquivariantChainMap*(). This ZK-resolution is not reduced. i.e. it has more than one generator in dimension 0.

::::::::::::::::::::::::
`ResolutionSubnormalSeries(L,n)`

Inputs a positive integer n and a list $L = [L_1, \ldots, L_k]$ of subgroups L_i of a finite group $G = L_1$ such that $L_1 > L2 \ldots > L_k$ is a subnormal series in G (meaning that each L_{i+1} must be normal in L_i). It returns a ZG-resolution by repeatedly using the function $ResolutionFiniteExtension()$.

If L is a series of normal subgroups in G then the function $ResolutionNormalSeries(L,n)$ will possibly work more efficiently.

::::::::::::::::::::::::
`TwistedTensorProduct(R,S,EhomG,GmapE,NhomE,NEhomN,EltsE,Mult,InvE)`

Inputs a ZG-resolution R, a ZN-resolution S, and other data relating to a short exact sequence $1 \longrightarrow N \longrightarrow E \longrightarrow G \longrightarrow 1$. It uses a perturbation technique of CTC Wall to construct a ZE-resolution F. Both G and N could be infinite. The "length" of F is equal to the minimum of the "length"s of R and S. The resolution R needs no contracting homotopy if no such homotopy is requied for F.

::::::::::::::::::::::::
`ConjugatedResolution(R,x)`

Inputs a ZG-resoluton R and an element x from some group containing G. It returns a ZG^x-resolution S where the group G^x is the conjugate of G by x. (The component $S!.elts$ will be a pseudolist rather than a list.)

::::::::::::::::::::::::
`RecalculateIncidenceNumbers(R)`

Inputs a ZG-resoluton R which arises as the cellular chain complex of a regular CW-complex. (Thus the boundary of any cell is a list of distinct cells.) It recalculates the incidence numbers for R. If it is applied to a resolution that is not regular then a wrong answer may be returned.

Chapter 2

Resolutions of modules

::::::::::::::::::::::::
`ResolutionFpGModule(M,n)`

Inputs an FpG-module M and a positive integer n. It returns n terms of a minimal free FG-resolution of the module M (where G is a finite group and F the field of p elements).

Chapter 3

Induced equivariant chain maps

::::::::::::::::::::::::
`EquivariantChainMap(R,S,f)`

Inputs a ZG-resolution R, a ZG'-resolution S, and a group homomorphism $f : G \longrightarrow G'$. It outputs a component object M with the following components.

$M!.source$ is the resolution R.

$M!.target$ is the resolution S.

$M!.mapping(w,n)$ is a function which gives the image in S_n, under a chain map induced by f, of a word w in R_n. (Here R_n and S_n are the n-th modules in the resolutions R and S.)

$F!.properties$ is a list of pairs such as ["type", "equivariantChainMap"].

The resolution S must have a contracting homotopy.

Chapter 4

Functors

::::::::::::::::::::::::
`ExtendScalars(R,G,EltsG)`

Inputs a ZH-resolution R, a group G containing H as a subgroup, and a list $EltsG$ of elements of G. It returns the free ZG-resolution $(R \otimes_{ZH} ZG)$. The returned resolution S has S!.elts:=EltsG. This is a resolution of the ZG-module $(Z \otimes_{ZH} ZG)$. (Here $\otimes_{ZH}$ means tensor over ZH.)

::::::::::::::::::::::::
`HomToIntegers(X)`

Inputs either a ZG-resolution $X = R$, or an equivariant chain map $X = (F : R \longrightarrow S)$. It returns the cochain complex or cochain map obtained by applying $HomZG(\_, Z)$ where Z is the trivial module of integers (characteristic 0).

::::::::::::::::::::::::
`HomToIntegersModP(R)`

Inputs a ZG-resolution R and returns the cochain complex obtained by applying $HomZG(\_, Z_p)$ where Z_p is the trivial module of integers mod p. (At present this functor does not handle equivariant chain maps.)

::::::::::::::::::::::::
`HomToIntegralModule(R,f)`

Inputs a ZG-resolution R and a group homomorphism $f : G \longrightarrow GL_n(Z)$ to the group of $n \times n$ invertible integer matrices. Here Z must have characteristic 0. It returns the cochain complex obtained by applying $HomZG(\_, A)$ where A is the ZG-module Z^n with G action via f. (At present this function

does not handle equivariant chain maps.)

::::::::::::::::::::::::
`TensorWithIntegralModule(R,f)`

Inputs a ZG-resolution R and a group homomorphism $f : G \longrightarrow GL_n(Z)$ to the group of $n \times n$ invertible integer matrices. Here Z must have characteristic 0. It returns the chain complex obtained by tensoring over ZG with the ZG-module $A = Z^n$ with G action via f. (At present this function does not handle equivariant chain maps.)

::::::::::::::::::::::::
`HomToGModule(R,A)`

Inputs a ZG-resolution R and an abelian G-outer group A. It returns the G-cocomplex obtained by applying $HomZG(\_, A)$. (At present this function does not handle equivariant chain maps.)

::::::::::::::::::::::::
`InduceScalars(R,hom)`

Inputs a ZQ-resolution R and a surjective group homomorphism $hom : G \to Q$. It returns the unduced non-free ZG-resolution.

::::::::::::::::::::::::
`LowerCentralSeriesLieAlgebra(G)`
`LowerCentralSeriesLieAlgebra(f)`

Inputs a pcp group G. If each quotient G_c/G_{c+1} of the lower central series is free abelian or p-elementary abelian (for fixed prime p) then a Lie algebra $L(G)$ is returned. The abelian group underlying $L(G)$ is the direct sum of the quotients G_c/G_{c+1} . The Lie bracket on $L(G)$ is induced by the commutator in G. (Here $G_1 = G$, $G_{c+1} = [G_c, G]$.)

The function can also be applied to a group homomorphism $f : G \longrightarrow G'$. In this case the induced homomorphism of Lie algebras $L(f) : L(G) \longrightarrow L(G')$ is returned.

If the quotients of the lower central series are not all free or p-elementary abelian then the function returns fail.

This function was written by Pablo Fernandez Ascariz

::::::::::::::::::::::::

`TensorWithIntegers(X)`

Inputs either a ZG-resolution $X = R$, or an equivariant chain map $X = (F : R \longrightarrow S)$. It returns the chain complex or chain map obtained by tensoring with the trivial module of integers (characteristic 0).

::::::::::::::::::::::::
`FilteredTensorWithIntegers(R)`

Inputs a ZG-resolution R for which "filteredDimension" lies in NamesOfComponents(R). (Such a resolution can be produced using TwisterTensorProduct(), ResolutionNormalSubgroups() or FreeGResolution().) It returns the filtered chain complex obtained by tensoring with the trivial module of integers (characteristic 0).

::::::::::::::::::::::::
`TensorWithTwistedIntegers(X,rho)`

Inputs either a ZG-resolution $X = R$, or an equivariant chain map $X = (F : R \longrightarrow S)$. It also inputs a function $rho: G \to \mathbb{Z}$ where the action of $g \in G$ on $\mathbb{Z}$ is such that $g.1 = rho(g)$. It returns the chain complex or chain map obtained by tensoring with the (twisted) module of integers (characteristic 0).

::::::::::::::::::::::::
`TensorWithIntegersModP(X,p)`

Inputs either a ZG-resolution $X = R$, or a characteristics 0 chain complex, or an equivariant chain map $X = (F : R \longrightarrow S)$, or a chain map between characteristic 0 chain complexes, together with a prime p. It returns the chain complex or chain map obtained by tensoring with the trivial module of integers modulo p.

::::::::::::::::::::::::
`TensorWithTwistedIntegersModP(X,p,rho)`

Inputs either a ZG-resolution $X = R$, or an equivariant chain map $X = (F : R \longrightarrow S)$, and a prime p. It also inputs a function $rho: G \to \mathbb{Z}$ where the action of $g \in G$ on $\mathbb{Z}$ is such that $g.1 = rho(g)$. It returns the chain complex or chain map obtained by tensoring with the trivial module of integers modulo p.

::::::::::::::::::::::::
`TensorWithRationals(R)`

Inputs a ZG-resolution R and returns the chain complex obtained by tensoring with the trivial module of rational numbers.

Chapter 5

Chain complexes

::::::::::::::::::::::::
`ChainComplex(T)`

Inputs a pure cubical complex, or cubical complex, or simplicial complex T and returns the (often very large) cellular chain complex of T.

::::::::::::::::::::::::
`ChainComplexOfPair(T,S)`

Inputs a pure cubical complex or cubical complex T and contractible subcomplex S. It returns the quotient $C(T)/C(S)$ of cellular chain complexes.

::::::::::::::::::::::::
`ChevalleyEilenbergComplex(X,n)`

Inputs either a Lie algebra $X = A$ (over the ring of integers Z or over a field K) or a homomorphism of Lie algebras $X = (f : A \longrightarrow B)$, together with a positive integer n. It returns either the first n terms of the Chevalley-Eilenberg chain complex $C(A)$, or the induced map of Chevalley-Eilenberg complexes $C(f) : C(A) \longrightarrow C(B)$.

(The homology of the Chevalley-Eilenberg complex $C(A)$ is by definition the homology of the Lie algebra A with trivial coefficients in Z or K).

This function was written by PABLO FERNANDEZ ASCARIZ

::::::::::::::::::::::::
`LeibnizComplex(X,n)`

Inputs either a Lie or Leibniz algebra $X = A$ (over the ring of integers Z or over a field K) or

a homomorphism of Lie or Leibniz algebras $X = (f : A \longrightarrow B)$, together with a positive integer n. It returns either the first n terms of the Leibniz chain complex $C(A)$, or the induced map of Leibniz complexes $C(f) : C(A) \longrightarrow C(B)$.

(The Leibniz complex $C(A)$ was defined by J.-L.Loday. Its homology is by definition the Leibniz homology of the algebra A).

This function was written by PABLO FERNANDEZ ASCARIZ

:::::::::::::::::::::::::
`SuspendedChainComplex(C)`

Inputs a chain complex C and returns the chain complex S defined by applying the degree shift $S_n = C_{n-1}$ to chain groups and boundary homomorphisms.

:::::::::::::::::::::::::
`ReducedSuspendedChainComplex(C)`

Inputs a chain complex C and returns the chain complex S defined by applying the degree shift $S_n = C_{n-1}$ to chain groups and boundary homomorphisms for all $n > 0$. The chain complex S has trivial homology in degree 0 and $S_0 = \mathbb{Z}$.

:::::::::::::::::::::::::
`CoreducedChainComplex(C)`
`CoreducedChainComplex(C,2)`

Inputs a chain complex C and returns a quasi-isomorphic chain complex D. In many cases the complex D should be smaller than C. If an optional second input argument is set equal to 2 then an alternative method is used for reducing the size of the chain complex.

:::::::::::::::::::::::::
`TensorProductOfChainComplexes(C,D)`

Inputs two chain complexes C and D of the same characteristic and returns their tensor product as a chain complex.

This function was written by LE VAN LUYEN.

:::::::::::::::::::::::::
`LefschetzNumber(F)`

Inputs a chain map $F:C \to C$ with common source and target. It returns the Lefschetz number of the map (that is, the alternating sum of the traces of the homology maps in each degree).

Chapter 6

Sparse Chain complexes

::::::::::::::::::::::::
`SparseMat(A)`

Inputs a matrix *A* and returns the matrix in sparse format.

::::::::::::::::::::::::
`TransposeOfSparseMat(A)`

Inputs a sparse matrix *A* and returns its transpose sparse format.

::::::::::::::::::::::::
`ReverseSparseMat(A)`

Inputs a sparse matrix *A* and modifies it by reversing the order of the columns. This function modifies *A* and returns no value.

::::::::::::::::::::::::
`SparseRowMult(A,i,k)`

Multiplies the i-th row of a sparse matrix *A* by *k*. The sparse matrix *A* is modified but nothing is returned.

::::::::::::::::::::::::
`SparseRowInterchange(A,i,k)`

Interchanges the i-th and j-th rows of a sparse matrix A by k. The sparse matrix A is modified but nothing is returned.

::::::::::::::::::::::::
`SparseRowAdd(A,i,j,k)`

Adds k times the j-th row to the i-th row of a sparse matrix A. The sparse matrix A is modified but nothing is returned.

::::::::::::::::::::::::
`SparseSemiEchelon(A)`

Converts a sparse matrix A to semi-echelon form (which means echelon form up to a permutation of rows). The sparse matrix A is modified but nothing is returned.

::::::::::::::::::::::::
`RankMatDestructive(A)`

Returns the rank of a sparse matrix A. The sparse matrix A is modified during the calculation.

::::::::::::::::::::::::
`RankMat(A)`

Returns the rank of a sparse matrix A.

::::::::::::::::::::::::
`SparseChainComplex(Y)`

Inputs a regular CW-complex Y and returns a sparse chain complex which is chain homotopy equivalent to the cellular chain complex of Y. The function uses discrete vector fields to calculate a smallish chain complex.

::::::::::::::::::::::::
`SparseChainComplexOfRegularCWComplex(Y)`

Inputs a regular CW-complex Y and returns its cellular chain complex as a sparse chain complex. The function SparseChainComplex(Y) will usually return a smaller chain complex.

::::::::::::::::::::::::
`SparseBoundaryMatrix(C,n)`

Inputs a sparse chain complex C and integer n. Returns the n-th boundary matrix of the chain complex in sparse format.

::::::::::::::::::::::::
`Bettinumbers(C,n)`

Inputs a sparse chain complex C and integer n. Returns the n-th Netti number of the chain complex.

Chapter 7

Homology and cohomology groups

::::::::::::::::::::::::
`Cohomology(X,n)`

Inputs either a cochain complex $X = C$ (or G-cocomplex C) or a cochain map $X = (C \longrightarrow D)$ in characteristic p together with a non-negative intereg n.

If $X = C$ and $p = 0$ then the torsion coefficients of $H^n(C)$ are retuned. If $X = C$ and p is prime then the dimension of $H^n(C)$ are retuned.

If $X = (C \longrightarrow D)$ then the induced homomorphism $H^n(C) \longrightarrow H^n(D)$ is returned as a homomorphism of finitely presented groups.

A G-cocomplex C can also be input. The cohomology groups of such a complex may not be abelian. WARNING: in this case Cohomology(C,n) returns the abelian invariants of the n-th cohomology group of C.

::::::::::::::::::::::::
`CohomologyModule(C,n)`

Inputs a G-cocomplex C together with a non-negative integer n. It returns the cohomology $H^n(C)$ as a G-outer group. If C was constructed from a resolution R by homing to an abelian G-outer group A then, for each x in H:=CohomologyModule(C,n), there is a function f:=H!.representativeCocycle(x) which is a standard n-cocycle corresponding to the cohomology class x. (At present this works only for n=1,2,3.)

::::::::::::::::::::::::
`CohomologyPrimePart(C,n,p)`

Inputs a cochain complex C in characteristic 0, a positive integer n, and a prime p. It returns a

list of those torsion coefficients of $H^n(C)$ that are positive powers of p. The function uses the EDIM package by Frank Luebeck.

::::::::::::::::::::::::
`GroupCohomology(X,n)`
`GroupCohomology(X,n,p)`

Inputs a positive integer n and either

a finite group $X = G$

or a nilpotent Pcp-group $X = G$

or a space group $X = G$

or a list $X = D$ representing a graph of groups

or a pair $X = [\text{"}Artin\text{"}, D]$ where D is a Coxeter diagram representing an infinite Artin group G.

or a pair $X = [\text{"}Coxeter\text{"}, D]$ where D is a Coxeter diagram representing a finite Coxeter group G.

