ComplexesForCAP

Gap package to create (co)chain complexes category of a given Cap category

2018.06.15

1 June 2018

Kamal Saleh

Kamal Saleh
Email: kamal.saleh@uni-siegen.de
Homepage: https://github.com/kamalsaleh/
Address: Walter-Flex-Str. 3
57068 Siegen
Germany

Contents

Chapter 1

Complexes categories

1.1 Constructing chain and cochain categories

1.1.1 IsChainOrCochainComplexCategory (for IsCapCategory)

▷ `IsChainOrCochainComplexCategory(`*arg*`)` (filter)

Returns: `true` or `false`

Gap-categories of the chain or cochain complexes category.

1.1.2 IsChainComplexCategory (for IsChainOrCochainComplexCategory)

▷ `IsChainComplexCategory(`*arg*`)` (filter)

Returns: `true` or `false`

Gap-categories of the chain complexes category.

1.1.3 IsCochainComplexCategory (for IsChainOrCochainComplexCategory)

▷ `IsCochainComplexCategory(`*arg*`)` (filter)

Returns: `true` or `false`

Gap-category of the cochain complexes category.

1.1.4 ChainComplexCategory (for IsCapCategory)

▷ `ChainComplexCategory(`*A*`)` (attribute)

Returns: a CAP category

Creates the chain complex category $\mathrm{Ch}_\bullet(A)$ an additive category A. If you want to contruct the category without finalizing it so that you can add your own methods, you can run the command `ChainComplexCategory(A:FinalizeCategory := false)`.

1.1.5 CochainComplexCategory (for IsCapCategory)

▷ `CochainComplexCategory(`*A*`)` (attribute)

Returns: a CAP category

Creates the cochain complex category $\mathrm{Ch}^\bullet(A)$ an additive category A. If you want to contruct the category without finalizing it so that you can add your own methods, you can run the command `CochainComplexCategory(A:FinalizeCategory := false)`.

1.1.6 UnderlyingCategory (for IsChainOrCochainComplexCategory)

▷ `UnderlyingCategory(B)` (attribute)

Returns: a CAP category

The input is a chain or cochain complex category $B = C(A)$ constructed by one of the previous commands. The outout is A

1.1.7 AddIsNullHomotopic (for IsCapCategory, IsFunction)

▷ `AddIsNullHomotopic(A, F)` (operation)

Returns: `true` or `false`

The input is chain (or cochain category) $Ch(A)$ of some additive category A and a function F. This operation adds the given function F to the category $Ch(A)$ for the basic operation `IsNullHomotopic`. So, F should be a function whose input is a chain or cochain morphism $\phi \in Ch(A)$ and output is *true* if ϕ is null-homotopic and *false* otherwise.

1.2 Examples

Let $\mathbb{Q}$ be the field of rationals and let $\mathrm{Vec}_{\mathbb{Q}}$ be the category of $\mathbb{Q}$-vector spaces. The cochain complex category of $\mathrm{Vec}_{\mathbb{Q}}$ can be constructed as follows

Example
```
gap> LoadPackage( "LinearAlgebraForCap" );;
gap> LoadPackage( "ComplexesForCAP" );;
gap> Q := HomalgFieldOfRationals( );;
gap> matrix_category := MatrixCategory( Q );
Category of matrices over Q
gap> cochain_cat := CochainComplexCategory( matrix_category );
Cochain complexes category over category of matrices over Q
```

Chapter 2

Complexes

2.1 Categories and filters

2.1.1 IsChainOrCochainComplex (for IsCapCategoryObject)

▷ `IsChainOrCochainComplex(C)` (filter)
▷ `IsChainComplex(C)` (filter)
▷ `IsCochainComplex(C)` (filter)
▷ `IsBoundedBelowChainOrCochainComplex(C)` (filter)
▷ `IsBoundedAboveChainOrCochainComplex(C)` (filter)
▷ `IsBoundedChainOrCochainComplex(C)` (filter)
▷ `IsBoundedBelowChainComplex(C)` (filter)
▷ `IsBoundedAboveChainComplex(C)` (filter)
▷ `IsBoundedChainComplex(C)` (filter)
▷ `IsBoundedBelowCochainComplex(C)` (filter)
▷ `IsBoundedAboveCochainComplex(C)` (filter)
▷ `IsBoundedCochainComplex(C)` (filter)

Returns: `true` or `false`

Gap-categories for chain and cochain complexes.

2.2 Creating chain and cochain complexes

2.2.1 ChainComplex (for IsCapCategory, IsZList)

▷ `ChainComplex(A, diffs)` (operation)
▷ `CochainComplex(A, diffs)` (operation)

Returns: a chain complex

The input is category `A` and an infinite list `diffs`. The output is the chain (resp. cochain) complex $M_\bullet \in \mathrm{Ch}(A)$ ($M^\bullet \in \mathrm{Ch}^\bullet(A)$) where $d_i^M = \mathrm{diffs}[i]$($d^i_M = \mathrm{diffs}[i]$).

2.2.2 ChainComplex (for IsDenseList, IsInt)

▷ `ChainComplex(diffs, n)` (operation)
▷ `CochainComplex(diffs, n)` (operation)

Returns: a (co)chain complex

The input is a finite dense list `diffs` and an integer `n` . The output is the chain (resp. cochain) complex $M_\bullet \in \mathrm{Ch}(A)$ $(M^\bullet \in \mathrm{Ch}^\bullet(A))$ where $d_n^M := \mathrm{diffs}[1](d_M^n := \mathrm{diffs}[1]), d_{n+1}^M = \mathrm{diffs}[2](d_M^{n+1} := \mathrm{diffs}[2])$, etc.

2.2.3 ChainComplex (for IsDenseList)

▷ `ChainComplex(diffs)` (operation)

▷ `CochainComplex(diffs)` (operation)

Returns: a (co)chain complex

The same as the previous operations but with $n = 0$.

2.2.4 StalkChainComplex (for IsCapCategoryObject, IsInt)

▷ `StalkChainComplex(diffs, n)` (operation)

▷ `StalkCochainComplex(diffs, n)` (operation)

Returns: a (co)chain complex

The input is an object $M \in A$. The output is chain (resp. cochain) complex $M_\bullet \in \mathrm{Ch}_\bullet(A)(M^\bullet \in \mathrm{Ch}^\bullet(A))$ where $M_n = M(M^n = M)$ and $M_i = 0(M^i = 0)$ whenever $i \neq n$.

2.2.5 ChainComplexWithInductiveSides (for IsCapCategoryMorphism, IsFunction, IsFunction)

▷ `ChainComplexWithInductiveSides(d, G, F)` (operation)

Returns: a chain complex

The input is a morphism $d \in A$ and two functions F, G. The output is chain complex $M_\bullet \in \mathrm{Ch}_\bullet(A)$ where $d_0^M = d$ and $d_i^M = G^i(d)$ for all $i \leq -1$ and $d_i^M = F^i(d)$ for all $i \geq 1$.

2.2.6 CochainComplexWithInductiveSides (for IsCapCategoryMorphism, IsFunction, IsFunction)

▷ `CochainComplexWithInductiveSides(d, G, F)` (operation)

Returns: a cochain complex

The input is a morphism $d \in A$ and two functions F, G. The output is cochain complex $M^\bullet \in \mathrm{Ch}^\bullet(A)$ where $d_M^0 = d$ and $d_M^i = G^i(d)$ for all $i \leq -1$ and $d_M^i = F^i(d)$ for all $i \geq 1$.

2.2.7 ChainComplexWithInductiveNegativeSide (for IsCapCategoryMorphism, IsFunction)

▷ `ChainComplexWithInductiveNegativeSide(d, G)` (operation)

Returns: a chain complex

The input is a morphism $d \in A$ and a functions G. The output is chain complex $M_\bullet \in \mathrm{Ch}_\bullet(A)$ where $d_0^M = d$ and $d_i^M = G^i(d)$ for all $i \leq -1$ and $d_i^M = 0$ for all $i \geq 1$.

2.2.8 ChainComplexWithInductivePositiveSide (for IsCapCategoryMorphism, IsFunction)

▷ `ChainComplexWithInductivePositiveSide(d, F)` (operation)

Returns: a chain complex

The input is a morphism $d \in A$ and a functions F. The output is chain complex $M_\bullet \in \mathrm{Ch}_\bullet(A)$ where $d_0^M = d$ and $d_i^M = F^i(d)$ for all $i \geq 1$ and $d_i^M = 0$ for all $i \leq 1$.

2.2.9 CochainComplexWithInductiveNegativeSide (for IsCapCategoryMorphism, IsFunction)

▷ `CochainComplexWithInductiveNegativeSide(d, G)` (operation)

Returns: a cochain complex

The input is a morphism $d \in A$ and a functions G. The output is cochain complex $M^\bullet \in \mathrm{Ch}^\bullet(A)$ where $d_M^0 = d$ and $d_M^i = G^i(d)$ for all $i \leq -1$ and $d_M^i = 0$ for all $i \geq 1$.

2.2.10 CochainComplexWithInductivePositiveSide (for IsCapCategoryMorphism, IsFunction)

▷ `CochainComplexWithInductivePositiveSide(d, F)` (operation)

Returns: a cochain complex

The input is a morphism $d \in A$ and a functions F. The output is cochain complex $M^\bullet \in \mathrm{Ch}^\bullet(A)$ where $d_M^0 = d$ and $d_M^i = F^i(d)$ for all $i \geq 1$ and $d_M^i = 0$ for all $i \leq 1$.

2.3 Attributes

2.3.1 Differentials (for IsChainOrCochainComplex)

▷ `Differentials(C)` (attribute)

Returns: an infinite list

The command returns the differentials of the chain or cochain complex as an infinite list.

2.3.2 Objects (for IsChainOrCochainComplex)

▷ `Objects(C)` (attribute)

Returns: an infinite list

The command returns the objects of the chain or cochain complex as an infinite list.

2.3.3 CatOfComplex (for IsChainOrCochainComplex)

▷ `CatOfComplex(C)` (attribute)

Returns: a Cap category

The command returns the category in which all objects and differentials of C live.