It returns the torsion coefficients of the integral cohomology $H^n(G,Z)$.

There is an optional third argument which, when set equal to a prime p, causes the function to return the the mod p cohomology $H^n(G,Z_p)$ as a list of length equal to its rank.

This function is a composite of more basic functions, and makes choices for a number of parameters. For a particular group you would almost certainly be better using the more basic functions and making the choices yourself!

::::::::::::::::::::::::
`GroupHomology(X,n)`
`GroupHomology(X,n,p)`

Inputs a positive integer n and either

a finite group $X = G$

or a nilpotent Pcp-group $X = G$

or a space group $X = G$

or a list $X = D$ representing a graph of groups

or a pair $X = [\text{"}Artin\text{"}, D]$ where D is a Coxeter diagram representing an infinite Artin group G.

or a pair $X = [\text{"}Coxeter\text{"}, D]$ where D is a Coxeter diagram representing a finite Coxeter group G.

It returns the torsion coefficients of the integral homology $H_n(G,Z)$.

There is an optional third argument which, when set equal to a prime p, causes the function to return the mod p homology $H_n(G,Z_p)$ as a list of lenth equal to its rank.

This function is a composite of more basic functions, and makes choices for a number of parameters. For a particular group you would almost certainly be better using the more basic functions and making the choices yourself!

::::::::::::::::::::::::
`PersistentHomologyOfQuotientGroupSeries(S,n)`
`PersistentHomologyOfQuotientGroupSeries(S,n,p,Resolution_Algorithm)`

Inputs a positive integer n and a decreasing chain $S = [S_1, S_2, ..., S_k]$ of normal subgroups in a finite p-group $G = S_1$. It returns the bar code of the persistent mod p homology in degree n of the sequence of quotient homomorphisms $G \to G/S_k \to G/S_{k-1} \to ... \to G/S_2$. The bar code is returned as a matrix containing the dimensions of the images of the induced homology maps.

If one sets $p = 0$ then the integral persitent homology bar code is returned. This is a matrix whose entries are pairs of the lists: the list of abelian invariants of the images of the induced homology maps and the cokernels of the induced homology maps. (The matrix probably does not uniquely determine the induced homology maps.)

Non prime-power (and possibly infinite) groups G can also be handled; in this case the prime must be entered as a third argument, and the resolution algorithm (e.g. ResolutionNilpotentGroup) can be entered as a fourth argument. (The default algorithm is ResolutionFiniteGroup, so this must be changed for infinite groups.)

::::::::::::::::::::::::
`PersistentCohomologyOfQuotientGroupSeries(S,n)`
`PersistentCohomologyOfQuotientGroupSeries(S,n,p,Resolution_Algorithm)`

Inputs a positive integer n and a decreasing chain $S = [S_1, S_2, ..., S_k]$ of normal subgroups in a finite p-group $G = S_1$. It returns the bar code of the persistent mod p cohomology in degree n of the sequence of quotient homomorphisms $G \to G/S_k \to G/S_{k-1} \to ... \to G/S_2$. The bar code is returned as a matrix containing the dimensions of the images of the induced homology maps.

If one sets $p = 0$ then the integral persitent cohomology bar code is returned. This is a matrix whose entries are pairs of the lists: the list of abelian invariants of the images of the induced cohomology maps and the cokernels of the induced cohomology maps. (The matrix probably does not uniquely determine the induced homology maps.)

Non prime-power (and possibly infinite) groups G can also be handled; in this case the prime must be entered as a third argument, and the resolution algorithm (e.g. ResolutionNilpotentGroup) can be entered as a fourth argument. (The default algorithm is ResolutionFiniteGroup, so this must be changed for infinite groups.)

(The implementation is possibly a little less efficient than that of the corresponding persistent homology function.)

::::::::::::::::::::::::
`UniversalBarCode("UpperCentralSeries",n,d)`
`UniversalBarCode("UpperCentralSeries",n,d,k)`

Inputs integers n,d that identify a prime power group G=SmallGroup(n,d), together with one of the strings "UpperCentralSeries", LowerCentralSeries", "DerivedSeries", "UpperPCentralSeries", "LowerPCentralSeries". The function returns a matrix of rational functions; the coefficients of x^k in their expansions yield the persistence matrix for the degree k homology with trivial mod p coefficients associated to the quotients of G by the terms of the given series.

If the additional integer argument k is supplied then the function returns the degree k homology persistence matrix.

::::::::::::::::::::::::
`PersistentHomologyOfSubGroupSeries(S,n)`
`PersistentHomologyOfSubGroupSeries(S,n,p,Resolution_Algorithm)`

Inputs a positive integer n and a decreasing chain $S = [S_1, S_2, ..., S_k]$ of subgroups in a finite p-group $G = S_1$. It returns the bar code of the persistent mod p homology in degree n of the sequence of inclusion homomorphisms $S_k \to S_{k-1} \to ... \to S_1 = G$. The bar code is returned as a binary matrix.

Non prime-power (and possibly infinite) groups G can also be handled; in this case the prime must be entered as a third argument, and the resolution algorithm (e.g. ResolutionNilpotentGroup) must be entered as a fourth argument.

::::::::::::::::::::::::
`PersistentHomologyOfFilteredChainComplex(C,n,p)`

Inputs a filtered chain complex C (of characteristic 0 or p) together with a positive integer n and prime p. It returns the bar code of the persistent mod p homology in degree n of the filtered chain complex C. (This function needs a more efficient implementation. Its fine as it stands for investigation in group homology, but not sufficiently efficient for the homology of large complexes arising in applied topology.)

`PersistentHomologyOfCommutativeDiagramOfPGroups(D,n)`

Inputs a commutative diagram D of finite p-groups and a positive integer n. It returns a list containing, for each homomorphism in the nerve of D, a triple $[k,l,m]$ where k is the dimension of the source of the induced mod p homology map in degree n, l is the dimension of the image, and m is the dimension of the cokernel.

::::::::::::::::::::::::
`PersistentHomologyOfFilteredCubicalComplex(M,n)`

Inputs a filtered pure cubical complex M and a non-negative integer n. It returns the degree n persistent homology of M with rational coefficients.

::::::::::::::::::::::::
`PersistentHomologyOfPureCubicalComplex(L,n,p)`

Inputs a positive integer n, a prime p and an increasing chain $L = [L_1, L_2, ..., L_k]$ of subcomplexes in a pure cubical complex L_k. It returns the bar code of the persistent mod p homology in degree n of the sequence of inclusion maps. The bar code is returned as a matrix. (This function is extremely inefficient and it is better to use PersistentHomologyOFilteredfPureCubicalComplex.

::::::::::::::::::::::::
`ZZPersistentHomologyOfPureCubicalComplex(L,n,p)`

Inputs a positive integer n, a prime p and any sequence $L = [L_1, L_2, ..., L_k]$ of subcomplexes of some pure cubical complex. It returns the bar code of the zig-zag persistent mod p homology in degree n of the sequence of maps $L_1 \to L_1 \cup L_2 \leftarrow L_2 \to L_2 \cup L_3 \leftarrow L_4 \to ... \leftarrow L_k$. The bar code is returned as a matrix.

::::::::::::::::::::::::
`RipsHomology(G,n)`
`RipsHomology(G,n,p)`

Inputs a graph G, a non-negative integer n (and optionally a prime number p). It returns the integral homology (or mod p homology) in degree n of the Rips complex of G.

::::::::::::::::::::::::
`BarCode(P)`

Inputs an integer persistence matrix P and returns the same information in the form of a binary matrix (corresponding to the usual bar code).

::::::::::::::::::::::::
::::::::::::::::::::::::
`BarCodeDisplay(P)`
`BarCodeDisplay(P,"mozilla")`
`BarCodeCompactDisplay(P)`
`BarCodeCompactDisplay(P,"mozilla")`

Inputs an integer persistence matrix P, and an optional string specifying a viewer/browser. It displays a picture of the bar code (using GraphViz software). The compact display is better for large bar codes.

::::::::::::::::::::::::
`Homology(X,n)`

Inputs either a chain complex $X = C$ or a chain map $X = (C \longrightarrow D)$.

If $X = C$ then the torsion coefficients of $H_n(C)$ are retuned.

If $X = (C \longrightarrow D)$ then the induced homomorphism $H_n(C) \longrightarrow H_n(D)$ is returned as a homomorphism of finitely presented groups.

A G-complex C can also be input. The homology groups of such a complex may not be abelian. WARNING: in this case Homology(C,n) returns the abelian invariants of the n-th homology group of C.

::::::::::::::::::::::::
`HomologyPb(C,n)`

This is a back-up function which might work in some instances where $Homology(C,n)$ fails. It is most useful for chain complexes whose boundary homomorphisms are sparse.

It inputs a chain complex C in characteristic 0 and returns the torsion coefficients of $H_n(C)$. There is a small probability that an incorrect answer could be returned. The computation relies on probabilistic Smith Normal Form algorithms implemented in the Simplicial Homology GAP package. This package therefore needs to be loaded. The computation is stored as a component of C so, when called a second time for a given C and n, the calculation is recalled without rerunning the algorithm.

The choice of probabalistic algorithm can be changed using the command

SetHomologyAlgorithm(HomologyAlgorithm[i]);

where i = 1,2,3 or 4. The upper limit for the probability of an incorrect answer can be set to any rational number 0<e<= 1 using the following command.

SetUncertaintyTolerence(e);

See the Simplicial Homology package manual for further details.

::::::::::::::::::::::::
`HomologyVectorSpace(X,n)`

Inputs either a chain complex $X = C$ or a chain map $X = (C \longrightarrow D)$ in prime characteristic.

If $X = C$ then $H_n(C)$ is retuned as a vector space.

If $X = (C \longrightarrow D)$ then the induced homomorphism $H_n(C) \longrightarrow H_n(D)$ is returned as a homomorphism of vector spaces.

::::::::::::::::::::::::
`HomologyPrimePart(C,n,p)`

Inputs a chain complex C in characteristic 0, a positive integer n, and a prime p. It returns a list of those torsion coefficients of $H_n(C)$ that are positive powers of p. The function uses the EDIM GAP package by Frank Luebeck.

::::::::::::::::::::::::
`LeibnizAlgebraHomology(A,n)`

Inputs a Lie or Leibniz algebra $X = A$ (over the ring of integers Z or over a field K), together with a positive integer n. It returns the n-dimensional Leibniz homology of A.

::::::::::::::::::::::::
`LieAlgebraHomology(A,n)`

Inputs a Lie algebra A (over the integers or a field) and a positive integer n. It returns the homology $H_n(A,k)$ where k denotes the ground ring.

::::::::::::::::::::::::
`PrimePartDerivedFunctor(G,R,F,n)`

Inputs a finite group G, a positive integer n, at least $n+1$ terms of a ZP-resolution for a Sylow subgroup P<G and a "mathematically suitable" covariant additive functor F such as TensorWithIntegers . It returns the abelian invariants of the p-component of the homology $H_n(F(R))$.

Warning: All calculations are assumed to be in characteristic 0. The function should not be used if the coefficient module is over the field of p elements.

"Mathematically suitable" means that the Cartan-Eilenberg double coset formula must hold.

::::::::::::::::::::::::
`RankHomologyPGroup(G,n)`
`RankHomologyPGroup(R,n)`
`RankHomologyPGroup(G,n,"empirical")`

Inputs a (smallish) p-group G, or n terms of a minimal Z_pG-resolution R of Z_p , together with a positive integer n. It returns the minimal number of generators of the integral homology group $H_n(G,Z)$.

If an option third string argument "empirical" is included then an empirical algorithm will be used. This is one which always seems to yield the right answer but which we can't prove yields the correct answer.

::::::::::::::::::::::::
`RankPrimeHomology(G,n)`

Inputs a (smallish) p-group G together with a positive integer n. It returns a function $dim(k)$ which gives the rank of the vector space $H_k(G,Z_p)$ for all 0 <= k <= n.

Chapter 8

Poincare series

`EfficientNormalSubgroups(G)`
`EfficientNormalSubgroups(G,k)`

Inputs a prime-power group G and, optionally, a positive integer k. The default is $k = 4$. The function returns a list of normal subgroups N in G such that the Poincare series for G equals the Poincare series for the direct product $(N \times (G/N))$ up to degree k.

::::::::::::::::::::::::::
`ExpansionOfRationalFunction(f,n)`

Inputs a positive integer n and a rational function $f(x) = p(x)/q(x)$ where the degree of the polynomial $p(x)$ is less than that of $q(x)$. It returns a list $[a_0, a_1, a_2, a_3, \ldots, a_n]$ of the first $n+1$ coefficients of the infinite expansion

$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots$.

::::::::::::::::::::::::::
`PoincareSeries(G,n)`
`PoincareSeries(R,n)`
`PoincareSeries(L,n)`
`PoincareSeries(G)`

Inputs a finite p-group G and a positive integer n. It returns a quotient of polynomials $f(x) = P(x)/Q(x)$ whose coefficient of x^k equals the rank of the vector space $H_k(G, Z_p)$ for all k in the range $k = 1$ to $k = n$. (The second input variable can be omitted, in which case the function tries to choose a "reasonable" value for n. For 2-groups the function PoincareSeriesLHS(G) can be used to produce an $f(x)$ that is correct in all degrees.)

In place of the group G the function can also input (at least n terms of) a minimal mod p resolution R for G.

Alternatively, the first input variable can be a list L of integers. In this case the coefficient of x^k in

$f(x)$ is equal to the $(k+1)$st term in the list.

::::::::::::::::::::::::
`PoincareSeriesPrimePart(G,p,n)`

Inputs a finite group G, a prime p, and a positive integer n. It returns a quotient of polynomials $f(x) = P(x)/Q(x)$ whose coefficient of x^k equals the rank of the vector space $H_k(G, Z_p)$ for all k in the range $k = 1$ to $k = n$.

The efficiency of this function needs to be improved.

`PoincareSeriesLHS(G)`

Inputs a finite 2-group G and returns a quotient of polynomials $f(x) = P(x)/Q(x)$ whose coefficient of x^k equals the rank of the vector space $H_k(G, Z_2)$ for all k.

This function was written by PAUL SMITH. It use the Singular system for commutative algebra.

::::::::::::::::::::::::
`Prank(G)`

Inputs a p-group G and returns the rank of the largest elementary abelian subgroup.

Chapter 9

Cohomology ring structure

::::::::::::::::::::::::
`IntegralCupProduct(R,u,v,p,q)`
`IntegralCupProduct(R,u,v,p,q,P,Q,N)`

(Various functions used to construct the cup product are also available.)

Inputs a ZG-resolution R, a vector u representing an element in $H^p(G,Z)$, a vector v representing an element in $H^q(G,Z)$ and the two integers p,q >0. It returns a vector w representing the cup product $u \cdot v$ in $H^{p+q}(G,Z)$. This product is associative and $u \cdot v = (-1)pqv \cdot u$. It provides $H^*(G,Z)$ with the structure of an anti-commutative graded ring. This function implements the cup product for characteristic 0 only.

The resolution R needs a contracting homotopy.

To save the function from having to calculate the abelian groups $H^n(G,Z)$ additional input variables can be used in the form $IntegralCupProduct(R,u,v,p,q,P,Q,N)$, where

P is the output of the command $CR_{C}ocyclesAndCoboundaries(R,p,true)$

Q is the output of the command $CR_{C}ocyclesAndCoboundaries(R,q,true)$

N is the output of the command $CR_{C}ocyclesAndCoboundaries(R,p+q,true)$.

::::::::::::::::::::::::
`IntegralRingGenerators(R,n)`

Inputs at least $n+1$ terms of a ZG-resolution and integer n> 0. It returns a minimal list of cohomology classes in $H^n(G,Z)$ which, together with all cup products of lower degree classes, generate the group $H^n(G,Z)$.

(Let a_i be the i-th canonical generator of the d-generator abelian group $H^n(G,Z)$. The cohomology class $n_1a_1 + ... + n_da_d$ is represented by the integer vector $u = (n_1,...,n_d)$.)

::::::::::::::::::::::::
`ModPCohomologyGenerators(G,n)`
`ModPCohomologyGenerators(R)`

Inputs either a p-group G and positive integer n, or else n terms of a minimal Z_pG-resolution R of Z_p. It returns a pair whose first entry is a minimal set of homogeneous generators for the cohomology ring $A = H^*(G,Z_p)$ modulo all elements in degree greater than n. The second entry of the pair is a function deg which, when applied to a minimal generator, yields its degree.

WARNING: the following rule must be applied when multiplying generators x_i together. Only products of the form $x_1 * (x_2 * (x_3 * (x_4 * ...)))$ with $deg(x_i) \leq deg(x_{i+1})$ should be computed (since the x_i belong to a structure constant algebra with only a partially defined structure constants table).

::::::::::::::::::::::::
`ModPCohomologyRing(G,n)`
`ModPCohomologyRing(G,n,level)`
`ModPCohomologyRing(R)`
`ModPCohomologyRing(R,level)`

Inputs either a p-group G and positive integer n, or else n terms of a minimal Z_pG-resolution R of Z_p. It returns the cohomology ring $A = H^*(G,Z_p)$ modulo all elements in degree greater than n.

The ring is returned as a structure constant algebra A.

The ring A is graded. It has a component $A!.degree(x)$ which is a function returning the degree of each (homogeneous) element x in $GeneratorsOfAlgebra(A)$.

An optional input variable "level" can be set to one of the strings "medium" or "high". These settings determine parameters in the algorithm. The default setting is "medium".

When "level" is set to "high" the ring A is returned with a component $A!.niceBasis$. This component is a pair $[Coeff, Bas]$. Here Bas is a list of integer lists; a "nice" basis for the vector space A can be constructed using the command $List(Bas, x- > Product(List(x, i- > Basis(A)[i]))$. The coefficients of the canonical basis element $Basis(A)[i]$ are stored as $Coeff[i]$.

If the ring A is computed using the setting "level"="medium" then the component $A!.niceBasis$ can be added to A using the command $A := ModPCohomologyRing_part_2(A)$.

::::::::::::::::::::::::
`ModPRingGenerators(A)`

Inputs a mod p cohomology ring A (created using the preceeding function). It returns a minimal generating set for the ring A. Each generator is homogeneous.

`Mod2CohomologyRingPresentation(G)`

```
Mod2CohomologyRingPresentation(G,n)
Mod2CohomologyRingPresentation(A)
Mod2CohomologyRingPresentation(R)
```

When applied to a finite 2-group G this function returns a presentation for the mod 2 cohomology ring $H^*(G,Z_2)$. The Lyndon-Hochschild-Serre spectral sequence is used to prove that the presentation is correct.

When the function is applied to a 2-group G and positive integer n the function first constructs n terms of a free Z_2G-resolution R, then constructs the finite-dimensional graded algebra $A = H^(* \leq n)(G,Z_2)$, and finally uses A to approximate a presentation for $H^*(G,Z_2)$. For "sufficiently large" the approximation will be a correct presentation for $H^*(G,Z_2)$.

Alternatively, the function can be applied directly to either the resolution R or graded algebra A.

This function was written by PAUL SMITH. It uses the Singular commutative algebra package to handle the Lyndon-Hochschild-Serre spectral sequence.

Chapter 10

Cohomology rings of p-groups (mainly $p = 2$)

The functions on this page were written by PAUL SMITH. (They are included in HAP but they are also independently included in Paul Smiths HAPprime package.)

::::::::::::::::::::::::
`Mod2CohomologyRingPresentation(G)`
`Mod2CohomologyRingPresentation(G,n)`
`Mod2CohomologyRingPresentation(A)`
`Mod2CohomologyRingPresentation(R)`

When applied to a finite 2-group G this function returns a presentation for the mod 2 cohomology ring $H^*(G,Z_2)$. The Lyndon-Hochschild-Serre spectral sequence is used to prove that the presentation is correct.

When the function is applied to a 2-group G and positive integer n the function first constructs n terms of a free Z_2G-resolution R, then constructs the finite-dimensional graded algebra $A = H^(* \leq n)(G,Z_2)$, and finally uses A to approximate a presentation for $H^*(G,Z_2)$. For "sufficiently large" the approximation will be a correct presentation for $H^*(G,Z_2)$.

Alternatively, the function can be applied directly to either the resolution R or graded algebra A.

This function was written by PAUL SMITH. It uses the Singular commutative algebra package to handle the Lyndon-Hochschild-Serre spectral sequence.

::::::::::::::::::::::::
`PoincareSeriesLHS(G)`

Inputs a finite 2-group G and returns a quotient of polynomials $f(x) = P(x)/Q(x)$ whose coefficient of x^k equals the rank of the vector space $H_k(G,Z_2)$ for all k.

This function was written by PAUL SMITH. It use the Singular system for commutative algebra.

Chapter 11

Commutator and nonabelian tensor computations

::::::::::::::::::::::::
`BaerInvariant(G,c)`

Inputs a nilpotent group G and integer c>0. It returns the Baer invariant $M^(c)(G)$ defined as follows. For an arbitrary group G let $L^*_{c+1}(G)$ be the $(c+1)$-st term of the upper central series of the group $U = F/[[[R,F],F]...]$ (with c copies of F in the denominator) where F/R is any free presentation of G. This is an invariant of G and we define $M^{(c)}(G)$ to be the kernel of the canonical homomorphism $M^{(c)}(G) \longrightarrow G$. For $c = 1$ the Baer invariant $M^(1)(G)$ is isomorphic to the second integral homology $H_2(G,Z)$.

This function requires the NQ package.

::::::::::::::::::::::::
`BogomolovMultiplier(G)`

```
BogomolovMultiplier(G, "standard")
BogomolovMultiplier(G, "homology")
BogomolovMultiplier(G, "tensor")
```

Inputs a finite group G and returns the quotient $H_2(G,Z)/K(G)$ of the second integral homology of G where $K(G)$ is the subgroup of $H_2(G,Z)$ generated by the images of all homomorphisms $H_2(A,Z) \to H_2(G,Z)$ induced from abelian subgroups of G.

Three slight variants of the implementation are available. The defaults "standard" implementation seems to work best on average. But for some groups the "homology" implementation or the "tensor" implementation will be faster. The variants are called by including the appropriate string as the second argument.

::::::::::::::::::::::::

`Bogomology(G,n)`

Inputs a finite group G and positive integer n, and returns the quotient $H_n(G,Z)/K(G)$ of the degree n integral homology of G where $K(G)$ is the subgroup of $H_n(G,Z)$ generated by the images of all homomorphisms $H_n(A,Z) \to H_n(G,Z)$ induced from abelian subgroups of G.

::::::::::::::::::::::::
`Coclass(G)`

Inputs a group G of prime-power order p^n and nilpotency class c say. It returns the integer $r = n - c$.

::::::::::::::::::::::::
`EpiCentre(G,N)`
`EpiCentre(G)`

Inputs a finite group G and normal subgroup N and returns a subgroup $Z^*(G,N)$ of the centre of N. The group $Z^*(G,N)$ is trivial if and only if there is a crossed module $d : E \longrightarrow G$ with $N = Image(d)$ and with $Ker(d)$ equal to the subgroup of E consisting of those elements on which G acts trivially.

If no value for N is entered then it is assumed that $N = G$. In this case the group $Z^*(G,G)$ is trivial if and only if G is isomorphic to a quotient $G = E/Z(E)$ of some group E by the centre of E. (See also the command $UpperEpicentralSeries(G,c)$.)

::::::::::::::::::::::::
`NonabelianExteriorProduct(G,N)`

Inputs a finite group G and normal subgroup N. It returns a record E with the following components.

$E.homomorphism$ is a group homomorphism $\mu : (G \wedge N) \longrightarrow G$ from the nonabelian exterior product $(G \wedge N)$ to G. The kernel of μ is the relative Schur multiplier.

$E.pairing(x,y)$ is a function which inputs an element x in G and an element y in N and returns $(x \wedge y)$ in the exterior product $(G \wedge N)$.

This function should work for reasonably small nilpotent groups or extremely small non-nilpotent groups.

::::::::::::::::::::::::

`NonabelianSymmetricKernel(G)`
`NonabelianSymmetricKernel(G,m)`

Inputs a finite or nilpotent infinite group G and returns the abelian invariants of the Fourth homotopy group SG of the double suspension $SSK(G,1)$ of the Eilenberg-Mac Lane space $K(G,1)$.

For non-nilpotent groups the implementation of the function $NonabelianSymmetricKernel(G)$ is far from optimal and will soon be improved. As a temporary solution to this problem, an optional second variable m can be set equal to 0, and then the function efficiently returns the abelian invariants of groups A and B such that there is an exact sequence $0 \longrightarrow B \longrightarrow SG \longrightarrow A \longrightarrow 0$.

Alternatively, the optional second varible m can be set equal to a positive multiple of the order of the symmetric square $(G\tilde{\otimes}G)$. In this case the function returns the abelian invariants of SG. This might help when G is solvable but not nilpotent (especially if the estimated upper bound m is reasonable accurate).

::::::::::::::::::::::::
`NonabelianSymmetricSquare(G)`
`NonabelianSymmetricSquare(G,m)`

Inputs a finite or nilpotent infinite group G and returns a record T with the following components.

$T.homomorphism$ is a group homomorphism $\mu : (G\tilde{\otimes}G) \longrightarrow G$ from the nonabelian symmetric square of G to G. The kernel of μ is isomorphic to the fourth homotopy group of the double suspension $SSK(G,1)$ of an Eilenberg-Mac Lane space.

$T.pairing(x,y)$ is a function which inputs two elements x,y in G and returns the tensor $(x \otimes y)$ in the symmetric square $(G \otimes G)$.

An optional second varible m can be set equal to a multiple of the order of the symmetric square $(G\tilde{\otimes}G)$. This might help when G is solvable but not nilpotent (especially if the estimated upper bound m is reasonable accurate) as the bound is used in the solvable quotient algorithm.

The optional second variable m can also be set equal to 0. In this case the Todd-Coxeter procedure will be used to enumerate the symmetric square even when G is solvable.

This function should work for reasonably small solvable groups or extremely small non-solvable groups.

::::::::::::::::::::::::
`NonabelianTensorProduct(G,N)`

Inputs a finite group G and normal subgroup N. It returns a record E with the following components.

$E.homomorphism$ is a group homomorphism $\mu : (G \otimes N) \longrightarrow G$ from the nonabelian exterior product $(G \otimes N)$ to G.

$E.pairing(x,y)$ is a function which inputs an element x in G and an element y in N and returns $(x \otimes y)$ in the tensor product $(G \otimes N)$.

This function should work for reasonably small nilpotent groups or extremely small non-nilpotent groups.

::::::::::::::::::::::::
`NonabelianTensorSquare(G)`
`NonabelianTensorSquare(G,m)`

Inputs a finite or nilpotent infinite group G and returns a record T with the following components.

$T.homomorphism$ is a group homomorphism $\mu : (G \otimes G) \longrightarrow G$ from the nonabelian tensor square of G to G. The kernel of μ is isomorphic to the third homotopy group of the suspension $SK(G,1)$ of an Eilenberg-Mac Lane space.

$T.pairing(x,y)$ is a function which inputs two elements x,y in G and returns the tensor $(x \otimes y)$ in the tensor square $(G \otimes G)$.

An optional second varible m can be set equal to a multiple of the order of the tensor square $(G \otimes G)$. This might help when G is solvable but not nilpotent (especially if the estimated upper bound m is reasonable accurate) as the bound is used in the solvable quotient algorithm.

The optional second variable m can also be set equal to 0. In this case the Todd-Coxeter procedure will be used to enumerate the tensor square even when G is solvable.

This function should work for reasonably small solvable groups or extremely small non-solvable groups.

::::::::::::::::::::::::
`RelativeSchurMultiplier(G,N)`

Inputs a finite group G and normal subgroup N. It returns the homology group $H_2(G,N,Z)$ that fits into the exact sequence

$\ldots \longrightarrow H_3(G,Z) \longrightarrow H_3(G/N,Z) \longrightarrow H_2(G,N,Z) \longrightarrow H_3(G,Z) \longrightarrow H_3(G/N,Z) \longrightarrow \ldots.$

This function should work for reasonably small nilpotent groups G or extremely small non-nilpotent groups.

::::::::::::::::::::::::
`TensorCentre(G)`

Inputs a group G and returns the largest central subgroup N such that the induced homomorphism

of nonabelian tensor squares $(G \otimes G) \longrightarrow (G/N \otimes G/N)$ is an isomorphism. Equivalently, N is the largest central subgroup such that $\pi_3(SK(G,1)) \longrightarrow \pi_3(SK(G/N,1))$ is injective.

::::::::::::::::::::::::
`ThirdHomotopyGroupOfSuspensionB(G)`
`ThirdHomotopyGroupOfSuspensionB(G,m)`

Inputs a finite or nilpotent infinite group G and returns the abelian invariants of the third homotopy group JG of the suspension $SK(G,1)$ of the Eilenberg-Mac Lane space $K(G,1)$.

For non-nilpotent groups the implementation of the function $ThirdHomotopyGroupOfSuspensionB(G)$ is far from optimal and will soon be improved. As a temporary solution to this problem, an optional second variable m can be set equal to 0, and then the function efficiently returns the abelian invariants of groups A and B such that there is an exact sequence $0 \longrightarrow B \longrightarrow JG \longrightarrow A \longrightarrow 0$.

Alternatively, the optional second varible m can be set equal to a positive multiple of the order of the tensor square $(G \otimes G)$. In this case the function returns the abelian invariants of JG. This might help when G is solvable but not nilpotent (especially if the estimated upper bound m is reasonable accurate).

::::::::::::::::::::::::
`UpperEpicentralSeries(G,c)`

Inputs a nilpotent group G and an integer c. It returns the c-th term of the upper epicentral series $1 < Z_1^*(G) < Z_2^*(G) < \ldots$.

The upper epicentral series is defined for an arbitrary group G. The group $Z_c^*(G)$ is the image in G of the c-th term $Z_c(U)$ of the upper central series of the group $U = F/[[[R,F],F]\ldots]$ (with c copies of F in the denominator) where F/R is any free presentation of G.

This functions requires the NQ package.

Chapter 12

Lie commutators and nonabelian Lie tensors

Functions on this page are joint work with HAMID MOHAMMADZADEH, and implemented by him.

::::::::::::::::::::::::
`LieCoveringHomomorphism(L)`

Inputs a finite dimensional Lie algebra L over a field, and returns a surjective Lie homomorphism $phi : C \to L$ where:

the kernel of *phi* lies in both the centre of C and the derived subalgebra of C,

the kernel of *phi* is a vector space of rank equal to the rank of the second Chevalley-Eilenberg homology of L.