2.4 Operations

2.4.1 \[\] (for IsChainOrCochainComplex, IsInt)

▷ `\[\](C, i)` (operation)

Returns: an object

The command returns the object of the chain or cochain complex in index i.

2.4.2 \^ (for IsChainOrCochainComplex, IsInt)

▷ `\^(C, i)` (operation)

Returns: a morphism

The command returns the differential of the chain or cochain complex in index i.

2.4.3 CyclesAt (for IsChainOrCochainComplex, IsInt)

▷ `CyclesAt(C, n)` (operation)

Returns: a morphism

The input is a chain or cochain complex C and an integer n. The output is the kernel embedding of the differential in index n.

2.4.4 BoundariesAt (for IsChainOrCochainComplex, IsInt)

▷ `BoundariesAt(C, n)` (operation)

Returns: a morphism

The input is a chain (resp. cochain) complex C and an integer n. The output is the image embeddin of $i+1$'th (resp. $i-1$'th) differential of C.

2.4.5 GeneralizedEmbeddingOfHomologyAt (for IsChainComplex, IsInt)

▷ `GeneralizedEmbeddingOfHomologyAt(C, n)` (operation)

Returns: a generalized morphism

The input is a chain complex category and an integer n. The output is the generalized embedding (defined by span) of the homology object at index n.

$$\begin{tikzcd}
C_{n-1} & C_n \arrow[l, "d_n"'] & C_{n+1} \arrow[l, "d_{n+1}"'] \\
H^n_\bullet(C) \arrow[ur, Rightarrow, "(\tau,\iota)^{\mathrm{span}}"] & \ker(d_n) \arrow[l, two heads, "\tau"] \arrow[u, tail, "\iota"] & \mathrm{im}(d_{n+1}) \arrow[l, "\beta"] \arrow[ul, tail, "\alpha"']
\end{tikzcd}$$

$$\iota := \texttt{KernelEmbedding}(d_n)$$

$$\alpha := \texttt{ImageEmbedding}(d_{n+1})$$

$$\beta := \texttt{KernelLift}(d_n, \alpha)$$

$$\tau := \texttt{CokernelProjection}(\beta)$$

2.4.6 GeneralizedProjectionOntoHomologyAt (for IsChainComplex, IsInt)

▷ `GeneralizedProjectionOntoHomologyAt(C, n)` (operation)

Returns: a generalized morphism

The input is a chain complex category and an integer n. The output is the generalized embedding (defined by span) on the homology object at index n.

$$\begin{tikzcd}
C_{n-1} & C_n \arrow[l, "d_n"'] \arrow[dl, Rightarrow, "(\iota,\tau)^{\mathrm{span}}"'] & C_{n+1} \arrow[l, "d_{n+1}"'] \\
H_\bullet^n(C) & \ker(d_n) \arrow[l, two heads, "\tau"] \arrow[u, tail, "\iota"] & \mathrm{im}(d_{n+1}) \arrow[l, "\beta"] \arrow[ul, tail, "\alpha"']
\end{tikzcd}$$

$$\iota := \texttt{KernelEmbedding}(d_n)$$

$$\alpha := \texttt{ImageEmbedding}(d_{n+1})$$

$$\beta := \texttt{KernelLift}(d_n, \alpha)$$

$$\tau := \texttt{CokernelProjection}(\beta)$$

2.4.7 GeneralizedEmbeddingOfCohomologyAt (for IsCochainComplex, IsInt)

▷ `GeneralizedEmbeddingOfCohomologyAt(C, n)` (operation)

Returns: a generalized morphism

The input is a chain complex category and an integer n. The output is the generalized embedding (defined by span) of the cohomology object at index n.

$$\begin{tikzcd}
C^{n-1} \arrow[r, "d^{n-1}"] & C^n \arrow[r, "d^n"] & C^{n+1} \\
\mathrm{im}(d^{n-1}) \arrow[r, "\beta"'] \arrow[ur, tail, "\alpha"] & \ker(d^n) \arrow[r, two heads, "\tau"'] \arrow[u, tail, "\iota"] & H_n^\bullet(C) \arrow[ul, Rightarrow, "(\tau,\iota)^{\mathrm{span}}"']
\end{tikzcd}$$

$$\iota := \texttt{KernelEmbedding}(d^n)$$

$$\alpha := \texttt{ImageEmbedding}(d^{n-1})$$

$$\beta := \texttt{KernelLift}(d^n, \alpha)$$

$$\tau := \texttt{CokernelProjection}(\beta)$$

2.4.8 GeneralizedProjectionOntoCohomologyAt (for IsCochainComplex, IsInt)

▷ `GeneralizedProjectionOntoCohomologyAt(C, n)` (operation)

Returns: a generalized morphism

The input is a chain complex category and an integer n. The output is the generalized projection (defined by span) on the cohomology object at index n.

$$
\begin{tikzcd}
C^{n-1} \arrow[r, "d^{n-1}"] & C^n \arrow[r, "d^n"] \arrow[dr, Rightarrow, "(\iota, \tau)^{\mathrm{span}}"] & C^{n+1} \\
\mathrm{im}(d^{n-1}) \arrow[ur, tail, "\alpha"] \arrow[r, "\beta"'] & \ker(d^n) \arrow[u, tail, "\iota"'] \arrow[r, two heads, "\tau"'] & H_n^\bullet(C)
\end{tikzcd}
$$

$$\iota := \texttt{KernelEmbedding}(d^n)$$

$$\alpha := \texttt{ImageEmbedding}(d^{n-1})$$

$$\beta := \texttt{KernelLift}(d^n, \alpha)$$

$$\tau := \texttt{CokernelProjection}(\beta)$$

2.4.9 DefectOfExactnessAt (for IsChainOrCochainComplex, IsInt)

▷ `DefectOfExactnessAt(C, n)` (operation)
▷ `CohomologyAt(C, n)` (operation)
▷ `HomologyAt(C, n)` (operation)

Returns: a object

The input is a chain (resp. cochain) complex C and an integer n. The outout is the homology (resp. cohomology) object of C in index n.

2.4.10 HomologySupport (for IsChainComplex, IsInt, IsInt)

▷ `HomologySupport(C, m, n)` (operation)
▷ `CohomologySupport(C, m, n)` (operation)

Returns: a list

The input is a chain (resp. cochain) complex C and two integers m, n. The outout is the list of indices where the homology (resp. cohomology) objects of C are not zero.

2.4.11 HomologySupport (for IsBoundedChainComplex)

▷ `HomologySupport(C)` (operation)
▷ `CohomologySupport(C)` (operation)

Returns: a list

The same as above but for bounded complexes.

2.4.12 ObjectsSupport (for IsChainOrCochainComplex, IsInt, IsInt)

▷ `ObjectsSupport(C, m, n)` (operation)
▷ `DifferentialsSupport(C, m, n)` (operation)

Returns: a list

The input is a chain (resp. cochain) complex C and two integers m, n. The outout is the list of indices where the objects (resp. differentials) of C are not zero.

2.4.13 ObjectsSupport (for IsBoundedChainOrCochainComplex)

▷ `ObjectsSupport(C)` (operation)

▷ `DifferentialsSupport(C)` (operation)

Returns: a list

The same as above but for bounded complexes.

2.4.14 IsWellDefined (for IsChainOrCochainComplex, IsInt, IsInt)

▷ `IsWellDefined(C, m, n)` (operation)

Returns: a object

The input is a chain (resp. cochain) complex C and two integers m,n. The output is true when C is well defined in the interval $[m,\dots,n]$ and false otherwise.

2.4.15 IsWellDefined (for IsBoundedChainOrCochainComplex)

▷ `IsWellDefined(arg)` (property)

Returns: `true` or `false`

2.4.16 IsExactInIndex (for IsChainOrCochainComplex, IsInt)

▷ `IsExactInIndex(C, n)` (operation)

Returns: true or false

The input is a chain or cochain complex C and an integer n. The outout is *`true`* if C is exact in i. Otherwise the output is *`false`*.

2.4.17 SetUpperBound (for IsChainOrCochainComplex, IsInt)

▷ `SetUpperBound(C, n)` (operation)

Returns: Side effect

The command sets an upper bound n to the chain (resp. cochain) complex C. This means $C_{i\geq n} = 0(C^{\geq n} = 0)$. This upper bound will be called *active* upper bound of C. If C already has an active upper bound m, then m will be replaced by n only if n is better upper bound than m, i.e., $n \leq m$. If C has an active lower bound l and $n \leq l$ then the upper bound will set to equal l and as a consequence C will be set to zero.

2.4.18 SetLowerBound (for IsChainOrCochainComplex, IsInt)

▷ `SetLowerBound(C, n)` (operation)

Returns: Side effect

The command sets an lower bound n to the chain (resp. cochain) complex C. This means $C_{i\leq n} = 0(C^{\leq n} = 0)$. This lower bound will be called *active* lower bound of C. If C already has an active lower bound m, then m will be replaced by n only if n is better lower bound than m, i.e., $n \geq m$. If C has an active upper bound u and $n \geq u$ then the lower bound will set to equal u and as a consequence C will be set to zero.

2.4.19 HasActiveUpperBound (for IsChainOrCochainComplex)

▷ `HasActiveUpperBound(C)` (operation)

Returns: true or false

The input is chain or cochain complex. The output is *true* if an upper bound has been set to C and *false* otherwise.

2.4.20 HasActiveLowerBound (for IsChainOrCochainComplex)

▷ `HasActiveLowerBound(C)` (operation)

Returns: true or false

The input is chain or cochain complex. The output is *true* if a lower bound has been set to C and *false* otherwise.

2.4.21 ActiveUpperBound (for IsChainOrCochainComplex)

▷ `ActiveUpperBound(C)` (operation)

Returns: an integer

The input is chain or cochain complex. The output is its active upper bound if such has been set to C. Otherwise we get error.