::::::::::::::::::::::::
`LeibnizQuasiCoveringHomomorphism(L)`

Inputs a finite dimensional Lie algebra L over a field, and returns a surjective homomorphism $phi : C \to L$ of Leibniz algebras where:

the kernel of *phi* lies in both the centre of C and the derived subalgebra of C,

the kernel of *phi* is a vector space of rank equal to the rank of the kernel J of the homomorphism $L \otimes L \to L$ from the tensor square to L. (We note that, in general, J is NOT equal to the second Leibniz homology of L.)

::::::::::::::::::::::::
`LieEpiCentre(L)`

Inputs a finite dimensional Lie algebra L over a field, and returns an ideal $Z^*(L)$ of the centre of L. The ideal $Z^*(L)$ is trivial if and only if L is isomorphic to a quotient $L = E/Z(E)$ of some Lie algebra E by the centre of E.

::::::::::::::::::::::::
`LieExteriorSquare(L)`

Inputs a finite dimensional Lie algebra L over a field. It returns a record E with the following components.

$E.homomorphism$ is a Lie homomorphism $\mu : (L \wedge L) \longrightarrow L$ from the nonabelian exterior square $(L \wedge L)$ to L. The kernel of μ is the Lie multiplier.

$E.pairing(x,y)$ is a function which inputs elements x,y in L and returns $(x \wedge y)$ in the exterior square $(L \wedge L)$.

::::::::::::::::::::::::
`LieTensorSquare(L)`

Inputs a finite dimensional Lie algebra L over a field and returns a record T with the following components.

$T.homomorphism$ is a Lie homomorphism $\mu : (L \otimes L) \longrightarrow L$ from the nonabelian tensor square of L to L.

$T.pairing(x,y)$ is a function which inputs two elements x,y in L and returns the tensor $(x \otimes y)$ in the tensor square $(L \otimes L)$.

::::::::::::::::::::::::
`LieTensorCentre(L)`

Inputs a finite dimensional Lie algebra L over a field and returns the largest ideal N such that the induced homomorphism of nonabelian tensor squares $(L \otimes L) \longrightarrow (L/N \otimes L/N)$ is an isomorphism.

Chapter 13

Generators and relators of groups

::::::::::::::::::::::::
`CayleyGraphOfGroupDisplay(G,X)`
`CayleyGraphOfGroupDisplay(G,X,"mozilla")`

Inputs a finite group G together with a subset X of G. It displays the corresponding Cayley graph as a .gif file. It uses the Mozilla web browser as a default to view the diagram. An alternative browser can be set using a second argument.

The argument G can also be a finite set of elements in a (possibly infinite) group containing X. The edges of the graph are coloured according to which element of X they are labelled by. The list X corresponds to the list of colours [blue, red, green, yellow, brown, black] in that order.

This function requires Graphviz software.

::::::::::::::::::::::::
`IdentityAmongRelatorsDisplay(R,n)`
`IdentityAmongRelatorsDisplay(R,n,"mozilla")`

Inputs a free ZG-resolution R and an integer n. It displays the boundary R!.boundary(3,n) as a tessellation of a sphere. It displays the tessellation as a .gif file and uses the Mozilla web browser as a default display mechanism. An alternative browser can be set using a second argument. (The resolution R should be reduced and, preferably, in dimension 1 it should correspond to a Cayley graph for G.)

This function uses GraphViz software.

::::::::::::::::::::::::
`IsAspherical(F,R)`

Inputs a free group F and a set R of words in F. It performs a test on the 2-dimensional CW-space K associated to this presentation for the group $G = F/\langle R \rangle^F$.

The function returns "true" if K has trivial second homotopy group. In this case it prints: Presen-

tation is aspherical.

Otherwise it returns "fail" and prints: Presentation is NOT piece-wise Euclidean non-positively curved. (In this case K may or may not have trivial second homotopy group. But it is NOT possible to impose a metric on K which restricts to a Euclidean metric on each 2-cell.)

The function uses Polymake software.

::::::::::::::::::::::::
`PresentationOfResolution(R)`

Inputs at least two terms of a reduced ZG-resolution R and returns a record P with components

$P.freeGroup$ is a free group F,

$P.relators$ is a list S of words in F,

$P.gens$ is a list of positive integers such that the i-th generator of the presentation corresponds to the group element R!.elts[P[i]] .

where G is isomorphic to F modulo the normal closure of S. This presentation for G corresponds to the 2-skeleton of the classifying CW-space from which R was constructed. The resolution R requires no contracting homotopy.

::::::::::::::::::::::::
`TorsionGeneratorsAbelianGroup(G)`

Inputs an abelian group G and returns a generating set $[x_1, \ldots, x_n]$ where no pair of generators have coprime orders.

Chapter 14

Orbit polytopes and fundamental domains

::::::::::::::::::::::::
`CoxeterComplex(D)`
`CoxeterComplex(D,n)`

Inputs a Coxeter diagram D of finite type. It returns a non-free ZW-resolution for the associated Coxeter group W. The non-free resolution is obtained from the permutahedron of type W. A positive integer n can be entered as an optional second variable; just the first n terms of the non-free resolution are then returned.

::::::::::::::::::::::::
`ContractibleGcomplex("PSL(4,Z)")`

Inputs one of the following strings:

"SL(2,Z)" , "SL(3,Z)" , "PGL(3,Z[i])" , "PGL(3,Eisenstein_Integers)" , "PSL(4,Z)" , "PSL(4,Z)_b" , "PSL(4,Z)_c" , "PSL(4,Z)_d" , "Sp(4,Z)"

or one of the following strings

"SL(2,O-2)" , "SL(2,O-7)" , "SL(2,O-11)" , "SL(2,O-19)" , "SL(2,O-43)" , "SL(2,O-67)" , "SL(2,O-163)"

It returns a non-free ZG-resolution for the group G described by the string. The stabilizer groups of cells are finite. (Subscripts _b , _c , _d denote alternative non-free ZG-resolutions for a given group G.)

Data for the first list of non-free resolutions was provided provided by MATHIEU DUTOUR. Data for the second list was provided by ALEXANDER RAHM.

::::::::::::::::::::::::
`QuotientOfContractibleGcomplex(C,D)`

Inputs a non-free ZG-resolution C and a finite subgroup D of G which is a subgroup of each cell stabilizer group for C. Each element of D must preserves the orientation of any cell stabilized by it. It returns the corresponding non-free $Z(G/D)$-resolution. (So, for instance, from the $SL(2,O)$ complex $C = ContractibleGcomplex(\text{"}SL(2,O-2)\text{"})$; we can construct a $PSL(2,O)$-complex using this function.)

::::::::::::::::::::::::
`TruncatedGComplex(R,m,n)`

Inputs a non-free ZG-resolution R and two positive integers m and n. It returns the non-free ZG-resolution consisting of those modules in R of degree at least m and at most n.

::::::::::::::::::::::::
`FundamentalDomainStandardSpaceGroup(v,G)`

Inputs a crystallographic group G (represented using AffineCrystGroupOnRight as in the GAP package Cryst). It also inputs a choice of vector v in the euclidean space R^n on which G acts. It returns the Dirichlet-Voronoi fundamental cell for the action of G on euclidean space corresponding to the vector v. The fundamental cell is a fundamental domain if G is Bieberbach. The fundamental cell/domain is returned as a "Polymake object". Currently the function only applies to certain crystallographic groups. See the manuals to HAPcryst and HAPpolymake for full details.

This function is part of the HAPcryst package written by MARC ROEDER and is thus only available if HAPcryst is loaded.

The function requires the use of Polymake software.

::::::::::::::::::::::::
`OrbitPolytope(G,v,L)`

Inputs a permutation group or matrix group G of degree n and a rational vector v of length n. In both cases there is a natural action of G on v. Let $P(G,v)$ be the convex polytope arising as the convex hull of the Euclidean points in the orbit of v under the action of G. The function also inputs a sublist L of the following list of strings:

["dimension","vertex_degree", "visual_graph", "schlegel","visual"]

Depending on the sublist, the function:

prints the dimension of the orbit polytope $P(G,v)$;

prints the degree of a vertex in the graph of $P(G,v)$;

visualizes the graph of $P(G,v)$;

visualizes the Schlegel diagram of $P(G,v)$;

visualizes $P(G,v)$ if the polytope is of dimension 2 or 3.

The function uses Polymake software.

::::::::::::::::::::::::
`PolytopalComplex(G,v)`
`PolytopalComplex(G,v,n)`

Inputs a permutation group or matrix group G of degree n and a rational vector v of length n. In both cases there is a natural action of G on v. Let $P(G,v)$ be the convex polytope arising as the convex hull of the Euclidean points in the orbit of v under the action of G. The cellular chain complex $C_* = C_*(P(G,v))$ is an exact sequence of (not necessarily free) ZG-modules. The function returns a component object R with components:

$R!.dimension(k)$ is a function which returns the number of G-orbits of the k-dimensional faces in $P(G,v)$. If each k-face has trivial stabilizer subgroup in G then C_k is a free ZG-module of rank $R.dimension(k)$.

$R!.stabilizer(k,n)$ is a function which returns the stabilizer subgroup for a face in the n-th orbit of k-faces.

If all faces of dimension $<k+1$ have trivial stabilizer group then the first k terms of C_* constitute part of a free ZG-resolution. The boundary map is described by the function $boundary(k,n)$. (If some faces have non-trivial stabilizer group then C_* is not free and no attempt is made to determine signs for the boundary map.)

$R!.elements$, $R!.group$, $R!.properties$ are as in a ZG-resolution.

If an optional third input variable n is used, then only the first n terms of the resolution C_* will be computed.

The function uses Polymake software.

::::::::::::::::::::::::
`PolytopalGenerators(G,v)`

Inputs a permutation group or matrix group G of degree n and a rational vector v of length n. In

both cases there is a natural action of G on v, and the vector v must be chosen so that it has trivial stabilizer subgroup in G. Let $P(G,v)$ be the convex polytope arising as the convex hull of the Euclidean points in the orbit of v under the action of G. The function returns a record P with components:

P.generators is a list of all those elements g in G such that $g \cdot v$ has an edge in common with v. The list is a generating set for G.

P.vector is the vector v.

P.hasseDiagram is the Hasse diagram of the cone at v.

The function uses Polymake software. The function is joint work with Seamus Kelly.

::::::::::::::::::::::::
`VectorStabilizer(G,v)`

Inputs a permutation group or matrix group G of degree n and a rational vector of degree n. In both cases there is a natural action of G on v and the function returns the group of elements in G that fix v.

Chapter 15

Cocycles

::::::::::::::::::::::::
`CcGroup(A,f)`

Inputs a G-module A (i.e. an abelian G-outer group) and a standard 2-cocycle f $GxG - -- > A$. It returns the extension group determined by the cocycle. The group is returned as a CcGroup.

This is a HAPcocyclic function and thus only works when HAPcocyclic is loaded.

::::::::::::::::::::::::
`CocycleCondition(R,n)`

Inputs a resolution R and an integer n>0. It returns an integer matrix M with the following property. Suppose $d = R.dimension(n)$. An integer vector $f = [f_1, \ldots, f_d]$ then represents a ZG-homomorphism $R_n \longrightarrow Z_q$ which sends the ith generator of R_n to the integer f_i in the trivial ZG-module Z_q (where possibly $q = 0$). The homomorphism f is a cocycle if and only if $M^t f = 0$ mod q.

::::::::::::::::::::::::
`StandardCocycle(R,f,n)`

`StandardCocycle(R,f,n,q)`

Inputs a ZG-resolution R (with contracting homotopy), a positive integer n and an integer vector f representing an n-cocycle $R_n \longrightarrow Z_q$ where G acts trivially on Z_q. It is assumed $q = 0$ unless a value for q is entered. The command returns a function $F(g_1, ..., g_n)$ which is the standard cocycle $G_n \longrightarrow Z_q$ corresponding to f. At present the command is implemented only for $n = 2$ or 3.

::::::::::::::::::::::::

`Syzygy(R,g)`

Inputs a ZG-resolution R (with contracting homotopy) and a list $g = [g[1], ..., g[n]]$ of elements in G. It returns a word w in R_n. The word w is the image of the n-simplex in the standard bar resolution corresponding to the n-tuple g. This function can be used to construct explicit standard n-cocycles. (Currently implemented only for n<4.)

Chapter 16

Words in free ZG-modules

::::::::::::::::::::::::
`AddFreeWords(v,w)`

Inputs two words v, w in a free ZG-module and returns their sum $v+w$. If the characteristic of Z is greater than 0 then the next function might be more efficient.

::::::::::::::::::::::::
`AddFreeWordsModP(v,w,p)`

Inputs two words v, w in a free ZG-module and the characteristic p of Z. It returns the sum $v+w$. If $p = 0$ the previous function might be fractionally quicker.

::::::::::::::::::::::::
`AlgebraicReduction(w)`

`AlgebraicReduction(w,p)`

Inputs a word w in a free ZG-module and returns a reduced version of the word in which all pairs of mutually inverse letters have been cancelled. The reduction is performed in a free abelian group unless the characteristic p of Z is entered.

::::::::::::::::::::::::
`Multiply Word(n,w)`

Inputs a word w and integer n. It returns the scalar multiple $n \cdot w$.

::::::::::::::::::::::::
`Negate([i,j])`

Inputs a pair $[i,j]$ of integers and returns $[-i,j]$.

::::::::::::::::::::::::
`NegateWord(w)`

Inputs a word w in a free ZG-module and returns the negated word $-w$.

::::::::::::::::::::::::
`PrintZGword(w,elts)`

Inputs a word w in a free ZG-module and a (possibly partial but sufficient) listing elts of the elements of G. The function prints the word w to the screen in the form

$r_1E_1 + \ldots + r_nE_n$

where r_i are elements in the group ring ZG, and E_i denotes the i-th free generator of the module.

::::::::::::::::::::::::
`TietzeReduction(S,w)`

Inputs a set S of words in a free ZG-module, and a word w in the module. The function returns a word w' such that $\{S,w'\}$ generates the same abelian group as $\{S,w\}$. The word w' is possibly shorter (and certainly no longer) than w. This function needs to be improved!

::::::::::::::::::::::::
`ResolutionBoundaryOfWord(R,n,w)`

Inputs a resolution R, a positive integer n and a list w representing a word in the free module R_n. It returns the image of w under the n-th boundary homomorphism.

Chapter 17

FpG-modules

::::::::::::::::::::::::
`CompositionSeriesOfFpGModules(M)`

Inputs an FpG-module M and returns a list of FpG-modules that constitute a composition series for M.

::::::::::::::::::::::::
`DirectSumOfFpGModules(M,N)`
`DirectSumOfFpGModules([ M[1], M[2], ..., M[k] ]))`

Inputs two FpG-modules M and N with common group and characteristic. It returns the direct sum of M and N as an FpG-Module.

Alternatively, the function can input a list of FpG-modules with common group G. It returns the direct sum of the list.

::::::::::::::::::::::::
`FpGModule(A,P)`
`FpGModule(A,G,p)`

Inputs a p-group P and a matrix A whose rows have length a multiple of the order of G. It returns the "canonical" FpG-module generated by the rows of A.

A small non-prime-power group G can also be input, provided the characteristic p is entered as a third input variable.

::::::::::::::::::::::::
`FpGModuleDualBasis(M)`

Inputs an FpG-module M. It returns a record R with two components:

$R.freeModule$ is the free module FG of rank one.

$R.basis$ is a list representing an F-basis for the module $Hom_{FG}(M,FG)$. Each term in the list is a matrix A whose rows are vectors in FG such that $M!.generators[i] \longrightarrow A[i]$ extends to a module homomorphism $M \longrightarrow FG$.

::::::::::::::::::::::::
`FpGModuleHomomorphism(M,N,A)`
`FpGModuleHomomorphismNC(M,N,A)`

Inputs FpG-modules M and N over a common p-group G. Also inputs a list A of vectors in the vector space spanned by $N!.matrix$. It tests that the function

$M!.generators[i] \longrightarrow A[i]$

extends to a homomorphism of FpG-modules and, if the test is passed, returns the corresponding FpG-module homomorphism. If the test is failed it returns fail.

The "NC" version of the function assumes that the input defines a homomorphism and simply returns the FpG-module homomorphism.

::::::::::::::::::::::::
`DesuspensionFpGModule(M,n)`
`DesuspensionFpGModule(R,n)`

Inputs a positive integer n and and FpG-module M. It returns an FpG-module D^nM which is mathematically related to M via an exact sequence $0 \longrightarrow D^nM \longrightarrow R_n \longrightarrow \ldots \longrightarrow R_0 \longrightarrow M \longrightarrow 0$ where R_* is a free resolution. (If $G = Group(M)$ is of prime-power order then the resolution is minimal.)

Alternatively, the function can input a positive integer n and at least n terms of a free resolution R of M.

::::::::::::::::::::::::
`RadicalOfFpGModule(M)`

Inputs an FpG-module M with G a p-group, and returns the Radical of M as an FpG-module. (Ig G is not a p-group then a submodule of the radical is returned.

::::::::::::::::::::::::
`RadicalSeriesOfFpGModule(M)`

Inputs an FpG-module M and returns a list of FpG-modules that constitute the radical series for M.

::::::::::::::::::::::::
`GeneratorsOfFpGModule(M)`

Inputs an FpG-module M and returns a matrix whose rows correspond to a minimal generating set for M.

::::::::::::::::::::::::
`ImageOfFpGModuleHomomorphism(f)`

Inputs an FpG-module homomorphism $f : M \longrightarrow N$ and returns its image $f(M)$ as an FpG-module.

::::::::::::::::::::::::
`GroupAlgebraAsFpGModule(G)`

Inputs a p-group G and returns its mod p group algebra as an FpG-module.

::::::::::::::::::::::::
`IntersectionOfFpGModules(M,N)`

Inputs two FpG-modules M,N arising as submodules in a common free module $(FG)^n$ where G is a finite group and F the field of p-elements. It returns the FpG-module arising as the intersection of M and N.

::::::::::::::::::::::::
`IsFpGModuleHomomorphismData(M,N,A)`

Inputs FpG-modules M and N over a common p-group G. Also inputs a list A of vectors in the vector space spanned by $N!.matrix$. It returns true if the function

$M!.generators[i] \longrightarrow A[i]$

extends to a homomorphism of FpG-modules. Otherwise it returns false.

::::::::::::::::::::::::
`MaximalSubmoduleOfFpGModule(M)`

Inputs an FpG-module M and returns one maximal FpG-submodule of M.

::::::::::::::::::::::::
`MaximalSubmodulesOfFpGModule(M)`

Inputs an FpG-module M and returns the list of maximal FpG-submodules of M.

::::::::::::::::::::::::
`MultipleOfFpGModule(w,M)`

Inputs an FpG-module M and a list $w := [g_1, ..., g_t]$ of elements in the group $G = M!.group$. The list w can be thought of as representing the element $w = g_1 + \ldots + g_t$ in the group algebra FG, and the function returns a semi-echelon matrix B which is a basis for the vector subspace wM .

::::::::::::::::::::::::
`ProjectedFpGModule(M,k)`

Inputs an FpG-module M of ambient dimension $n|G|$, and an integer k between 1 and n. The module M is a submodule of the free module $(FG)^n$. Let M_k denote the intersection of M with the last k summands of $(FG)^n$. The function returns the image of the projection of M_k onto the k-th summand of $(FG)^n$. This image is returned an FpG-module with ambient dimension $|G|$.

::::::::::::::::::::::::
`RandomHomomorphismOfFpGModules(M,N)`

Inputs two FpG-modules M and N over a common group G. It returns a random matrix A whose rows are vectors in N such that the function

$M!.generators[i] \longrightarrow A[i]$

extends to a homomorphism $M \longrightarrow N$ of FpG-modules. (There is a problem with this function at present.)

::::::::::::::::::::::::
`Rank(f)`

Inputs an FpG-module homomorphism $f : M \longrightarrow N$ and returns the dimension of the image of f as a vector space over the field F of p elements.

::::::::::::::::::::::::
`SumOfFpGModules(M,N)`

Inputs two FpG-modules M,N arising as submodules in a common free module $(FG)^n$ where G is a finite group and F the field of p-elements. It returns the FpG-Module arising as the sum of M and N.

::::::::::::::::::::::::
`SumOp(f,g)`

Inputs two FpG-module homomorphisms $f,g : M \longrightarrow N$ with common sorce and common target. It returns the sum $f+g : M \longrightarrow N$. (This operation is also available using "+".

::::::::::::::::::::::::
`VectorsToFpGModuleWords(M,L)`

Inputs an FpG-module M and a list $L = [v_1, \ldots, v_k]$ of vectors in M. It returns a list $L' = [x_1, ..., x_k]$. Each $x_j = [[W_1, G_1], ..., [W_t, G_t]]$ is a list of integer pairs corresponding to an expression of v_j as a word

$v_j = g_1 * w_1 + g_2 * w_1 + ... + g_t * w_t$

where

$g_i = Elements(M!.group)[G_i]$

$w_i = GeneratorsOfFpGModule(M)[W_i]$.

Chapter 18

Meataxe modules

::::::::::::::::::::::::
`DesuspensionMtxModule(M)`

Inputs a meataxe module M over the field of p elements and returns an FpG-module DM. The two modules are related mathematically by the existence of a short exact sequence $DM \longrightarrow FM \longrightarrow M$ with FM a free module. Thus the homological properties of DM are equal to those of M with a dimension shift.

(If $G := Group(M.generators)$ is a p-group then FM is a projective cover of M in the sense that the homomorphism $FM \longrightarrow M$ does not factor as $FM \longrightarrow P \longrightarrow M$ for any projective module P.)

::::::::::::::::::::::::
`FpG_to_MtxModule(M)`

Inputs an FpG-module M and returns an isomorphic meataxe module.

::::::::::::::::::::::::
`GeneratorsOfMtxModule(M)`

Inputs a meataxe module M acting on, say, the vector space V. The function returns a minimal list of row vectors in V which generate V as a G-module (where G=Group(M.generators)).

Chapter 19

G-Outer Groups

::::::::::::::::::::::::
`GOuterGroup(E,N)`
`GOuterGroup()`

Inputs a group E and normal subgroup N. It returns N as a G-outer group where $G = E/N$.

The function can be used without an argument. In this case an empty outer group C is returned. The components must be set using SetActingGroup(C,G), SetActedGroup(C,N) and SetOuterAction(C,alpha).

::::::::::::::::::::::::
`GOuterGroupHomomorphismNC(A,B,phi)`
`GOuterGroupHomomorphismNC()`

Inputs G-outer groups A and B with common acting group, and a group homomorphism phi:ActedGroup(A) –> ActedGroup(B). It returns the corresponding G-outer homomorphism PHI:A–> B. No check is made to verify that phi is actually a group homomorphism which preserves the G-action.

The function can be used without an argument. In this case an empty outer group homomorphism *PHI* is returned. The components must then be set.

::::::::::::::::::::::::
`GOuterHomomorphismTester(A,B,phi)`

Inputs G-outer groups A and B with common acting group, and a group homomorphism phi:ActedGroup(A) –> ActedGroup(B). It tests whether phi is a group homomorphism which preserves the G-action.

The function can be used without an argument. In this case an empty outer group homomorphism *PHI* is returned. The components must then be set.

::::::::::::::::::::::::
`Centre(A)`

Inputs G-outer group *A* and returns the group theoretic centre of ActedGroup(A) as a G-outer group.

::::::::::::::::::::::::
`DirectProductGog(A,B)`
`DirectProductGog(Lst)`

Inputs G-outer groups *A* and *B* with common acting group, and returns their group-theoretic direct product as a G-outer group. The outer action on the direct product is the diagonal one.

The function also applies to a list Lst of G-outer groups with common acting group.

For a direct product D constructed using this function, the embeddings and projections can be obtained (as G-outer group homomorphisms) using the functions Embedding(D,i) and Projection(D,i).

Chapter 20

Cat-1-groups

::::::::::::::::::::::::
`AutomorphismGroupAsCatOneGroup(G)`

Inputs a group G and returns the Cat-1-group C corresponding to the crossed module $G \to Aut(G)$.

::::::::::::::::::::::::
`HomotopyGroup(C,n)`

Inputs a cat-1-group C and an integer n. It returns the nth homotopy group of C.

::::::::::::::::::::::::
`HomotopyModule(C,2)`

Inputs a cat-1-group C and an integer n=2. It returns the second homotopy group of C as a G-module (i.e. abelian G-outer group) where G is the fundamental group of C.

::::::::::::::::::::::::
`QuasiIsomorph(C)`

Inputs a cat-1-group C and returns a cat-1-group D for which there exists some homomorphism $C \to D$ that induces isomorphisms on homotopy groups.

This function was implemented by LE VAN LUYEN.

::::::::::::::::::::::::
`ModuleAsCatOneGroup(G,alpha,M)`

Inputs a group G, an abelian group M and a homomorphism $\alpha: G \to Aut(M)$. It returns the Cat-1-group C corresponding th the zero crossed module $0: M \to G$.

::::::::::::::::::::::::
`MooreComplex(C)`

Inputs a cat-1-group C and returns its Moore complex as a G-complex (i.e. as a complex of groups considered as 1-outer groups).

::::::::::::::::::::::::
`NormalSubgroupAsCatOneGroup(G,N)`

Inputs a group G with normal subgroup N. It returns the Cat-1-group C corresponding th the inclusion crossed module $N \to G$.

::::::::::::::::::::::::
`XmodToHAP(C)`

Inputs a cat-1-group C obtained from the Xmod package and returns a cat-1-group D for which IsHapCatOneGroup(D) returns true.

It returns "fail" id C has not been produced by the Xmod package.

Chapter 21

Simplicial groups

::::::::::::::::::::::::
`NerveOfCatOneGroup(G,n)`

Inputs a cat-1-group G and a positive integer n. It returns the low-dimensional part of the nerve of G as a simplicial group of length n.

This function applies both to cat-1-groups for which IsHapCatOneGroup(G) is true, and to cat-1-groups produced using the Xmod package.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`EilenbergMacLaneSimplicialGroup(G,n,dim)`

Inputs a group G, a positive integer n, and a positive integer dim. The function returns the first $1+dim$ terms of a simplicial group with $n-1$st homotopy group equal to G and all other homotopy groups equal to zero.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`EilenbergMacLaneSimplicialGroupMap(f,n,dim)`

Inputs a group homomorphism $f : G \to Q$, a positive integer n, and a positive integer dim. The function returns the first $1+dim$ terms of a simplicial group homomorphism $f : K(G,n) \to K(Q,n)$ of Eilenberg-MacLane simplicial groups.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`MooreComplex(G)`

Inputs a simplicial group G and returns its Moore complex as a G-complex.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`ChainComplexOfSimplicialGroup(G)`

Inputs a simplicial group G and returns the cellular chain complex C of a CW-space X represented by the homotopy type of the simplicial group. Thus the homology groups of C are the integral homology groups of X.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`SimplicialGroupMap(f)`

Inputs a homomorphism $f : G \to Q$ of simplicial groups. The function returns an induced map $f : C(G) \to C(Q)$ of chain complexes whose homology is the integral homology of the simplicial group G and Q respectively.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`HomotopyGroup(G,n)`

Inputs a simplicial group G and a positive integer n. The integer n must be less than the length of G. It returns, as a group, the (n)-th homology group of its Moore complex. Thus HomotopyGroup(G,0) returns the "fundamental group" of G.

::::::::::::::::::::::::
`Representation of elements in the bar resolution`

For a group G we denote by $B_n(G)$ the free $\mathbb{Z}G$-module with basis the lists $[g_1|g_2|...|g_n]$ where the g_i range over G.

We represent a word

$w = h_1.[g_{11}|g_{12}|...|g_{1n}] - h_2.[g_{21}|g_{22}|...|g_{2n}] + ... + h_k.[g_{k1}|g_{k2}|...|g_{kn}]$

in $B_n(G)$ as a list of lists:

$[[+1, h_1, g_{11}, g_{12}, ..., g_{1n}], [-1, h_2, g_{21}, g_{22}, ...|g_{2n}] + ... + [+1, h_k, g_{k1}, g_{k2}, ..., g_{kn}].$

::::::::::::::::::::::::
`BarResolutionBoundary(w)`

This function inputs a word w in the bar resolution module $B_n(G)$ and returns its image under the boundary homomorphism $d_n: B_n(G) \to B_{n-1}(G)$ in the bar resolution.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`BarResolutionHomotopy(w)`

This function inputs a word w in the bar resolution module $B_n(G)$ and returns its image under the contracting homotopy $h_n: B_n(G) \to B_{n+1}(G)$ in the bar resolution.

This function is currently being implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`Representation of elements in the bar complex`

For a group G we denote by $BC_n(G)$ the free abelian group with basis the lists $[g_1|g_2|...|g_n]$ where the g_i range over G.

We represent a word

$w = [g_{11}|g_{12}|...|g_{1n}] - [g_{21}|g_{22}|...|g_{2n}] + ... + [g_{k1}|g_{k2}|...|g_{kn}]$

in $BC_n(G)$ as a list of lists:

$[[+1, g_{11}, g_{12}, ..., g_{1n}], [-1, g_{21}, g_{22}, ...|g_{2n}] + ... + [+1, g_{k1}, g_{k2}, ..., g_{kn}].$

::::::::::::::::::::::::
`BarComplexBoundary(w)`

This function inputs a word w in the n-th term of the bar complex $BC_n(G)$ and returns its image under the boundary homomorphism $d_n: BC_n(G) \to BC_{n-1}(G)$ in the bar complex.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`BarResolutionEquivalence(R)`

This function inputs a free ZG-resolution R. It returns a component object HE with components

HE!.phi(n,w) is a function which inputs a non-negative integer n and a word w in $B_n(G)$. It returns the image of w in R_n under a chain equivalence $\phi: B_n(G) \to R_n$.

HE!.psi(n,w) is a function which inputs a non-negative integer n and a word w in R_n. It returns the image of w in $B_n(G)$ under a chain equivalence $\psi: R_n \to B_n(G)$.

HE!.equiv(n,w) is a function which inputs a non-negative integer n and a word w in $B_n(G)$. It returns the image of w in $B_{n+1}(G)$ under a ZG-equivariant homomorphism

$equiv(n,-): B_n(G) \to B_{n+1}(G)$

satisfying

$$w - \psi(\phi(w)) = d(n+1, equiv(n,w)) + equiv(n-1, d(n,w)).$$

where $d(n,-): B_n(G) \to B_{n-1}(G)$ is the boundary homomorphism in the bar resolution.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`BarComplexEquivalence(R)`

This function inputs a free ZG-resolution R. It first constructs the chain complex $T = TensorWithIntegerts(R)$. The function returns a component object HE with components

HE!.phi(n,w) is a function which inputs a non-negative integer n and a word w in $BC_n(G)$. It returns the image of w in T_n under a chain equivalence $\phi: BC_n(G) \to T_n$.