2.4.22 ActiveLowerBound (for IsChainOrCochainComplex)

▷ `ActiveLowerBound(C)` (operation)

Returns: an integer

The input is chain or cochain complex. The output is its active lower bound if such has been set to C. Otherwise we get error.

2.4.23 Display (for IsChainOrCochainComplex, IsInt, IsInt)

▷ `Display(C, m, n)` (operation)

Returns: nothing

The input is chain or cochain complex C and two integers m and n. The command displays all components of C between the indices m,n.

2.5 Truncations

2.5.1 GoodTruncationBelow (for IsChainComplex, IsInt)

▷ `GoodTruncationBelow(C, n)` (operation)

Returns: chain complex

Let $C_\bullet$ be chain complex. A good truncation of $C_\bullet$ below n is the chain complex $\tau_{\geq n}C_\bullet$ whose differentials are defined by

$$d_i^{\tau_{\geq n}C_\bullet} = \begin{cases} 0:0 \leftarrow 0 & \text{if} \quad i<n, \\ 0:0 \leftarrow Z_n & \text{if} \quad i=n, \\ \mathrm{KernelLift}(d_n^C, d_{n+1}^C) : Z_n \leftarrow C_{n+1} & \text{if} \quad i=n+1, \\ d_i^C : C_{i-1} \leftarrow C_i & \text{if} \quad i>n+1. \end{cases}$$

where Z_n is the cycle in index n. It can be shown that $H_i(\tau_{\geq n}C_\bullet) = 0$ for $i < n$ and $H_i(\tau_{\geq n}C_\bullet) = H_i(C_\bullet)$ for $i \geq n$.

$$
\begin{tikzcd}
C_\bullet & \cdots & C_{n-1} \arrow[l] & C_n \arrow[l] & C_{n+1} \arrow[l] \arrow[dl] & C_{n+2} \arrow[l] & \cdots \arrow[l] \\
\tau_{\geq n}C_\bullet & \cdots & 0 \arrow[l] & Z_n \arrow[l] \arrow[u, hook] & & &
\end{tikzcd}
$$

2.5.2 GoodTruncationAbove (for IsChainComplex, IsInt)

▷ `GoodTruncationAbove(C, n)` (operation)

Returns: chain complex

Let $C_\bullet$ be chain complex. A good truncation of $C_\bullet$ above n is the quotient chain complex $\tau_{<n}C_\bullet = C_\bullet/\tau_{\geq n}C_\bullet$. It can be shown that $H_i(\tau_{<n}C_\bullet) = 0$ for $i \geq n$ and $H_i(\tau_{<n}C_\bullet) = H_i(C_\bullet)$ for $i < n$.

2.5.3 GoodTruncationAbove (for IsCochainComplex, IsInt)

▷ `GoodTruncationAbove(C, n)` (operation)

Let $C^\bullet$ be cochain complex. A good truncation of $C^\bullet$ above n is the cochain complex $\tau^{\leq n}C^\bullet$ whose differentials are defined by

$$
d^i_{\tau^{\leq n}C^\bullet} = \begin{cases} 0 : 0 \to 0 & \text{if} \quad i > n, \\ 0 : Z^n \to 0 & \text{if} \quad i = n, \\ \mathrm{KernelLift}(d^n_C, d^{n-1}_C) : C^{n-1} \to Z^n & \text{if} \quad i = n-1, \\ d^i_C : C^i \to C^{i+1} & \text{if} \quad i < n-1. \end{cases}
$$

where Z_n is the cycle in index n. It can be shown that $H^i(\tau^{\leq n}C^\bullet) = 0$ for $i > n$ and $H^i(\tau^{\leq n}C^\bullet) = H_i(C^\bullet)$ for $i \leq n$.

$$
\begin{tikzcd}
\cdots \arrow[r] & C^{n-2} \arrow[r] & C^{n-1} \arrow[r] \arrow[dr] & C^n \arrow[r] & C^{n+1} \arrow[r] & \cdots & C^\bullet \\
 & & & Z^n \arrow[u, hook] \arrow[r] & 0 \arrow[r] & \cdots & \tau^{\leq n}C^\bullet
\end{tikzcd}
$$

2.5.4 GoodTruncationBelow (for IsCochainComplex, IsInt)

▷ `GoodTruncationBelow(C, n)` (operation)

Returns: cochain complex

Let $C^\bullet$ be cochain complex. A good truncation of $C^\bullet$ below n is the quotient cochain complex $\tau^{>n}C^\bullet = C^\bullet/\tau^{\leq n}C^\bullet$. It can be shown that $H^i(\tau^{>n}C^\bullet) = 0$ for $i \leq n$ and $H^i(\tau^{>n}C^\bullet) = H_i(C^\bullet)$ for $i > n$.

2.5.5 BrutalTruncationBelow (for IsChainComplex, IsInt)

▷ `BrutalTruncationBelow(C, n)` (operation)

Returns: chain complex

Let $C_\bullet$ be chain complex. A brutal truncation of $C_\bullet$ below n is the chain complex $\sigma_{\geq n}C_\bullet$ where $(\sigma_{\geq n}C_\bullet)_i = C_i$ when $i \geq n$ and $(\sigma_{\geq n}C_\bullet)_i = 0$ otherwise.

2.5.6 BrutalTruncationAbove (for IsChainComplex, IsInt)

▷ `BrutalTruncationAbove(C, n)` (operation)

Returns: chain complex

Let $C_\bullet$ be chain complex. A brutal truncation of $C_\bullet$ above n is the chain quotient chain complex $\sigma_{<n}C_\bullet := C_\bullet/\sigma_{\geq n}C_\bullet$. Hence $(\sigma_{<n}C_\bullet)_i = C_i$ when $i < n$ and $(\sigma_{<n}C_\bullet)_i = 0$ otherwise.

2.5.7 BrutalTruncationAbove (for IsCochainComplex, IsInt)

▷ `BrutalTruncationAbove(C, n)` (operation)

Returns: chain complex

Let $C^\bullet$ be cochain complex. A brutal truncation of $C_\bullet$ above n is the cochain complex $\sigma^{\leq n}C^\bullet$ where $(\sigma^{\leq n}C^\bullet)_i = C_i$ when $i \leq n$ and $(\sigma^{\leq n}C^\bullet)_i = 0$ otherwise.

2.5.8 BrutalTruncationBelow (for IsCochainComplex, IsInt)

▷ `BrutalTruncationBelow(C, n)` (operation)

Returns: chain complex

Let $C^\bullet$ be cochain complex. A brutal truncation of $C^\bullet$ bellow n is the quotient cochain complex $\sigma^{>n}C^\bullet := C^\bullet/\sigma^{\leq n}C_\bullet$. Hence $(\sigma^{>n}C^\bullet)_i = C_i$ when $i > n$ and $(\sigma^{<n}C^\bullet)_i = 0$ otherwise.

2.6 Examples

Below we define the complex

$$\begin{array}{ccccccccccccccc}
\cdots & & 2 & & 3 & & 4 & & 5 & & 6 & & 7 & & \cdots\\
\cdots & \longrightarrow & 0 & \longrightarrow & \mathbb{Q}^{1\times 1} & \xrightarrow{\begin{pmatrix}1 & 3\end{pmatrix}} & \mathbb{Q}^{1\times 2} & \xrightarrow{\begin{pmatrix}0\\0\end{pmatrix}} & \mathbb{Q}^{1\times 1} & \xrightarrow{\begin{pmatrix}2 & 6\end{pmatrix}} & \mathbb{Q}^{1\times 2} & \longrightarrow & 0 & \longrightarrow & \cdots
\end{array}$$

Example

```
(continued)
gap> A := VectorSpaceObject( 1, Q );
<A vector space object over Q of dimension 1>
gap> B := VectorSpaceObject( 2, Q );
<A vector space object over Q of dimension 2>
gap> f := VectorSpaceMorphism( A, HomalgMatrix( [ [ 1, 3 ] ], 1, 2, Q ), B );
<A morphism in Category of matrices over Q>
gap> g := VectorSpaceMorphism( B, HomalgMatrix( [ [ 0 ], [ 0 ] ], 2, 1, Q ), A );
<A morphism in Category of matrices over Q>
gap> C := CochainComplex( [ f, g, 2*f ], 3 );
<A bounded object in cochain complexes category over category of matrices over Q
with active lower bound 2 and active upper bound 7>
gap> ActiveUpperBound( C );
7
gap> ActiveLowerBound( C );
2
gap> C[ 1 ];
```

```
<A vector space object over Q of dimension 0>
gap> C[ 3 ];
<A vector space object over Q of dimension 1>
gap> C^3;
<A morphism in Category of matrices over Q>
gap> C^3 = f;
true
gap> Display( CyclesAt( C, 4 ) );
[ [  1,  0 ],
  [  0,  1 ] ]

A split monomorphism in Category of matrices over Q
gap> diffs := Differentials( C );
<An infinite list>
gap> diffs[ 1 ];
<A zero, isomorphism in Category of matrices over Q>
gap> diffs[ 10000 ];
<A zero, isomorphism in Category of matrices over Q>
gap> objs := Objects( C );
<An infinite list>
gap> DefectOfExactnessAt( C, 4 );
<A vector space object over Q of dimension 1>
gap> DefectOfExactnessAt( C, 3 );
<A vector space object over Q of dimension 0>
gap> IsExactInIndex( C, 4 );
false
gap> IsExactInIndex( C, 3 );
true
gap> C;
<A not cyclic, bounded object in cochain complexes category over category of
matrices over Q with active lower bound 2 and active upper bound 7>
gap> P := CochainComplex( matrix_category, diffs );
<An object in Cochain complexes category over category of matrices over Q>
gap> SetUpperBound( P, 15 );
gap> P;
<A bounded from above object in cochain complexes category over category of
matrices over Q with active upper bound 15>
gap> SetUpperBound( P, 20 );
gap> P;
<A bounded from above object in cochain complexes category over category of
matrices over Q with active upper bound 15>
gap> ActiveUpperBound( P );
15
gap> SetUpperBound( P, 7 );
gap> P;
<A bounded from above object in cochain complexes category over category of
matrices over Q with active upper bound 7>
gap> ActiveUpperBound( P );
7
```

Chapter 3

Complexes morphisms

3.1 Categories and filters

3.1.1 IsChainOrCochainMorphism (for IsCapCategoryMorphism)

▷ `IsChainOrCochainMorphism(phi)` (filter)
▷ `IsBoundedBelowChainOrCochainMorphism(phi)` (filter)
▷ `IsBoundedAboveChainOrCochainMorphism(phi)` (filter)
▷ `IsBoundedChainOrCochainMorphism(phi)` (filter)
▷ `IsChainMorphism(phi)` (filter)
▷ `IsBoundedBelowChainMorphism(phi)` (filter)
▷ `IsBoundedAboveChainMorphism(phi)` (filter)
▷ `IsBoundedChainMorphism(phi)` (filter)
▷ `IsCochainMorphism(phi)` (filter)
▷ `IsBoundedBelowCochainMorphism(phi)` (filter)
▷ `IsBoundedAboveCochainMorphism(phi)` (filter)
▷ `IsBoundedCochainMorphism(phi)` (filter)

Returns: `true` or `false`

Gap-categories for chain and cochains morphisms.