HE!.psi(n,w) is a function which inputs a non-negative integer n and an element w in T_n. It returns the image of w in $BC_n(G)$ under a chain equivalence $\psi: T_n \to BC_n(G)$.

HE!.equiv(n,w) is a function which inputs a non-negative integer n and a word w in $BC_n(G)$. It returns the image of w in $BC_{n+1}(G)$ under a homomorphism

$equiv(n,-)\colon BC_n(G) \to BC_{n+1}(G)$

satisfying

$$w - \psi(\phi(w)) = d(n+1, equiv(n,w)) + equiv(n-1, d(n,w)).$$

where $d(n,-)\colon BC_n(G) \to BC_{n-1}(G)$ is the boundary homomorphism in the bar complex.

This function was implemented by VAN LUYEN LE.

::::::::::::::::::::::::
`Representation of elements in the bar cocomplex`

For a group G we denote by $BC^n(G)$ the free abelian group with basis the lists $[g_1|g_2|...|g_n]$ where the g_i range over G.

We represent a word

$w = [g_{11}|g_{12}|...|g_{1n}] - [g_{21}|g_{22}|...|g_{2n}] + ... + [g_{k1}|g_{k2}|...|g_{kn}]$

in $BC^n(G)$ as a list of lists:

$[[+1, g_{11}, g_{12}, ..., g_{1n}], [-1, g_{21}, g_{22}, ...|g_{2n}] + ... + [+1, g_{k1}, g_{k2}, ..., g_{kn}].$

::::::::::::::::::::::::
`BarCocomplexCoboundary(w)`

This function inputs a word w in the n-th term of the bar cocomplex $BC^n(G)$ and returns its image under the coboundary homomorphism $d^n\colon BC^n(G) \to BC^{n+1}(G)$ in the bar cocomplex.

This function was implemented by VAN LUYEN LE.

Chapter 22

Coxeter diagrams and graphs of groups

::::::::::::::::::::::::
`CoxeterDiagramComponents(D)`

Inputs a Coxeter diagram D and returns a list $[D_1, ..., D_d]$ of the maximal connected subgraphs D_i.

::::::::::::::::::::::::
`CoxeterDiagramDegree(D,v)`

Inputs a Coxeter diagram D and vertex v. It returns the degree of v (i.e. the number of edges incident with v).

::::::::::::::::::::::::
`CoxeterDiagramDisplay(D)`
`CoxeterDiagramDisplay(D,"web browser")`

Inputs a Coxeter diagram D and displays it as a .gif file. It uses the Mozilla web browser as a default to view the diagram. An alternative browser can be set using a second argument.

This function requires Graphviz software.

::::::::::::::::::::::::
`CoxeterDiagramFpArtinGroup(D)`

Inputs a Coxeter diagram D and returns the corresponding finitely presented Artin group.

::::::::::::::::::::::::

`CoxeterDiagramFpCoxeterGroup(D)`

Inputs a Coxeter diagram D and returns the corresponding finitely presented Coxeter group.

::::::::::::::::::::::::
`CoxeterDiagramIsSpherical(D)`

Inputs a Coxeter diagram D and returns "true" if the associated Coxeter groups is finite, and returns "false" otherwise.

::::::::::::::::::::::::
`CoxeterDiagramMatrix(D)`

Inputs a Coxeter diagram D and returns a matrix representation of it. The matrix is given as a function $DiagramMatrix(D)(i, j)$ where i, j can range over the vertices.

::::::::::::::::::::::::
`CoxeterSubDiagram(D,V)`

Inputs a Coxeter diagram D and a subset V of its vertices. It returns the full sub-diagram of D with vertex set V.

::::::::::::::::::::::::
`CoxeterDiagramVertices(D)`

Inputs a Coxeter diagram D and returns its set of vertices.

::::::::::::::::::::::::
`EvenSubgroup(G)`

Inputs a group G and returns a subgroup G^+. The subgroup is that generated by all products xy where x and y range over the generating set for G stored by GAP. The subgroup is probably only meaningful when G is an Artin or Coxeter group.

::::::::::::::::::::::::

```
GraphOfGroupsDisplay(D)
GraphOfGroupsDisplay(D,"web browser")
```

Inputs a graph of groups D and displays it as a .gif file. It uses the Mozilla web browser as a default to view the diagram. An alternative browser can be set using a second argument.

This function requires Graphviz software.

::::::::::::::::::::::::
```
GraphOfResolutions(D,n)
```

Inputs a graph of groups D and a positive integer n. It returns a graph of resolutions, each resolution being of length n. It uses the function ResolutionGenericGroup() to produce the resolutions.

::::::::::::::::::::::::
```
GraphOfGroups(D)
```

Inputs a graph of resolutions D and returns the corresponding graph of groups.

::::::::::::::::::::::::
```
GraphOfResolutionsDisplay(D)
```

Inputs a graph of resolutions D and displays it as a .gif file. It uses the Mozilla web browser as a default to view the diagram.

This function requires Graphviz software.

::::::::::::::::::::::::
```
GraphOfGroupsTest(D)
```

Inputs an object D and itries to test whether it is a Graph of Groups. However, it DOES NOT test the injectivity of any homomorphisms. It returns true if D passes the test, and false otherwise.

Note that there is no function $IsHapGraphOfGroups()$ because no special data type has been created for these graphs.

::::::::::::::::::::::::
```
TreeOfGroupsToContractibleGcomplex(D,G)
```

Inputs a graph of groups D which is a tree, and also inputs the fundamental group G of the tree

in a form which contains each of the groups in the graph as subgroups. It returns a corresponding contractible G-complex.

::::::::::::::::::::::::
`TreeOfResolutionsToContractibleGcomplex(D,G)`

Inputs a graph of resolutions D which is a tree, and also inputs the fundamental group G of the tree in a form which contains each of the groups in the graph as subgroups. It returns a corresponding contractible G-complex. The resolutions are stored as a component of the contractible G-complex.

#

Chapter 23

Torsion subcomplexes

The torsion subcomplexes subpackage has been conceived and implemented by ALEXANDER D. RAHM.
`IsPnormal( G, p)` Inputs a finite group *G* and a prime *p*. Checks if the group G is p-normal for the prime p. Zasse
`TorsionSubcomplex( groupName, p)` Inputs a cell complex with action of a group. In HAP, presently the followi

"SL(2,O-2)" , "SL(2,O-7)" , "SL(2,O-11)" , "SL(2,O-19)" , "SL(2,O-43)" , "SL(2,O-67)" , "SL(2,O-163)",

where the symbol O[-m] stands for the ring of integers in the imaginary quadratic number field Q(sqrt(-m)), the latte

The function TorsionSubcomplex prints the cells with p-torsion in their stabilizer on the screen and returns the incide

It is also possible to input the cell complexes

"SL(2,Z)" , "SL(3,Z)" , "PGL(3,Z[i])" , "PGL(3,Eisenstein_Integers)" , "PSL(4,Z)" , "PSL(4,Z)_b" , "PSL(4,Z)_c" ,

provided by MATHIEU DUTOUR.
`DisplayAvailableCellComplexes();` Displays the cell complexes that are available in HAP.
`VisualizeTorsionSkeleton( groupName, p)` Executes the function TorsionSubcomplex(groupName, p) and vi
`ReduceTorsionSubcomplex( groupName, p)` This function start with the same operations as the function Torsion

It prints on the screen which cells to merge and which edges to cut off in order to reduce the p-torsion subcomplex w

Chapter 24

Simplicial Complexes

`Homology(T,n)`
`Homology(T)`

Inputs a pure cubical complex, or cubical complex, or simplicial complex T and a non-negative integer n. It returns the n-th integral homology of T as a list of torsion integers. If no value of n is input then the list of all homologies of T in dimensions 0 to Dimension(T) is returned .

`RipsHomology(G,n)`
`RipsHomology(G,n,p)`

Inputs a graph G, a non-negative integer n (and optionally a prime number p). It returns the integral homology (or mod p homology) in degree n of the Rips complex of G.

::::::::::::::::::::::::
`Bettinumbers(T,n)`
`Bettinumbers(T)`

Inputs a pure cubical complex, or cubical complex, simplicial complex or chain complex T and a non-negative integer n. The rank of the n-th rational homology group $H_n(T,\mathbb{Q})$ is returned. If no value for n is input then the list of Betti numbers in dimensions 0 to Dimension(T) is returned .

`ChainComplex(T)`

Inputs a pure cubical complex, or cubical complex, or simplicial complex T and returns the (often very large) cellular chain complex of T.

::::::::::::::::::::::::
`CechComplexOfPureCubicalComplex(T)`

Inputs a d-dimensional pure cubical complex T and returns a simplicial complex S. The simplicial complex S has one vertex for each d-cube in T, and an n-simplex for each collection of n+1 d-cubes with non-trivial common intersection. The homotopy types of T and S are equal.

::::::::::::::::::::::::
`PureComplexToSimplicialComplex(T,k)`

Inputs either a d-dimensional pure cubical complex T or a d-dimensional pure permutahedral complex T together with a non-negative integer k. It returns the first k dimensions of a simplicial complex S. The simplicial complex S has one vertex for each d-cell in T, and an n-simplex for each collection of n+1 d-cells with non-trivial common intersection. The homotopy types of T and S are equal.

For a pure cubical complex T this uses a slightly different algorithm to the function CechComplexOfPureCubicalComplex(T) but constructs the same simplicial complex.

::::::::::::::::::::::::
`RipsChainComplex(G,n)`

Inputs a graph G and a non-negative integer n. It returns $n+1$ terms of a chain complex whose homology is that of the nerve (or Rips complex) of the graph in degrees up to n.

::::::::::::::::::::::::
`VectorsToSymmetricMatrix(M)`
`VectorsToSymmetricMatrix(M,distance)`

Inputs a matrix M of rational numbers and returns a symmetric matrix S whose (i, j) entry is the distance between the i-th row and j-th rows of M where distance is given by the sum of the absolute values of the coordinate differences.

Optionally, a function distance(v,w) can be entered as a second argument. This function has to return a rational number for each pair of rational vectors v, w of length Length(M[1]).

::::::::::::::::::::::::
`EulerCharacteristic(T)`

Inputs a pure cubical complex, or cubical complex, or simplicial complex T and returns its Euler

characteristic.

::::::::::::::::::::::::
`MaximalSimplicesToSimplicialComplex(L)`

Inputs a list L whose entries are lists of vertices representing the maximal simplices of a simplicial complex. The simplicial complex is returned. Here a "vertex" is a GAP object such as an integer or a subgroup.

::::::::::::::::::::::::
`SkeletonOfSimplicialComplex(S,k)`

Inputs a simplicial complex S and a positive integer k less than or equal to the dimension of S. It returns the truncated k-dimensional simplicial complex S^k (and leaves S unchanged).

::::::::::::::::::::::::
`GraphOfSimplicialComplex(S)`

Inputs a simplicial complex S and returns the graph of S.

::::::::::::::::::::::::
`ContractibleSubcomplexOfSimplicialComplex(S)`

Inputs a simplicial complex S and returns a (probably maximal) contractible subcomplex of S.

::::::::::::::::::::::::
`PathComponentsOfSimplicialComplex(S,n)`

Inputs a simplicial complex S and a nonnegative integer n. If $n = 0$ the number of path components of S is returned. Otherwise the n-th path component is returned (as a simplicial complex).

::::::::::::::::::::::::
`QuillenComplex(G)`

Inputs a finite group G and returns, as a simplicial complex, the order complex of the poset of

non-trivial elementary abelian subgroups of G.

::::::::::::::::::::::::
`SymmetricMatrixToIncidenceMatrix(S,t)`
`SymmetricMatrixToIncidenceMatrix(S,t,d)`

Inputs a symmetric integer matrix S and an integer t. It returns the matrix M with $M_{ij} = 1$ if I_{ij} is less than t and $I_{ij} = 1$ otherwise.

An optional integer d can be given as a third argument. In this case the incidence matrix should have roughly at most d entries in each row (corresponding to the d smallest entries in each row of S).

::::::::::::::::::::::::
`IncidenceMatrixToGraph(M)`

Inputs a symmetric 0/1 matrix M. It returns the graph with one vertex for each row of M and an edges between vertices i and j if the (i, j) entry in M equals 1.

::::::::::::::::::::::::
`CayleyGraphOfGroup(G,A)`

Inputs a group G and a set A of generators. It returns the Cayley graph.

::::::::::::::::::::::::
`PathComponentsOfGraph(G,n)`

Inputs a graph G and a nonnegative integer n. If $n = 0$ the number of path components is returned. Otherwise the n-th path component is returned (as a graph).

::::::::::::::::::::::::
`ContractGraph(G)`

Inputs a graph G and tries to remove vertices and edges to produce a smaller graph G' such that the indlusion $G' \to G$ induces a homotopy equivalence $RG \to RG'$ of Rips complexes. If the graph G is modified the function returns true, and otherwise returns false.

::::::::::::::::::::::::
`GraphDisplay(G)`

This function uses GraphViz software to display a graph G.

::::::::::::::::::::::::
::::::::::::::::::::::::
`SimplicialMap(K,L,f)`
`SimplicialMapNC(K,L,f)`

Inputs simplicial complexes K , L and a function f:K!.*vertices* $\to$ L!.*vertices* representing a simplicial map. It returns a simplicial map $K \to L$. If f does not happen to represent a simplicial map then SimplicialMap(K,L,f) will return fail; SimplicialMapNC(K,L,f) will not do any check and always return something of the data type "simplicial map".

::::::::::::::::::::::::
`ChainMapOfSimplicialMap(f)`

Inputs a simplicial map $f:K \to L$ and returns the corresponding chain map $C_*(f):C_*(K) \to C_*(L)$ of the simplicial chain complexes..

::::::::::::::::::::::::
`SimplicialNerveOfGraph(G,d)`

Inputs a graph G and returns a d-dimensional simplicial complex K whose 1-skeleton is equal to G. There is a simplicial inclusion $K \to RG$ where: (i) the inclusion induces isomorphisms on homotopy groups in dimensions less than d; (ii) the complex RG is the Rips complex (with one n-simplex for each complete subgraph of G on $n+1$ vertices).

Chapter 25

Cubical Complexes

::::::::::::::::::::::::
`ArrayToPureCubicalComplexA,n)`

Inputs an integer array *A* of dimension *d* and an integer *n*. It returns a d-dimensional pure cubical complex corresponding to the black/white "image" determined by the threshold *n* and the values of the entries in *A*. (Integers below the threshold correspond to a black pixel, and higher integers correspond to a white pixel.)

::::::::::::::::::::::::
`PureCubicalComplexA,n)`

Inputs a binary array *A* of dimension *d*. It returns the corresponding d-dimensional pure cubical complex.

::::::::::::::::::::::::
`FramedPureCubicalComplex(M)`

Inputs a pure cubical complex *M* and returns the pure cubical complex with a border of zeros attached the each face of the boundary array M!.boundaryArray. This function just adds a bit of space for performing operations such as thickenings to *M*.

::::::::::::::::::::::::
`RandomCubeOfPureCubicalComplex(M)`

Inputs a pure cubical complex *M* and returns a pure cubical complex *R* with precisely the same dimensions as *M*. The complex *R* consist of one cube selected at random from *M*.

::::::::::::::::::::::::
`PureCubicalComplexIntersection(S,T)`

Inputs two pure cubical complexes with common dimension and array size. It returns the intersection of the two complexes. (An entry in the binary array of the intersection has value 1 if and only if the corresponding entries in the binary arrays of S and T both have value 1.)

::::::::::::::::::::::::
`PureCubicalComplexUnion(S,T)`

Inputs two pure cubical complexes with common dimension and array size. It returns the union of the two complexes. (An entry in the binary array of the union has value 1 if and only if at least one of the corresponding entries in the binary arrays of S and T has value 1.)

::::::::::::::::::::::::
`PureCubicalComplexDifference(S,T)`

Inputs two pure cubical complexes with common dimension and array size. It returns the difference S-T. (An entry in the binary array of the difference has value 1 if and only if the corresponding entry in the binary array of S is 1 and the corresponding entry in the binary array of T is 0.)

::::::::::::::::::::::::
`ReadImageAsPureCubicalComplex("file.png",n)`

Reads an image file ("file.png", "file.eps", "file.bmp" etc) and an integer n between 0 and 765. It returns a 2-dimensional pure cubical complex based on the black/white version of the image determined by the threshold n.

::::::::::::::::::::::::
`ReadLinkImageAsPureCubicalComplex("file.png")`
`ReadLinkImageAsPureCubicalComplex("file.png",n)`

Reads an image file ("file.png", "file.eps", "file.bmp" etc) containing a knot or link diagram, and optionally a positive integer n. The integer n should be a little larger than the line thickness in the link diagram, and if not provided then n is set equal to 10. The function tries to output the corresponding knot or link as a 3-dimensional pure cubical complex. Ideally the link diagram should be produced with line thickness 6 in Xfig, and the under-crossing spaces should not be too large or too small or

too near one another. The function does not always succeed: it applies several checks, and if one of these checks fails then the function returns "fail".

::::::::::::::::::::::::
`ReadImageSequenceAsPureCubicalComplex("directory",n)`

Reads the name of a directory containing a sequence of image files (ordered alphanumerically), and an integer n between 0 and 765. It returns a 3-dimensional pure cubical complex based on the black/white version of the images determined by the threshold n.

`Size(T)`

This returns the number of non-zero entries in the binary array of the cubical complex, or pure cubical complex T.

`Dimension(T)`

This returns the dimension of the cubical complex, or pure cubical complex T.

::::::::::::::::::::::::
`WritePureCubicalComplexAsImage(T,"filename","ext")`

Inputs a 2-dimensional pure cubical complex T, and a filename followed by its extension (e.g. "myfile" followed by "png"). A black/white image is saved to the file.

::::::::::::::::::::::::
`ViewPureCubicalComplex(T)`
`ViewPureCubicalComplex(T,"mozilla")`

Inputs a 2-dimensional pure cubical complex T, and optionally a command such as "mozilla" for viewing image files. A black/white image is displayed.

::::::::::::::::::::::::
`Homology(T,n)`
`Homology(T)`

Inputs a pure cubical complex, or cubical complex, or simplicial complex T and a non-negative integer n. It returns the n-th integral homology of T as a list of torsion integers. If no value of n is input then the list of all homologies of T in dimensions 0 to Dimension(T) is returned .

::::::::::::::::::::::::
`Bettinumbers(T,n)`
`Bettinumbers(T)`

Inputs a pure cubical complex, or cubical complex, simplicial complex or chain complex T and a non-negative integer n. The rank of the n-th rational homology group $H_n(T,\mathbb{Q})$ is returned. If no value for n is input then the list of Betti numbers in dimensions 0 to Dimension(T) is returned .

::::::::::::::::::::::::
`DirectProductOfPureCubicalComplexes(M,N)`

Inputs two pure cubical complexes M,N and returns their direct product D as a pure cubical complex. The dimension of D is the sum of the dimensions of M and N.

::::::::::::::::::::::::
`SuspensionOfPureCubicalComplex(M)`

Inputs a pure cubical complex M and returns a pure cubical complex with the homotopy type of the suspension of M.

`EulerCharacteristic(T)`

Inputs a pure cubical complex, or cubical complex, or simplicial complex T and returns its Euler characteristic.

::::::::::::::::::::::::
`PathComponentOfPureCubicalComplex(T,n)`

Inputs a pure cubical complex T and an integer n in the rane 1, ..., Bettinumbers(T)[1] . It returns the n-th path component of T as a pure cubical complex. The value $n = 0$ is also allowed, in which case the number of path components is returned.

`ChainComplex(T)`

Inputs a pure cubical complex, or cubical complex, or simplicial complex T and returns the (often very large) cellular chain complex of T.

`ChainComplexOfPair(T,S)`

Inputs a pure cubical complex or cubical complex T and subcomplex S. It returns the quotient $C(T)/C(S)$ of cellular chain complexes.

`ExcisedPureCubicalPair(T,S)`

Inputs a pure cubical complex T and subcomplex S. It returns the pair $[T \setminus intS, S \setminus intS])$ of pure cubical complexes where $intS$ is the pure cubical complex obtained from S by removing its boundary.

::::::::::::::::::::::::
`ChainInclusionOfPureCubicalPair(S,T)`

Inputs a pure cubical complex T and subcomplex S. It returns the chain inclusion $C(S) \to C(T)$ of cellular chain complexes.

::::::::::::::::::::::::
`ChainMapOfPureCubicalPairs(M,S,N,T)`

Inputs a pure cubical complex N and subcomplexes M, T and S in T. It returns the chain map $C(M/S) \to C(N/T)$ of quotient cellular chain complexes.

::::::::::::::::::::::::
`ContractPureCubicalComplex(T)`

Inputs a pure cubical complex T of dimension d and removes d-dimensional cells from T without changing the homotopy type of T. When the function has been applied, no further d-cells can be removed from T without changing its homotopy type. This function modifies T.

::::::::::::::::::::::::
`ContractedComplex(T)`

Inputs a pure cubical complex T and returns a structural copy of the complex obtained from T by applying the function ContractPureCubicalComplex(T).

::::::::::::::::::::::::
`ZigZagContractedPureCubicalComplex(T)`

Inputs a pure cubical complex T and returns a homotopy equivalent pure cubical complex S. The aim is for S to involve fewer cells than T and certainly to involve no more cells than T.

::::::::::::::::::::::::
`ContractCubicalComplex(T)`

Inputs a cubical complex T and removes cells without changing the homotopy type of T. It changes T. In particular, it adds the components T.vectors and T.rewrite of a discrete vector field.

At present this function only works for cubical complexes of dimension 2 or 3.

::::::::::::::::::::::::
`DVFReducedCubicalComplex(T)`

Inputs a cubical complex T and returns a non-regular cubical complex R by constructing a discrete vector field. The vector field is designed to minimize the number of critical cells in R at the cost of allowing cell attaching maps that are not homeomorphisms on boundaries.

At present this function works only for 2- and 3-dimensional cubical complexes.

The function ChainComplex(R) can be used to obtain the cellular chain complex of R.

::::::::::::::::::::::::
`SkeletonOfCubicalComplex(T,n)`

Inputs a cubical complex, or pure cubical complex T and positive integer n. It returns the n-skeleton of T as a cubical complex.

::::::::::::::::::::::::
`ContractibleSubomplexOfPureCubicalComplex(T)`

Inputs a pure cubical complex T and returns a maximal contractible pure cubical subcomplex.

::::::::::::::::::::::::
`AcyclicSubomplexOfPureCubicalComplex(T)`

Inputs a pure cubical complex T and returns a (not necessarily connected) pure cubical subcomplex having trivial homology in all degrees greater than 0.

::::::::::::::::::::::::
`HomotopyEquivalentMaximalPureCubicalSubcomplex(T,S)`

Inputs a pure cubical complex T together with a pure cubical subcomplex S. It returns a pure cubical subcomplex H of T which contains S and is maximal with respect to the property that it is homotopy equivalent to S.

::::::::::::::::::::::::
`HomotopyEquivalentMinimalPureCubicalSubcomplex(T,S)`

Inputs a pure cubical complex T together with a pure cubical subcomplex S. It returns a pure cubical subcomplex H of T which contains S and is minimal with respect to the property that it is homotopy equivalent to T.

::::::::::::::::::::::::
`BoundaryOfPureCubicalComplex(T)`

Inputs a pure cubical complex T and returns its boundary as a pure cubical complex. The boundary consists of all cubes which have one or more facets that lie in just the one cube.

::::::::::::::::::::::::
`SingularitiesOfPureCubicalComplex(T,radius,tolerance)`

Inputs a pure cubical complex T together with a positive integer "radius" and an integer "tolerance" in the range 1..100. It returns the pure cubical subcomplex of those cells in the boundary where the boundary is not differentiable. (The method for deciding differentiability at a point is crude/discrete, prone to errors and depends on the radius and tolerance.)

::::::::::::::::::::::::
`ThickenedPureCubicalComplex(T)`

Inputs a pure cubical complex T and returns a pure cubical complex S. If a euclidean cube is in T then this cube and all its neighbouring cubes are included in S.

::::::::::::::::::::::::
`CropPureCubicalComplex(T)`

Inputs a pure cubical complex T and returns a pure cubical complex S obtained from T by removing any "zero boundary sheets" of the binary array. Thus S and T are isometric as euclidean spaces but there may be fewer zero entries in the binary array for S.

::::::::::::::::::::::::
`BoundingPureCubicalComplex(T)`

Inputs a pure cubical complex T and returns a contractible pure cubical complex S containing T.

::::::::::::::::::::::::
`MorseFiltration(M,i,t,bool)`
`MorseFiltration(M,i,t)`

Inputs a pure cubical complex M of dimension d, an integer i between 1 and d, a positive integer t and a boolean value True or False. The function returns a list $[M_1, M_2, ..., M_t]$ of pure cubical complexes with M_k a subcomplex of M_{k+1}. The list is constructed by setting all slices of M perpendicular to the i-th axis equal to zero if they meet the ith axis at a sufficiently high coordinate (if bool=True) or sufficiently low coordinate (if bool=False).

If the variable bool is not specified then it is assumed to have the value True.

::::::::::::::::::::::::
`ComplementOfPureCubicalComplex(T)`

Inputs a pure cubical complex T and returns a pure cubical complex S. A euclidean cube is in S precisely when the cube is not in T.

::::::::::::::::::::::::
`PureCubicalComplexToTextFile(file,M)`

Inputs a pure cubical complex M and a string containing the address of a file. A representation of this complex is written to the file in a format that can be read by the CAPD (Computer Assisted Proofs in Dynamics) software developed by Marian Mrozek and others.

::::::::::::::::::::::::
`ThickeningFiltration(M,n)`
`ThickeningFiltration(M,n,k)`

Inputs a pure cubical complex M and a positive integer n. It returns a filtered pure cubical complex constructed frim n thickenings of M. If a positive integer k is supplied as an optional third argument, then each step of the filtration is obtained from a k-fold thickening.

::::::::::::::::::::::::
`Dendrogram(M)`

Inputs a filtered pure cubical complex M and returns data that specifies the dendrogram (or phylogenetic tree) describing how path components are born and then merge during the filtration.

::::::::::::::::::::::::
`DendrogramDisplay(M)`

Inputs a filtered pure cubical complex M, or alternatively inputs the out from the command Dendrogram(M), and then uses GraphViz software to display the path component dendrogram of M.

::::::::::::::::::::::::
`DendrogramToPersistenceMat(D)`

Inputs the output of the function Dendrogram(M) and returns the corresponding degree 0 Betti bar code.

::::::::::::::::::::::::
`ReadImageAsFilteredCubicalComplex(file,n)`

Inputs a string containing the path to an image file, together with a positive integer n. It returns a

filtered pure cubical complex of filtration length n.

::::::::::::::::::::::::
`ComplementOfFilteredCubicalComplex(M)`

Inputs a filtered pure cubical complex M and returns the complement as a filtered pure cubical complex.

::::::::::::::::::::::::
`PersistentHomologyOfFilteredCubicalComplex(M,n)`

Inputs a filtered pure cubical complex M and a non-negative integer n. It returns the degree n persistent homology of M with rational coefficients.

Chapter 26

Regular CW-Complexes

`SimplicialComplexToRegularCWComplex(K)` Inputs a simplicial complex K and returns the corresponding regula
`CubicalComplexToRegularCWComplex(K)` `CubicalComplexToRegularCWComplex(K,n)` Inputs a pure cubical c
`CriticalCellsOfRegularCWComplex(Y)` `CriticalCellsOfRegularCWComplex(Y,n)` Inputs a regular CW-com
`ChainComplex(Y)` Inputs a regular CW-complex Y and returns the cellular chain complex of a CW-complex W who
`ChainComplexOfRegularCWComplex(Y)` Inputs a regular CW-complex Y and returns the cellular chain complex of
`FundamentalGroup(Y)` `FundamentalGroup(Y,n)` Inputs a regular CW-complex Y and, optionally, the number of

Chapter 27

Knots and Links

`PureCubicalKnot(L)` `PureCubicalKnot(n,i)` Inputs a list $L = [[m1,n1],[m2,n2],...,[mk,nk]]$ of pairs of integers
`ViewPureCubicalKnot(L)` Inputs a pure cubical link L and displays it.
`KnotSum(K,L)` Inputs two pure cubical knots K, L and returns their sum as a pure cubical knot. This function is not
`KnotGroup(K)` Inputs a pure cubical link K and returns the fundamental group of its complement. The group is retu
`AlexanderMatrix(G)` Inputs a finitely presented group G whose abelianization is infinite cyclic. It returns the Alex
`AlexanderPolynomial(K)` `AlexanderPolynomial(G)` Inputs either a pure cubical knot K or a finitely presented g
`ProjectionOfPureCubicalComplex(K)` Inputs an n-dimensional pure cubical complex K and returns an n-1-di
`ReadPDBfileAsPureCubicalComplex(file)` `ReadPDBfileAsPureCubicalComplex(file,m ,c)` Inputs a prote

Chapter 28

Finite metric spaces and their filtered complexes

`CayleyMetric(g,h,N)` `CayleyMetric(g,h)` Inputs two permutations g,h and optionally the degree N of a sym
`HammingMetric(g,h,N)` `HammingMetric(g,h)` Inputs two permutations g,h and optionally the degree N of a sy
`KendallMetric(g,h,N)` `KendallMetric(g,h)` Inputs two permutations g,h and optionally the degree N of a sy
`EuclideanSquaredMetric(v,w)` Inputs two vectors v,w of equal length and returns the sum of the squares of the
`EuclideanApproximatedMetric(v,w)` Inputs two vectors v,w of equal length and returns a rational approximatio
`ManhattanMetric(v,w)` Inputs two vectors v,w of equal length and returns the sum of the absolute values of the c
`VectorsToSymmetricMatrix(L)` `VectorsToSymmetricMatrix(L,D)` Inputs a list L of vectors and optionally a
`SymmetricMatDisplay(S)` `SymmetricMatDisplay(L,V)` Inputs an $n \times n$ symmetric matrix S of non-negative in
`SymmetricMatrixToFilteredGraph(S,t,m)` Inputs an integer symmetric matrix S, a positive integer t and a pos
`PermGroupToFilteredGraph(S,D)` Inputs a permutation group G and a metric D defined on permutations. The fu

Chapter 29

Commutative diagrams and abstract categories

COMMUTATIVE DIAGRAMS

::::::::::::::::::::::::
`HomomorphismChainToCommutativeDiagram(H)`

Inputs a list $H = [h_1, h_2, ..., h_n]$ of mappings such that the composite $h_1 h_2 ... h_n$ is defined. It returns the list of composable homomorphism as a commutative diagram.