3.2 Creating chain and cochain morphisms

3.2.1 ChainMorphism (for IsChainComplex, IsChainComplex, IsZList)

▷ `ChainMorphism(C, D, l)` (operation)

Returns: a chain morphism

The input is two chain complexes C, D and an infinite list l. The output is the chain morphism $\phi : C \to D$ defined by $\phi_i := l[i]$.

3.2.2 ChainMorphism (for IsChainComplex, IsChainComplex, IsDenseList, IsInt)

▷ `ChainMorphism(C, D, l, k)` (operation)

Returns: a chain morphism

The input is two chain complexes C, D, dense list l and an integer k. The output is the chain morphism $\phi : C \to D$ such that $\phi_k = l[1]$, $\phi_{k+1} = l[2]$, etc.

3.2.3 ChainMorphism (for IsDenseList, IsInt, IsDenseList, IsInt, IsDenseList, IsInt)

▷ `ChainMorphism(c, m, d, n, l, k)` (operation)

Returns: a chain morphism

The output is the chain morphism $\phi : C \to D$, where $d_m^C = c[1], d_{m+1}^C = c[2]$, etc. $d_n^D = d[1], d_{n+1}^D = d[2]$, etc. and $\phi_k = l[1]$, $\phi_{k+1} = l[2]$, etc.

3.2.4 CochainMorphism (for IsCochainComplex, IsCochainComplex, IsZList)

▷ `CochainMorphism(C, D, l)` (operation)

Returns: a cochain morphism

The input is two cochain complexes C,D and an infinite list l. The output is the cochain morphism $\phi : C \to D$ defined by $\phi_i := l[i]$.

3.2.5 CochainMorphism (for IsCochainComplex, IsCochainComplex, IsDenseList, IsInt)

▷ `CochainMorphism(C, D, l, k)` (operation)

Returns: a chain morphism

The input is two cochain complexes C,D, dense list l and an integer k. The output is the cochain morphism $\phi : C \to D$ such that $\phi^k = l[1]$, $\phi^{k+1} = l[2]$, etc.

3.2.6 CochainMorphism (for IsDenseList, IsInt, IsDenseList, IsInt, IsDenseList, IsInt)

▷ `CochainMorphism(c, m, d, n, l, k)` (operation)

Returns: a cochain morphism

The output is the cochain morphism $\phi : C \to D$, where $C^m = c[1], C^{m+1} = c[2]$, etc. $D^n = d[1], D^{n+1} = d[2]$, etc. and $\phi^k = l[1]$, $\phi^{k+1} = l[2]$, etc.

3.3 Attributes

3.3.1 Morphisms (for IsChainOrCochainMorphism)

▷ `Morphisms(phi)` (attribute)

Returns: infinite list

The output is morphisms of the chain or cochain morphism as an infinite list.

3.3.2 MappingCone (for IsChainOrCochainMorphism)

▷ `MappingCone(phi)` (attribute)

Returns: complex

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is its mapping cone chain (resp. cochain) complex Cone(ϕ).

3.3.3 NaturalInjectionInMappingCone (for IsChainOrCochainMorphism)

▷ `NaturalInjectionInMappingCone(phi)` (attribute)

Returns: chain (resp. cochain) morphism

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is the natural injection $i : D \to \mathrm{Cone}(\phi)$.

3.3.4 NaturalProjectionFromMappingCone (for IsChainOrCochainMorphism)

▷ `NaturalProjectionFromMappingCone(phi)` (attribute)

Returns: chain (resp. cochain) morphism

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is the natural projection $\pi : \mathrm{Cone}(\phi) \to C[u]$ where $u = -1$ if ϕ is chain morphism and $u = 1$ if ϕ is cochain morphism.

3.3.5 MappingCylinder (for IsChainOrCochainMorphism)

▷ `MappingCylinder(phi)` (attribute)

Returns: complex

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is its mapping cylinder chain (resp. cochain) complex $\mathrm{Cyl}(\phi)$.

3.3.6 NaturalInjectionOfSourceInMappingCylinder (for IsChainOrCochainMorphism)

▷ `NaturalInjectionOfSourceInMappingCylinder(phi)` (attribute)

Returns: morphism

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is the natural embedding $C \to \mathrm{Cyl}(\phi)$.

3.3.7 NaturalInjectionOfRangeInMappingCylinder (for IsChainOrCochainMorphism)

▷ `NaturalInjectionOfRangeInMappingCylinder(phi)` (attribute)

Returns: morphism

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is the natural embedding $D \to \mathrm{Cyl}(\phi)$. This morphism can be proven to be quasi-isomorphism. See Weibel, page 21.

3.3.8 NaturalMorphismFromMappingCylinderInRange (for IsChainOrCochainMorphism)

▷ `NaturalMorphismFromMappingCylinderInRange(phi)` (attribute)

Returns: morphism

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is the natural morphism $\mathrm{Cyl}(\phi) \to D$. It can be shown that D and $\mathrm{Cyl}(\phi)$ are homotopy equivalent. See Weibel, page 21.

3.3.9 NaturalMorphismFromMappingCylinderInMappingCone (for IsChainOrCochainMorphism)

▷ `NaturalMorphismFromMappingCylinderInMappingCone(phi)` (attribute)

Returns: morphism

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is the natural morphism $\mathrm{Cyl}(\phi) \to \mathrm{Cone}(\phi)$. It can be shown that $0 \to C \to \mathrm{Cyl}(\phi) \to \mathrm{Cone}(\phi) \to 0$ is a short exact sequence. See Weibel, page 21.

3.3.10 HomotopyMorphisms (for IsCapCategoryMorphism)

▷ `HomotopyMorphisms(phi)` (attribute)

Returns: Infinite list

The input is a null-homotopic chain (resp. cochain) morphism $\phi : C \to D$. The output is the homotopy morphisms given as an infinite list $(h_i : C_i \to D_{i+1})$ (resp. $(h_i : C_i \to D_{i-1})$).

3.4 Properties

3.4.1 IsQuasiIsomorphism (for IsChainOrCochainMorphism)

▷ `IsQuasiIsomorphism(phi)` (property)

Returns: `true` or `false`

The input a chain (resp. cochain) morphism $\phi : C \to D$. The output is *true* if ϕ is quasi-isomorphism and *false* otherwise. If ϕ is not bounded an error is raised.

3.4.2 IsNullHomotopic (for IsCapCategoryMorphism)

▷ `IsNullHomotopic(phi)` (property)

Returns: `true` or `false`

The input is a chain or cochain morphism ϕ and output is *true* if ϕ is null-homotopic and *false* otherwise.

3.5 Operations

3.5.1 SetUpperBound (for IsChainOrCochainMorphism, IsInt)

▷ `SetUpperBound(phi, n)` (operation)

Returns: a side effect

The command sets an upper bound to the morphism ϕ. An upper bound of ϕ is an integer u with $\phi_{i \geq u} = 0$. The integer u will be called *active* upper bound of ϕ. If ϕ already has an active upper bound, say u', then u' will be replaced by u only if $u \leq u'$.

3.5.2 SetLowerBound (for IsChainOrCochainMorphism, IsInt)

▷ `SetLowerBound(phi, n)` (operation)

Returns: a side effect

The command sets an lower bound to the morphism ϕ. A lower bound of ϕ is an integer l with $\phi_{i \leq l} = 0$. The integer l will be called *active* lower bound of ϕ. If ϕ already has an active lower bound, say l', then l' will be replaced by l only if $l \geq l'$.

3.5.3 HasActiveUpperBound (for IsChainOrCochainMorphism)

▷ `HasActiveUpperBound(phi)` (operation)

Returns: true or false

The input is chain or cochain morphism ϕ. The output is `true` if an upper bound has been set to ϕ and `false` otherwise.

3.5.4 HasActiveLowerBound (for IsChainOrCochainMorphism)

▷ `HasActiveLowerBound(phi)` (operation)

Returns: true or false

The input is chain or cochain morphism ϕ. The output is `true` if a lower bound has been set to ϕ and `false` otherwise.

3.5.5 ActiveUpperBound (for IsChainOrCochainMorphism)

▷ `ActiveUpperBound(phi)` (operation)

Returns: an integer

The input is chain or cochain morphism. The output is its active upper bound if such has been set to ϕ. Otherwise we get error.

3.5.6 ActiveLowerBound (for IsChainOrCochainMorphism)

▷ `ActiveLowerBound(phi)` (operation)

Returns: an integer

The input is chain or cochain morphism. The output is its active lower bound if such has been set to ϕ. Otherwise we get error.

3.5.7 MorphismAt (for IsChainOrCochainMorphism, IsInt)

▷ `MorphismAt(phi, n)` (operation)

Returns: a morphism

The input is chain (resp. cochain) morphism and an integer n. The output is the component of ϕ in index n, i.e., ϕ_n(resp. ϕ^n).

3.5.8 CyclesFunctorialAt (for IsChainOrCochainMorphism, IsInt)

▷ `CyclesFunctorialAt(phi, n)` (operation)

Returns: a morphism

The input is chain (resp. cochain) morphism and an integer n. The output is the morphism between the kernels in index n.

3.5.9 \[\] (for IsChainOrCochainMorphism, IsInt)

▷ `\[\](phi, n)` (operation)

Returns: an integer

The input is chain (resp. cochain) morphism and an integer n. The output is the component of ϕ in index n, i.e., ϕ_n(resp. ϕ^n).

3.5.10 IsQuasiIsomorphism (for IsChainOrCochainMorphism, IsInt, IsInt)

▷ `IsQuasiIsomorphism(phi, n)` (operation)

Returns: an integer

The input is chain (resp. cochain) morphism and an integer n. The output is the component of ϕ in index n, i.e., ϕ_n(resp. ϕ^n).

3.5.11 Display (for IsChainOrCochainMorphism, IsInt, IsInt)

▷ `Display(phi, m, n)` (operation)

The command displays the components of the morphism between m and n.

3.5.12 IsWellDefined (for IsChainOrCochainMorphism, IsInt, IsInt)

▷ `IsWellDefined(true, or, false)` (operation)

The command checks if the morphism is well defined between m and n.

3.6 Examples

Let us define a morphism

$$\begin{tikzcd}
\cdots & 2 & 3 & 4 & 5 & 6 & 7 & \cdots \\
\cdots \arrow[r] & 0 \arrow[r] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} 1 & 3 \end{pmatrix}"] & \mathbb{Q}^{1\times 2} \arrow[r, "\begin{pmatrix} 0 \\ 0 \end{pmatrix}"] \arrow[d, "\begin{pmatrix} 0 \\ 0 \end{pmatrix}"'] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} 2 & 6 \end{pmatrix}"] \arrow[d, "\begin{pmatrix} 10 \end{pmatrix}"] & \mathbb{Q}^{1\times 2} \arrow[r] & 0 \arrow[r] & \cdots \\
\cdots \arrow[r] & 0 \arrow[r] & 0 \arrow[r] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} 5 \end{pmatrix}"'] & \mathbb{Q}^{1\times 1} \arrow[r] & 0 \arrow[r] & 0 \arrow[r] & \cdots
\end{tikzcd}$$

Example

```
(continued)
gap> h := VectorSpaceMorphism( A, HomalgMatrix( [ [ 5 ] ], 1, 1, Q ), A );
<A morphism in Category of matrices over Q>
gap> phi4 := g;
<A morphism in Category of matrices over Q>
gap> phi5 := 2*h;
<A morphism in Category of matrices over Q>
gap> D := CochainComplex( [ h ], 4 );
<A bounded object in cochain complexes category over category of matrices
over Q with active lower bound 3 and active upper bound 6>
gap> phi := CochainMorphism( C, D, [ phi4, phi5 ], 4 );
<A bounded morphism in cochain complexes category over category of matrices
 over Q with active lower bound 3 and active upper bound 6>
```

```
gap> Display( phi[ 5 ] );
[ [ 10 ] ]

A morphism in Category of matrices over Q
gap> ActiveLowerBound( phi );
3
gap> IsZeroForMorphisms( phi );
false
gap> IsExact( D );
true
gap> IsExact( C );
false
```

Now lets define the previous morphism using the command `CochainMorphism(c, m, d, n, l, k)`.

Example
```
(continued)
gap> psi := CochainMorphism( [ f, g, 2*f ], 3, [ h ], 4, [ phi4, phi5 ], 4 );
<A bounded morphism in cochain complexes category over category of matrices
over Q with active lower bound 3 and active upper bound 6>
```

In some cases the morphism can change its lower bound when we apply the function `IsZeroForMorphisms` .

Example
```
(continued)
gap> IsZeroForMorphisms( psi );
false
gap> psi;
<A bounded morphism in cochain complexes category over category of matrices
over Q with active lower bound 4 and active upper bound 6>
```

In the following we compute the mapping cone of ψ and its natural injection and projection.

$$
\begin{tikzcd}
 & \cdots & 2 & 3 & 4 & 5 & 6 & \cdots \\
C: \arrow[d, "\psi"'] & {} \arrow[r] & 0 \arrow[r] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} 1 & 3 \end{pmatrix}"] & \mathbb{Q}^{1\times 2} \arrow[r, "\begin{pmatrix} 0 \\ 0 \end{pmatrix}"] \arrow[d, "\begin{pmatrix} 0 \\ 0 \end{pmatrix}"'] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} 2 & 6 \end{pmatrix}"] \arrow[d, "\begin{pmatrix} 10 \end{pmatrix}"] & \mathbb{Q}^{1\times 2} \arrow[r] & \cdots \\
D: \arrow[d, "i"'] & {} \arrow[r] & 0 \arrow[r] & 0 \arrow[r] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} 5 \end{pmatrix}"'] \arrow[d, "\begin{pmatrix} 0 & 1 \end{pmatrix}"'] & \mathbb{Q}^{1\times 1} \arrow[r] \arrow[d, "\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}"] & 0 \arrow[r] & \cdots \\
Cone(\psi): \arrow[d, "p"'] & {} \arrow[r] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} -1 & -3 \end{pmatrix}"] \arrow[d, "\begin{pmatrix} 1 \end{pmatrix}"'] & \mathbb{Q}^{1\times 2} \arrow[r, "\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}"] \arrow[d, "\begin{pmatrix} 1,0 \\ 0,1 \end{pmatrix}"'] & \mathbb{Q}^{1\times 2} \arrow[r, "\begin{pmatrix} -2 & -6 & 10 \\ 0 & 0 & 5 \end{pmatrix}"] \arrow[d, "\begin{pmatrix} 1 \\ 0 \end{pmatrix}"] & \mathbb{Q}^{1\times 3} \arrow[r] \arrow[d, "\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}"] & 0 \arrow[r] & \cdots \\
C[1]: & {} \arrow[r] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} -1 & -3 \end{pmatrix}"] & \mathbb{Q}^{1\times 2} \arrow[r, "\begin{pmatrix} 0 \\ 0 \end{pmatrix}"] & \mathbb{Q}^{1\times 1} \arrow[r, "\begin{pmatrix} -2 & -6 \end{pmatrix}"] & \mathbb{Q}^{1\times 2} \arrow[r] & 0 \arrow[r] & \cdots
\end{tikzcd}
$$

Example

```
(continued)
gap> cone := MappingCone( psi );
<A bounded object in cochain complexes category over category of matrices over
Q with active lower bound 1 and active upper bound 6>
gap> cone^4;
<A morphism in Category of matrices over Q>
gap> Display( cone^4 );
[ [   -2,   -6,   10 ],
  [    0,    0,    5 ] ]

A morphism in Category of matrices over Q
gap> i := NaturalInjectionInMappingCone( psi );
<A bounded morphism in cochain complexes category over category of matrices over
Q with active lower bound 3 and active upper bound 6>
gap> p := NaturalProjectionFromMappingCone( psi );
<A bounded morphism in cochain complexes category over category of matrices over
Q with active lower bound 1 and active upper bound 6>
```

Chapter 4

Functors

4.1 Basic functors for complex categories.

4.1.1 HomologyFunctorAt (for IsChainComplexCategory, IsCapCategory, IsInt)

▷ `HomologyFunctorAt(Ch_\bullet(A), A, n)` (operation)

▷ `CohomologyFunctorAt(Ch^\bullet(A), A, n)` (operation)

Returns: a functor

The first argument in the input must be the chain (resp. cochain) complex category of an abelian category A, the second argument is `A` and the third argument is an integer `n`. The output is the n'th homology (resp. cohomology) functor $\mathrm{Ch}_\bullet(A) \to A$ (resp. $\mathrm{Ch}^\bullet(A) \to A$)

4.1.2 ShiftFunctor (for IsChainOrCochainComplexCategory, IsInt)

▷ `ShiftFunctor(Comp(A), n)` (operation)

Returns: a functor

The inputs are complex category $\mathrm{Comp}(A)$ and an integer. The output is a the endofunctor $T[n]$ that sends any complex C to $C[n]$ and any complex morphism $\phi : C \to D$ to $\phi[n] : C[n] \to D[n]$. The shift chain complex $C[n]$ of a chain complex C is defined by $C[n]_i = C_{n+i}, d_i^{C[n]} = (-1)^n d_{n+i}^C$ and the same for chain complex morphisms, i.e., $\phi[n]_i = \phi_{n+i}$. The same holds for cochain complexes and morphisms.

4.1.3 UnsignedShiftFunctor (for IsChainOrCochainComplexCategory, IsInt)

▷ `UnsignedShiftFunctor(Comp(A), n)` (operation)

Returns: a functor

The inputs are complex category $\mathrm{Comp}(A)$ and an integer. The output is a the endofunctor $T[n]$ that sends any complex C to $C[n]$ and any complex morphism $\phi : C \to D$ to $\phi[n] : C[n] \to D[n]$. The shift chain complex $C[n]$ of a chain complex C is defined by $C[n]_i = C_{n+i}, d_i^{C[n]} = d_{n+i}^C$ and the same for chain complex morphisms, i.e., $\phi[n]_i = \phi_{n+i}$. The same holds for cochain complexes and morphisms.

4.1.4 ChainToCochainComplexFunctor (for IsChainComplexCategory, IsCochainComplexCategory)

▷ `ChainToCochainComplexFunctor(Ch(A)_\bullet, Ch(A)^\bullet)` (operation)

Returns: a functor

The arguments are $\mathrm{Ch}_\bullet(A)$ and $\mathrm{Ch}^\bullet(A)$ for some category A. The output is the functor $F : \mathrm{Ch}_\bullet(A) \to \mathrm{Ch}^\bullet(A)$ defined by $C_\bullet \mapsto C^\bullet$ for any for any chain complex $C_\bullet \in \mathrm{Ch}_\bullet(A)$ and by $\phi_\bullet \mapsto \phi^\bullet$ for any morphism $\phi_\bullet$ where $C_i^\bullet = C_\bullet^{-i}$ and $\phi_i^\bullet = \phi_\bullet^{-i}$ for any $i \in \mathbb{Z}$.