::::::::::::::::::::::::
`NormalSeriesToQuotientDiagram(L)`
`NormalSeriesToQuotientDiagram(L,M)`

Inputs an increasing (or decreasing) list $L = [L_1, L_2, ..., L_n]$ of normal subgroups of a group G with $G = L_n$. It returns the chain of quotient homomorphisms $G/L_i \to G/L_{i+1}$ as a commutative diagram.

Optionally a subseries M of L can be entered as a second variable. Then the resulting diagram of quotient groups has two rows of horizontal arrows and one row of vertical arrows.

::::::::::::::::::::::::
`NerveOfCommutativeDiagram(D)`

Inputs a commutative diagram D and returns the commutative diagram ND consisting of all possible composites of the arrows in D.

::::::::::::::::::::::::
`GroupHomologyOfCommutativeDiagram(D,n)`
`GroupHomologyOfCommutativeDiagram(D,n,prime)`
`GroupHomologyOfCommutativeDiagram(D,n,prime,Resolution_Algorithm)`

Inputs a commutative diagram D of p-groups and positive integer n. It returns the commutative diagram of vector spaces obtained by applying mod p homology.

Non-prime power groups can also be handled if a prime p is entered as the third argument. Integral homology can be obtained by setting $p = 0$. For $p = 0$ the result is a diagram of groups.

A particular resolution algorithm, such as ResolutionNilpotentGroup, can be entered as a fourth argument. For positive p the default is ResolutionPrimePowerGroup. For $p = 0$ the default is ResolutionFiniteGroup.

::::::::::::::::::::::::
`PersistentHomologyOfCommutativeDiagramOfPGroups(D,n)`

Inputs a commutative diagram D of finite p-groups and a positive integer n. It returns a list containing, for each homomorphism in the nerve of D, a triple $[k,l,m]$ where k is the dimension of the source of the induced mod p homology map in degree n, l is the dimension of the image, and m is the dimension of the cokernel.

ABSTRACT CATEGORIES

::::::::::::::::::::::::
`CategoricalEnrichment(X,Name)`

Inputs a structure X such as a group or group homomorphism, together with the name of some existing category such as Name:=Category_of_Groups or Category_of_Abelian_Groups. It returns, as appropriate, an object or arrow in the named category.

::::::::::::::::::::::::
`IdentityArrow(X)`

Inputs an object X in some category, and returns the identity arrow on the object X.

::::::::::::::::::::::::

`InitialArrow(X)`

Inputs an object X in some category, and returns the arrow from the initial object in the category to X.

::::::::::::::::::::::::
`TerminalArrow(X)`

Inputs an object X in some category, and returns the arrow from X to the terminal object in the category.

::::::::::::::::::::::::
`HasInitialObject(Name)`

Inputs the name of a category and returns true or false depending on whether the category has an initial object.

::::::::::::::::::::::::
`HasTerminalObject(Name)`

Inputs the name of a category and returns true or false depending on whether the category has a terminal object.

::::::::::::::::::::::::
`Source(f)`

Inputs an arrow f in some category, and returns its source.

::::::::::::::::::::::::
`Target(f)`

Inputs an arrow f in some category, and returns its target.

::::::::::::::::::::::::
`CategoryName(X)`

Inputs an object or arrow X in some category, and returns the name of the category.

`"*", "=", "+", "-"`

Composition of suitable arrows f,g is given by $f*g$ when the source of f equals the target of g. (Warning: this differes to the standard GAP convention.)

Equality is tested using $f=g$.

In an additive category the sum and difference of suitable arrows is given by $f+g$ and $f-g$.

::::::::::::::::::::::::
`Object(X)`

Inputs an object X in some category, and returns the GAP structure Y such that $X = CategoricalEnrichment(Y, CategoryName(X))$.

::::::::::::::::::::::::
`Mapping(X)`

Inputs an arrow f in some category, and returns the GAP structure Y such that $f = CategoricalEnrichment(Y, CategoryName(X))$.

::::::::::::::::::::::::
`IsCategoryObject(X)`

Inputs X and returns true if X is an object in some category.

::::::::::::::::::::::::
`IsCategoryArrow(X)`

Inputs X and returns true if X is an arrow in some category.

Chapter 30

Arrays and Pseudo lists

::::::::::::::::::::::::
`Array(A,f)`

Inputs an array A and a function f. It returns the the array obtained by applying f to each entry of A (and leaves A unchanged).

::::::::::::::::::::::::
`PermuteArray(A,f)`

Inputs an array A of dimension d and a permutation f of degree at most d. It returns the array B defined by $B[i1][i2]...[id] = A[f(i1)][f(i2)]...A[f(id)]$ (and leaves A unchanged).

::::::::::::::::::::::::
`ArrayDimension(A)`

Inputs an array A and returns its dimension.

::::::::::::::::::::::::
`ArrayDimensions(A)`

Inputs an array A and returns its dimensions.

::::::::::::::::::::::::
`ArraySum(A)`

Inputs an array A and returns the sum of its entries.

::::::::::::::::::::::::
`ArrayValue(A,x)`

Inputs an array A and a coordinate vector x. It returns the value of the entry in A with coordinate x.

::::::::::::::::::::::::
`ArrayValueFunctions(d)`

Inputs a positive integer d and returns an efficient version of the function ArrayValue for arrays of dimension d.

::::::::::::::::::::::::
`ArrayAssign(A,x,n)`

Inputs an array A and a coordinate vector x and an integer n. It sets the entry of A with coordinate x equal to n.

::::::::::::::::::::::::
`ArrayAssignFunctions(d)`

Inputs a positive integer d and returns an efficient version of the function ArrayAssign for arrays of dimension d.

::::::::::::::::::::::::
`ArrayIterate(d)`

Inputs a positive integer d and returns a function ArrayIt(Dimensions,f). This function inputs a list Dimensions of d positive integers and also a function $f(x)$. It applies the function $f(x)$ to each integer list x of length d with entries $x[i]$ in the range [1..Dimension[i]].

::::::::::::::::::::::::
`BinaryArrayToTextFile(file,A)`

Inputs a string containing the address of a file, and an array *A* of 0s and 1s. The array represents a pure cubical complex. A representation of this complex is written to the file in a format that can be read by the CAPD (Computer Assisted Proofs in Dynamics) software developed by Marian Mrozek and others.

The second input *A* can also be a pure cubical complex.

::::::::::::::::::::::::
`FrameArray(A)`

Inputs an array *A* and returns the array obtained by appending a 0 to the beginning and end of each "row" of the array.

::::::::::::::::::::::::
`UnframeArray(A)`

Inputs an array *A* and returns the array obtained by removing the first and last entry in each "row" of the array.

::::::::::::::::::::::::
`Add(L,x)`

Let *L* be a pseudo list of length *n*, and *x* an object compatible with the entries in *L*. If *x* is not in *L* then this operation converts *L* into a pseudo list of length n+1 by adding *x* as the final entry. If *x* is in *L* the operation has no effect on *L*.

::::::::::::::::::::::::
`Append(L,K)`

Let *L* be a pseudo list and *K* a list whose objects are compatible with those in *L*. This operation applies Add(L,x) for each x in *K*.

::::::::::::::::::::::::
`ListToPseudoList(L)`

Inputs a list *L* and returns the pseudo list representation of *L*.

Chapter 31

Parallel Computation - Core Functions

::::::::::::::::::::::::
`ChildProcess()`
`ChildProcess("computer.ac.wales")`
`ChildProcess(["-m", "100000M", "-T"])`
`ChildProcess("computer.ac.wales", ["-m", "100000M", "-T"])`

This starts a GAP session as a child process and returns a stream to the child process. If no argument is given then the child process is created on the local machine; otherwise the argument should be: 1) the address of a remote computer for which ssh has been configured to require no password from the user; (2) or a list of GAP command line options; (3) or the address of a computer followed by a list of command line options.

(To configure ssh so that the user can login without a password prompt from "thishost" to "remotehost" either consult "man ssh" or

- open a shell on thishost
- cd .ssh
- ls
-> if id_dsa, id_rsa etc exists, skip the next two steps!
- ssh-keygen -t rsa
- ssh-keygen -t dsa
- scp *.pub user@remotehost:~/
- ssh remotehost -l user
- cat id_rsa.pub >> .ssh/authorized_keys
- cat id_dsa.pub >> .ssh/authorized_keys
- rm id_rsa.pub id_dsa.pub
- exit

You should now be able to connect from "thishost" to "remotehost" without a password prompt.)

::::::::::::::::::::::::
`ChildClose(s)`

This closes the stream s to a child GAP process.

::::::::::::::::::::::::
`ChildCommand("cmd;",s)`

This runs a GAP command "cmd;" on the child process accessed by the stream s. Here "cmd;" is a string representing the command.

::::::::::::::::::::::::
`NextAvailableChild(L)`

Inputs a list *L* of child processes and returns a child in *L* which is ready for computation (as soon as such a child is available).

::::::::::::::::::::::::
`IsAvailableChild(s)`

Inputs a child process *s* and returns true if s is currently available for computations, and false otherwise.

::::::::::::::::::::::::
`ChildPut(A,"B",s)`

This copies a GAP object A on the parent process to an object B on the child process s. (The copying relies on the function PrintObj(A);)

::::::::::::::::::::::::
`ChildGet("A",s)`

This functions copies a GAP object A on the child process s and returns it on the parent process. (The copying relies on the function PrintObj(A);)

::::::::::::::::::::::::
`HAPPrintTo("file",R)`

Inputs a name "file" of a new text file and a HAP object R. It writes the object R to "file". Currently this is only implemented for R equal to a resolution.

::::::::::::::::::::::::
`HAPRead("file",R)`

Inputs a name "file" containing a HAP object R and returns the object. Currently this is only implemented for R equal to a resolution.

Chapter 32

Parallel Computation - Extra Functions

::::::::::::::::::::::::
`ChildFunction("function(arg);",s)`

This runs the GAP function "function(arg);" on a child process accessed by the stream s. The output from "func;" can be accessed via the stream.

::::::::::::::::::::::::
`ChildRead(s)`

This returns, as a string, the output of the last application of $ChildFunction(" function(arg);",s)$.

::::::::::::::::::::::::
`ChildReadEval(s)`

This returns, as an evaluated string, the output of the last application of $ChildFunction(" function(arg);",s)$.

::::::::::::::::::::::::
`ParallelList(I,fn,L)`

Inputs a list I, a function fn such that $fn(x)$ is defined for all x in I, and a list of children L. It uses the children in L to compute $List(I,x->fn(x))$. (Obviously the function fn must be defined on all child processes in L.)

Chapter 33

Some functions for accessing basic data

::::::::::::::::::::::::
`BoundaryMap(C)`

Inputs a resolution, chain complex or cochain complex C and returns the function $C!.boundary$.

::::::::::::::::::::::::
`BoundaryMatrix(C,n)`

Inputs a chain or cochain complex C and integer n>0. It returns the n-th boundary map of C as a matrix.

::::::::::::::::::::::::
`Dimension(C)`

`Dimension(M)`

Inputs a resolution, chain complex or cochain complex C and returns the function $C!.dimension$.

Alternatively, inputs an FpG-module M and returns its dimension as a vector space over the field of p elements.

::::::::::::::::::::::::
`EvaluateProperty(X,"name")`

Inputs a component object X (such as a ZG-resolution or chain map) and a string "name" (such as "characteristic" or "type"). It searches $X.property$ for the pair ["name",value] and returns value. If $X.property$ does not exist, or if ["name",value] does not exist, it returns fail.

::::::::::::::::::::::::
`GroupOfResolution(R)`

Inputs a ZG-resolution R and returns the group G.

::::::::::::::::::::::::
`Length(R)`

Inputs a resolution R and returns its length (i.e. the number of terms of R that HAP has computed).

::::::::::::::::::::::::
`Map(f)`

Inputs a chain map, or cochain map or equivariant chain map f and returns the mapping function (as opposed to the target or the source of f) .

::::::::::::::::::::::::
`Source(f)`

Inputs a chain map, or cochain map, or equivariant chain map, or FpG-module homomorphism f and returns it source.

::::::::::::::::::::::::
`Target(f)`

Inputs a chain map, or cochain map, or equivariant chain map, or FpG-module homomorphism f and returns its target.

Chapter 34

Miscellaneous

::::::::::::::::::::::::
`SL2Z(p)`
`SL2Z(1/m)`

Inputs a prime p or the reciprocal $1/m$ of a square free integer m. In the first case the function returns the conjugate $SL(2,Z)^P$ of the special linear group $SL(2,Z)$ by the matrix $P = [[1,0],[0,p]]$. In the second case it returns the group $SL(2,Z[1/m])$.

::::::::::::::::::::::::
`BigStepLCS(G,n)`

Inputs a group G and a positive integer n. It returns a subseries $G = L_1 > L_2 > \ldots L_k = 1$ of the lower central series of G such that L_i/L_{i+1} has order greater than n.

::::::::::::::::::::::::
`Classify(L,Inv)`

Inputs a list of objects L and a function Inv which computes an invariant of each object. It returns a list of lists which classifies the objects of L according to the invariant..

::::::::::::::::::::::::
`RefineClassification(C,Inv)`

Inputs a list $C := Classify(L, OldInv)$ and returns a refined classification according to the invariant Inv.

::::::::::::::::::::::::
`Compose(f,g)`

Inputs two FpG-module homomorphisms $f : M \longrightarrow N$ and $g : L \longrightarrow M$ with $Source(f) = Target(g)$. It returns the composite homomorphism $fg : L \longrightarrow N$.

This also applies to group homomorphisms f, g.

::::::::::::::::::::::::
`HAPcopyright()`

This function provides details of HAP'S GNU public copyright licence.

::::::::::::::::::::::::
`IsLieAlgebraHomomorphism(f)`

Inputs an object f and returns true if f is a homomorphism $f : A \longrightarrow B$ of Lie algebras (preserving the Lie bracket).

::::::::::::::::::::::::
`IsSuperperfect(G)`

Inputs a group G and returns "true" if both the first and second integral homology of G is trivial. Otherwise, it returns "false".

::::::::::::::::::::::::
`MakeHAPManual()`

This function creates the manual for HAP from an XML file.

::::::::::::::::::::::::
`PermToMatrixGroup(G,n)`

Inputs a permutation group G and its degree n. Returns a bijective homomorphism $f : G \longrightarrow M$ where M is a group of permutation matrices.

::::::::::::::::::::::::
`SolutionsMatDestructive(M,B)`

Inputs an $m \times n$ matrix M and a $k \times n$ matrix B over a field. It returns a k×m matrix S satisfying $SM = B$.

The function will leave matrix M unchanged but will probably change matrix B.

(This is a trivial rewrite of the standard GAP function *SolutionMatDestructive*(<*mat*>,<*vec*>) .)

::::::::::::::::::::::::
`LinearHomomorphismsPersistenceMat(L)`

Inputs a composable sequence L of vector space homomorphisms. It returns an integer matrix containing the dimensions of the images of the various composites. The sequence L is determined up to isomorphism by this matrix.

::::::::::::::::::::::::
`NormalSeriesToQuotientHomomorphisms(L)`

Inputs an (increasing or decreasing) chain L of normal subgroups in some group G. This G is the largest group in the chain. It returns the sequence of composable group homomorphisms $G/L[i] \to G/L[i+/-1]$.

::::::::::::::::::::::::
`TestHap()`

This runs a representative sample of HAP functions and checks to see that they produce the correct output.

Index