4.1.5 CochainToChainComplexFunctor (for IsCochainComplexCategory, IsChainComplexCategory)

▷ `CochainToChainComplexFunctor(Ch(A)^\bullet, Ch(A)_\bullet)` (operation)

Returns: a functor

The arguments are $\mathrm{Ch}^\bullet(A)$ and $\mathrm{Ch}_\bullet(A)$ for some category A. The output is the functor $F : \mathrm{Ch}^\bullet(A) \to \mathrm{Ch}_\bullet(A)$ defined by $C^\bullet \mapsto C_\bullet$ for any for any chain complex $C^\bullet \in \mathrm{Ch}^\bullet(A)$ and by $\phi^\bullet \mapsto \phi_\bullet$ for any morphism $\phi^\bullet$ where $C_\bullet^i = C_{-i}^\bullet$ and $\phi_\bullet^i = \phi_{-i}^\bullet$ for any $i \in \mathbb{Z}$.

4.1.6 ExtendFunctorToChainComplexCategoryFunctor (for IsCapFunctor)

▷ `ExtendFunctorToChainComplexCategoryFunctor(F)` (operation)

Returns: a functor

The input is a functor $F : A \to B$. The output is its extention functor

$$\mathrm{Ch}_\bullet F : \mathrm{Ch}_\bullet(A) \to \mathrm{Ch}_\bullet(B).$$

$$\begin{tikzcd}
C_\bullet \arrow[d, "\phi"'] & \cdots & C_{n-1} \arrow[l, "d_{n-1}"'] \arrow[d, "\phi_{n-1}"'] & C_n \arrow[l, "d_n"'] \arrow[d, "\phi_n"'] & C_{n+1} \arrow[l, "d_{n+1}"'] \arrow[d, "\phi_{n+1}"'] & \cdots \arrow[l] \\
D_\bullet & \cdots & D_{n-1} \arrow[l, "d_{n-1}"] & D_n \arrow[l, "d_n"] & D_{n+1} \arrow[l, "d_{n+1}"] & \cdots \arrow[l]
\end{tikzcd}$$

$$\begin{tikzcd}
\mathrm{Ch}_\bullet F(C_\bullet) \arrow[d, "F(\phi)"'] & \cdots & F(C_{n-1}) \arrow[l, "F(d_{n-1})"'] \arrow[d, "F(\phi_{n-1})"'] & F(C_n) \arrow[l, "F(d_n)"'] \arrow[d, "F(\phi_n)"'] & F(C_{n+1}) \arrow[l, "F(d_{n+1})"'] \arrow[d, "F(\phi_{n+1})"'] & \cdots \arrow[l] \\
\mathrm{Ch}_\bullet F(D_\bullet) & \cdots & F(D_{n-1}) \arrow[l, "F(d_{n-1})"] & F(D_n) \arrow[l, "F(d_n)"] & F(D_{n+1}) \arrow[l, "F(d_{n+1})"] & \cdots \arrow[l]
\end{tikzcd}$$

4.1.7 ExtendFunctorToCochainComplexCategoryFunctor (for IsCapFunctor)

▷ `ExtendFunctorToCochainComplexCategoryFunctor(F)` (operation)

Returns: a functor

The input is a functor $F : A \to B$. The output is its extention functor

$$\mathrm{Ch}^\bullet F : \mathrm{Ch}^\bullet(A) \to \mathrm{Ch}^\bullet(B)$$

.

$$
\begin{tikzcd}
C^\bullet \arrow[d, "\phi"'] & \cdots \arrow[r, "d_{n-2}"] & C_{n-1} \arrow[r, "d_{n-1}"] \arrow[d, "\phi_{n-1}"'] & C_n \arrow[r, "d_n"] \arrow[d, "\phi_n"'] & C_{n+1} \arrow[r] \arrow[d, "\phi_{n+1}"'] & \cdots \\
D^\bullet & \cdots \arrow[r, "d_{n-2}"'] & D_{n-1} \arrow[r, "d_{n-1}"'] & D_n \arrow[r, "d_n"'] & D_{n+1} \arrow[r] & \cdots
\end{tikzcd}
$$

$$
\begin{tikzcd}
\mathrm{Ch}^\bullet F(C^\bullet) \arrow[d, "\mathrm{Ch}^\bullet F(\phi)"'] & \cdots \arrow[r, "F(d_{n-2})"] & F(C_{n-1}) \arrow[r, "F(d_{n-1})"] \arrow[d, "F(\phi_{n-1})"'] & F(C_n) \arrow[r, "F(d_n)"] \arrow[d, "F(\phi_n)"'] & F(C_{n+1}) \arrow[r] \arrow[d, "F(\phi_{n+1})"'] & \cdots \\
\mathrm{Ch}^\bullet F(D^\bullet) & \cdots \arrow[r, "F(d_{n-2})"'] & F(D_{n-1}) \arrow[r, "F(d_{n-1})"'] & F(D_n) \arrow[r, "F(d_n)"'] & F(D_{n+1}) \arrow[r] & \cdots
\end{tikzcd}
$$

4.1.8 BrutalTruncationAboveFunctor (for IsCapCategory, IsInt)

▷ `BrutalTruncationAboveFunctor(Com(A), n)` (operation)

Returns: a endofunctor

The input is a complex category $\mathrm{Com}(A)$ of some Cap category A and an integer n. The output is an endofunctor from $\mathrm{Com}(A) \to \mathrm{Com}(A)$. If $\mathrm{Com}(A) = \mathrm{Ch}_\bullet(A)$ is a chain complex category then the output is the functor

$$\sigma_{<n} : \mathrm{Ch}_\bullet(A) \to \mathrm{Ch}_\bullet(A)$$

$$
\begin{tikzcd}
C_\bullet \arrow[d, "\phi"'] & \cdots & C_{n-1} \arrow[l, "d_{n-1}"] \arrow[d, "\phi_{n-1}"'] & C_n \arrow[l, "d_n"] \arrow[d, "\phi_n"'] & C_{n+1} \arrow[l, "d_{n+1}"] \arrow[d, "\phi_{n+1}"'] & \cdots \arrow[l] \\
D_\bullet & \cdots & D_{n-1} \arrow[l, "d_{n-1}"] & D_n \arrow[l, "d_n"] & D_{n+1} \arrow[l, "d_{n+1}"] & \cdots \arrow[l]
\end{tikzcd}
$$

$$
\begin{tikzcd}
\sigma_{<n}(C_\bullet) \arrow[d, "\sigma_{<n}(\phi)"'] & \cdots & C_{n-1} \arrow[l, "d_{n-1}"] \arrow[d, "\phi_{n-1}"'] & 0 \arrow[l] \arrow[d] & 0 \arrow[l] \arrow[d] & \cdots \arrow[l] \\
\sigma_{<n}(D_\bullet) & \cdots & D_{n-1} \arrow[l, "d_{n-1}"] & 0 \arrow[l] & 0 \arrow[l] & \cdots \arrow[l]
\end{tikzcd}
$$

If $\mathrm{Com}(A) = \mathrm{Ch}^\bullet(A)$ is a cochain complex category then the output is the functor

$$\sigma^{\leq n} : \mathrm{Ch}^\bullet(A) \to \mathrm{Ch}^\bullet(A)$$

$$
\begin{tikzcd}
C^\bullet \arrow[d, "\phi"'] & \cdots \arrow[r, "d_{n-2}"] & C_{n-1} \arrow[r, "d_{n-1}"] \arrow[d, "\phi_{n-1}"'] & C_n \arrow[r, "d_n"] \arrow[d, "\phi_n"'] & C_{n+1} \arrow[r] \arrow[d, "\phi_{n+1}"'] & \cdots \\
D^\bullet & \cdots \arrow[r, "d_{n-2}"'] & D_{n-1} \arrow[r, "d_{n-1}"'] & D_n \arrow[r, "d_n"'] & D_{n+1} \arrow[r] & \cdots
\end{tikzcd}
$$

$$
\begin{tikzcd}
\sigma^{\leq n}(C^\bullet) \arrow[d, "\sigma_{\leq n}(\phi)"'] & \cdots \arrow[r, "d_{n-2}"] & C_{n-1} \arrow[r, "d_{n-1}"] \arrow[d, "\phi_{n-1}"'] & C_n \arrow[r] \arrow[d, "\phi_n"'] & 0 \arrow[r] \arrow[d] & \cdots \\
\sigma^{\leq n}(D^\bullet) & \cdots \arrow[r, "d_{n-2}"'] & D_{n-1} \arrow[r, "d_{n-1}"'] & D_n \arrow[r] & 0 \arrow[r] & \cdots
\end{tikzcd}
$$

4.1.9 BrutalTruncationBelowFunctor (for IsCapCategory, IsInt)

▷ `BrutalTruncationBelowFunctor(Com(A), n)` (operation)

Returns: a endofunctor

The input is a complex category $\mathrm{Com}(A)$ of some Cap category A and an integer n. The output is an endofunctor from $\mathrm{Com}(A) \to \mathrm{Com}(A)$. If $\mathrm{Com}(A) = \mathrm{Ch}_\bullet(A)$ is a chain complex category then the output is the functor

$$\sigma_{\geq n} : \mathrm{Ch}_\bullet(A) \to \mathrm{Ch}_\bullet(A)$$

$$
\begin{tikzcd}
C_\bullet \arrow[d, "\phi"'] & \cdots & C_{n-1} \arrow[l, "d_{n-1}"] \arrow[d, "\phi_{n-1}"'] & C_n \arrow[l, "d_n"] \arrow[d, "\phi_n"'] & C_{n+1} \arrow[l, "d_{n+1}"] \arrow[d, "\phi_{n+1}"'] & \cdots \arrow[l] \\
D_\bullet & \cdots & D_{n-1} \arrow[l, "d_{n-1}"] & D_n \arrow[l, "d_n"] & D_{n+1} \arrow[l, "d_{n+1}"] & \cdots \arrow[l]
\end{tikzcd}
$$

$$
\begin{tikzcd}
\sigma_{\geq n}(C_\bullet) \arrow[d, "\sigma_{\geq n}(\phi)"'] & \cdots & 0 \arrow[l] \arrow[d] & C_n \arrow[l] \arrow[d, "\phi_n"'] & C_{n+1} \arrow[l, "d_{n+1}"'] \arrow[d, "\phi_{n+1}"'] & \cdots \arrow[l] \\
\sigma_{\geq n}(D_\bullet) & \cdots & 0 \arrow[l] & D_n \arrow[l] & D_{n+1} \arrow[l, "d_{n+1}"'] & \cdots \arrow[l]
\end{tikzcd}
$$

If $\mathrm{Com}(A) = \mathrm{Ch}^\bullet(A)$ is a cochain complex category then the output is the functor

$$\sigma^{>n} : \mathrm{Ch}^\bullet(A) \to \mathrm{Ch}^\bullet(A)$$

$$
\begin{tikzcd}
C^\bullet \arrow[d, "\phi"'] & \cdots \arrow[r, "d_{n-2}"] & C_{n-1} \arrow[r, "d_{n-1}"] \arrow[d, "\phi_{n-1}"'] & C_n \arrow[r, "d_n"] \arrow[d, "\phi_n"'] & C_{n+1} \arrow[r] \arrow[d, "\phi_{n+1}"'] & \cdots \\
D^\bullet & \cdots \arrow[r, "d_{n-2}"'] & D_{n-1} \arrow[r, "d_{n-1}"'] & D_n \arrow[r, "d_n"'] & D_{n+1} \arrow[r] & \cdots
\end{tikzcd}
$$

$$\begin{tikzcd}
\sigma^{>n}(C^\bullet) \arrow[d, "\sigma^{>n}(\phi)"'] & \cdots \arrow[r] & 0 \arrow[r] \arrow[d] & 0 \arrow[r] \arrow[d] & C_{n+1} \arrow[r, "d_{n+1}"] \arrow[d, "\phi_{n+1}"'] & \cdots \\
\sigma^{>n}(D^\bullet) & \cdots \arrow[r] & 0 \arrow[r] & 0 \arrow[r] & D_{n+1} \arrow[r, "d_{n+1}"] & \cdots
\end{tikzcd}$$

4.2 Examples

The theory tells us that the composition $i\psi$ is null-homotopic. That implies that the morphisms induced on cohomologies are all zero.

Example

```
(continued)
gap> i_o_psi := PreCompose( psi, i );
<A bounded morphism in cochain complexes category over category of matrices
over Q with active lower bound 4 and active upper bound 6>
gap> H5 := CohomologyFunctorAt( cochain_cat, matrix_category, 5 );
5-th cohomology functor in category of matrices over Q
gap> IsZeroForMorphisms( ApplyFunctor( H5, i_o_psi ) );
true
```

Next we define a functor $\mathbf{F} : \mathrm{Vec}_{\mathbb{Q}} \to \mathrm{Vec}_{\mathbb{Q}}$ that maps every $\mathbb{Q}$-vector space A to $A \oplus A$ and every morphism $f : A \to B$ to $f \oplus f$. Then we extend it to the functor $\mathbf{Coch}_{\mathbf{F}} : \mathrm{Coch}(\mathrm{Vec}_{\mathbb{Q}}) \to \mathrm{Coch}(\mathrm{Vec}_{\mathbb{Q}})$ that maps each cochain complex C to the cochain complex we get after applying the functor $\mathbf{F}$ on every object and differential in C and maps any morphism $\phi : C \to D$ to the morphism we get after applying the functor $\mathbf{F}$ on every object, differential or morphism in C, D and ϕ.

Example

```
(continued)
gap> F := CapFunctor( "double functor", matrix_category, matrix_category );
double functor
gap> u := function( obj ) return DirectSum( [ obj, obj ] ); end;;
gap> AddObjectFunction( F, u );
gap> v := function( s, mor, r ) return DirectSumFunctorial( [ mor, mor ] ); end;;
gap> AddMorphismFunction( F, v );
gap> Display( f );
[ [  1,  3 ] ]

A split monomorphism in Category of matrices over Q
gap> Display( ApplyFunctor( F, f ) );
[ [  1,  3,  0,  0 ],
  [  0,  0,  1,  3 ] ]

A morphism in Category of matrices over Q
gap> Coch_F := ExtendFunctorToCochainComplexCategoryFunctor( F );
Extended version of double functor from cochain complexes category over category
of matrices over Q to cochain complexes category over category of matrices over Q
gap> psi;
<A bounded morphism in cochain complexes category over category of matrices
over Q with active lower bound 4 and active upper bound 6>
```

```
gap> Coch_F_psi := ApplyFunctor( Coch_F, psi );
<A bounded morphism in cochain complexes category over category of matrices
over Q with active lower bound 4 and active upper bound 6>
gap> Display( psi[ 5 ] );
[ [  10 ] ]

A morphism in Category of matrices over Q
gap> Display( Coch_F_psi[ 5 ] );
[ [  10,   0 ],
  [   0,  10 ] ]

A morphism in Category of matrices over Q
```

Next we will compute the shift $C[3]$. As we know the standard shift functor may change the sign of the differentials since $d^i_{C[n]} = (-1)^n d^{i+n}_C$. Hence if we don't want the signs to be changed we may use the unsigned shift functor.

Example

```
(continued)
gap> T := ShiftFunctor( cochain_cat, 3 );
Shift (3 times to the left) functor in cochain complexes category over category
 of matrices over Q
gap> C;
<A not cyclic, bounded object in cochain complexes category over category of
matrices over Q with active lower bound 2 and active upper bound 7>
gap> C_3 := ApplyFunctor( T, C );
<A not cyclic, bounded object in cochain complexes category over category of
matrices over Q with active lower bound -1 and active upper bound 4>
gap> Display( C^3 );
[ [  1,  3 ] ]

A split monomorphism in Category of matrices over Q
gap> Display( C_3^0 );
[ [  -1,  -3 ] ]

A morphism in Category of matrices over Q
gap> S := UnsignedShiftFunctor( cochain_cat, 3 );
Unsigned shift (3 times to the left) functor in cochain complexes category over
category of matrices over Q
gap> C_3_unsigned := ApplyFunctor( S, C );
<A bounded object in cochain complexes category over category of matrices over
Q with active lower bound -1 and active upper bound 4>
gap> Display( C_3_unsigned^0 );
[ [  1,  3 ] ]

A split monomorphism in Category of matrices over Q
```

Let us demonstrate how to use the brutal truncations functors

Example

```
(continued)
gap> cochain_cat;
Cochain complexes category over category of matrices over Q
gap> chain_cat := ChainComplexCategory( matrix_category );
```

```
Chain complexes category over category of matrices over Q
gap> trunc_leq_4 := BrutalTruncationAboveFunctor( cochain_cat, 4 );
Functor of brutal truncation from above (C -> C^<= 4) in Cochain complexes
category over category of matrices over Q
gap> trunc_l_4 := BrutalTruncationAboveFunctor( chain_cat, 4 );
Functor of brutal truncation from above (C -> C_< 4) in Chain complexes
category over category of matrices over Q
gap> trunc_g_4 := BrutalTruncationBelowFunctor( cochain_cat, 4 );
Functor of brutal truncation from below (C -> C^> 4) in Cochain complexes
category over category of matrices over Q
gap> trunc_geq_4 := BrutalTruncationBelowFunctor( chain_cat, 4 );
Functor of brutal truncation from below (C -> C_>= 4) in Chain complexes
category over category of matrices over Q
gap> ApplyFunctor( trunc_leq_4, C );
<A bounded object in cochain complexes category over category of matrices over Q \
with active lower bound 2 and active upper bound 5>
gap> ApplyFunctor( trunc_g_4, C );
<A bounded object in cochain complexes category over category of matrices over Q \
with active lower bound 4 and active upper bound 7>
```

4.2.1 GoodTruncationAboveFunctor (for IsCapCategory, IsInt)

▷ GoodTruncationAboveFunctor(*A*) (operation)

Returns: a functor

To do.

Chapter 5

Double complexes

5.1 Creating double complexes

Let $\mathscr{A}$ be an additive category. A *double chain complex* in $\mathscr{A}$ is given by a system $(\{D_{i,j}, h^D_{i,j}, v^D_{i,j}\}_{i,j\in\mathbf{Z}})$, where each $D_{i,j}$ is an object of $\mathscr{A}$ and $h^D_{i,j} : D_{i,j} \to D_{i-1,j}$ and $v^D_{i,j} : D_{i,j} \to D_{i,j-1}$ are morphisms of $\mathscr{A}$ such that the following rules hold:

1. $h^D_{i-1,j} \circ h^D_{i,j} = 0$
2. $v^D_{i,j-1} \circ v^D_{i,j} = 0$
3. $h^D_{i,j-1} \circ v^D_{i,j} + v^D_{i-1,j} \circ h^D_{i,j} = 0$

for all $i, j \in \mathbf{Z}$. <P/>

$$
\begin{tikzcd}
 & \vdots \arrow[d] & \vdots \arrow[d] & & \\
\cdots & D_{i-1,j} \arrow[l] \arrow[d] & D_{i,j} \arrow[l, "h^D_{i,j}"'] \arrow[d, "v^D_{i,j}"] & \cdots \arrow[l] & row_j \\
\cdots & D_{i-1,j-1} \arrow[l] \arrow[d] & D_{i,j-1} \arrow[l] \arrow[d] & \cdots \arrow[l] & row_{j-1} \\
 & \vdots & \vdots & & \\
 & col_{i-1} & col_i & &
\end{tikzcd}
$$

5.1.1 DoubleChainComplex (for IsCapCategory, IsInfList, IsInfList)

▷ `DoubleChainComplex(`*`A, rows, cols`*`)` (operation)

Returns: a double chain complex

The input is a Cap category $\mathscr{A}$ and two infinite lists *rows* and *cols*. The entry in index j of *rows* should be an infinite list that represents the j'th row of the double complex. I.e., $h^D_{i,j} := rows[j][i]$ for all $i \in \mathbb{Z}$. Again, the entry in index i of *cols* should be an infinite list that represents the i'th column of the double complex. I.e., $v^D_{i,j} := cols[i][j]$.

5.1.2 DoubleChainComplex (for IsCapCategory, IsFunction, IsFunction)

▷ `DoubleChainComplex(A, H, V)` (operation)

Returns: a double chain complex

The input is a Cap category $\mathscr{A}$ and two functions R and V. The output is the double chain complex D defined by $h^D_{i,j} = H(i,j)$ and $v^D_{i,j} = V(i,j)$.

5.1.3 DoubleChainComplex (for IsChainComplex)

▷ `DoubleChainComplex(C)` (operation)

Returns: a double chain complex

The input is chain complex of chain complexes C. The output is the double chain complex D defined using sign trick. I.e., $h^D_{i,j} = (d^C_i)_j$ and $v^D_{i,j} = (-1)^i d^{C_i}_j$.

5.1.4 DoubleChainComplex (for IsDoubleCochainComplex)

▷ `DoubleChainComplex(C)` (operation)

Returns: a double chain complex

The input is double cochain complex D. The output is the double chain complex E defined by $h^E_{i,j} = h_D^{-i,-j}$ and $v^E_{i,j} = v_D^{-i,-j}$.

Let $\mathscr{A}$ be an additive category. A *double cochain complex* in $\mathscr{A}$ is given by a system $(\{D^{i,j}, h_D^{i,j}, v_D^{i,j}\}_{i,j\in\mathbf{Z}})$, where each $D^{i,j}$ is an object of $\mathscr{A}$ and $h_D^{i,j} : D^{i,j} \to D^{i+1,j}$ and $v_D^{i,j} : D^{i,j} \to D^{i,j+1}$ are morphisms of $\mathscr{A}$ such that the following rules hold:

1. $h_D^{i+1,j} \circ h_D^{i,j} = 0$
2. $v_D^{i,j+1} \circ v_D^{i,j} = 0$
3. $h_D^{i,j+1} \circ v_D^{i,j} + v_D^{i+1,j} \circ h_D^{i,j} = 0$

for all $i, j \in \mathbf{Z}$.

$$
\begin{tikzcd}
 & \vdots & \vdots & & \\
\cdots \arrow[r] & D^{i,j+1} \arrow[u] \arrow[r] & D^{i+1,j+1} \arrow[u] \arrow[r] & \cdots & row_{j+1} \\
\cdots \arrow[r] & D^{i,j} \arrow[u, "v_D^{i,j}"] \arrow[r, "h_D^{i,j}"] & D^{i+1,j} \arrow[u] \arrow[r] & \cdots & row_j \\
 & \vdots \arrow[u] & \vdots \arrow[u] & & \\
 & col_i & col_{i+1} & &
\end{tikzcd}
$$

5.1.5 DoubleCochainComplex (for IsCapCategory, IsInfList, IsInfList)

▷ `DoubleCochainComplex(A, rows, cols)` (operation)

Returns: a double cochain complex

The input is a Cap category $\mathscr{A}$ and two infinite lists *rows* and *cols*. The entry in index j of *rows* should be an infinite list that represents the j'th row of the double complex. I.e., $h_D^{i,j} := rows[j][i]$ for all $i \in \mathbb{Z}$. Again, the entry in index i of *cols* should be an infinite list that represents the i'th column of the double complex. I.e., $v_D^{i,j} := cols[i][j]$.

5.1.6 DoubleCochainComplex (for IsCapCategory, IsFunction, IsFunction)

▷ `DoubleCochainComplex(A, H, V)` (operation)

Returns: a double cochain complex

The input is a Cap category $\mathscr{A}$ and two functions R and V. The output is the double chain complex D defined by $h_D^{i,j} = H(i,j)$ and $v_D^{i,j} = V(i,j)$.

5.1.7 DoubleCochainComplex (for IsCochainComplex)

▷ `DoubleCochainComplex(C)` (operation)

Returns: a double cochain complex

The input is cochain complex of cochain complexes C. The output is the double cochain complex D defined using sign trick. I.e., $h_D^{i,j} = (d_C^i)^j$ and $v_D^{i,j} = (-1)^i d_{C^i}^j$.

5.1.8 DoubleCochainComplex (for IsDoubleChainComplex)

▷ `DoubleCochainComplex(C)` (operation)

Returns: a double cochain complex

The input is double chain complex D. The output is the double cochain complex E defined by $h_E^{i,j} = h_{-i,-j}^D$ and $v_E^{i,j} = v_{-i,-j}^D$.

5.2 Attributes and operations

5.2.1 Rows (for IsDoubleChainOrCochainComplex)

▷ `Rows(D)` (attribute)

Returns: an infinite list of infinite lists.

The input is double chain or cochain complex D. The output is the infinite list of rows.

5.2.2 Columns (for IsDoubleChainOrCochainComplex)

▷ `Columns(D)` (attribute)

Returns: an infinite list of infinite lists.

The input is double chain or cochain complex D. The output is the infinite list of columns.

5.2.3 CertainRow (for IsDoubleChainOrCochainComplex, IsInt)

▷ `CertainRow(D, j)` (operation)

Returns: an infinite list

The input is double chain or cochain complex D and integer j. The output is the infinite list that represents the j'th row of D.

5.2.4 CertainColumn (for IsDoubleChainOrCochainComplex, IsInt)

▷ `CertainColumn(D, i)` (operation)

Returns: an infinite list

The input is double chain or cochain complex D and integer i. The output is the infinite list that represents the i'th column of D.

5.2.5 ObjectAt (for IsDoubleChainOrCochainComplex, IsInt, IsInt)

▷ `ObjectAt(D, i, j)` (operation)

Returns: an infinite list

The input is double chain or cochain complex D and integers i, j. The output is the object of D in position (i, j).

5.2.6 HorizontalDifferentialAt (for IsDoubleChainOrCochainComplex, IsInt, IsInt)

▷ `HorizontalDifferentialAt(D, i, j)` (operation)

Returns: a morphism

The input is double chain (resp. cochain) complex D and integers i, j. The output is the horizontal differential $h^D_{i,j}$ (resp. $h_D^{i,j}$)

5.2.7 VerticalDifferentialAt (for IsDoubleChainOrCochainComplex, IsInt, IsInt)

▷ `VerticalDifferentialAt(D, i, j)` (operation)

Returns: a morphism

The input is double chain (resp. cochain) complex D and integers i, j. The output is the vertical differential $v^D_{i,j}$ (resp. $v_D^{i,j}$)

5.2.8 SetAboveBound (for IsDoubleChainOrCochainComplex, IsInt)

▷ `SetAboveBound(D, i)` (operation)
▷ `SetBelowBound(D, i)` (operation)
▷ `SetRightBound(D, i)` (operation)
▷ `SetLeftBound(D, i)` (operation)

Returns: a morphism

Here we can set bounds for the double complex.

5.2.9 TotalChainComplex (for IsDoubleChainComplex)

▷ `TotalChainComplex(D)` (attribute)
▷ `TotalCochainComplex(D)` (attribute)

Returns: a morphism

To be able to compute the total complex the double complex we must have one of the following cases: 1. *D* has left and right bounds. 2. *D* has below and above bounds. 3. *D* has left and below bounds. 4. *D* has right and above bounds.

Chapter 6

Resolutions

6.1 Definitions

Let A be an abelian category and is $C^\bullet$ is a complex in $\mathrm{Ch}^\bullet(A)$. A *projective resolution* of $C^\bullet$ is a complex $P^\bullet$ together with cochain morphism $\alpha : P^\bullet \to C^\bullet$ of complexes such that

- We have $P^n = 0$ for $n \gg 0$, i.e., $P^\bullet$ is bounded above.
- Each P^n is projective object of A.
- The morphism α is quasi-isomorphism.

It turns out that if A is abelian category that has enough projective, then every above bounded cochain complex admits a projective resolution.

6.2 Computing resolutions

6.2.1 ProjectiveResolution (for IsCapCategoryObject)

▷ `ProjectiveResolution(arg)` (attribute)

Returns: a (co)chain complex

If the input is bounded above cochain complex or bounded below chain complex then the output is projective resolution in the sense of the above definition. If the input is an object M which is not a complex and its category has enough projectives, then the output is its projective resolution in the classical sense , i.e., complex $P^\bullet$ which is exact everywhere but in index 0, where $H^0(P^\bullet) \cong M$.

6.2.2 InjectiveResolution (for IsCapCategoryObject)

▷ `InjectiveResolution(arg)` (attribute)

Returns: a (co)chain complex

If the input is bounded above chain complex or bounded below cochain complex then the output is injective resolution in the sense of the above definition. If the input is an object M which is not a complex and its category has enough injectives, then the output is its injective resolution in the classical sense , i.e., complex $I^\bullet$ which is exact everywhere but in index 0, where $H^0(I^\bullet) \cong M$.

6.2.3 QuasiIsomorphismFromProjectiveResolution (for IsBoundedAboveCochainComplex)

▷ `QuasiIsomorphismFromProjectiveResolution(C)` (attribute)
▷ `QuasiIsomorphismFromProjectiveResolution(C)` (attribute)

Returns: a (co)chain epimorphism

The input is an above bounded cochain complex $C^\bullet$. The output is a quasi-isomorphism $q : P^\bullet \to C^\bullet$ such that $P^\bullet$ is upper bounded and all its objects are projective in the underlying abelian category. In the second command the input is a below bounded chain complex $C_\bullet$. The output is a quasi-isomorphism $q : P_\bullet \to C_\bullet$ such that $P_\bullet$ is lower bounded and all its objects are projective in the underlying abelian category.

6.2.4 QuasiIsomorphismInInjectiveResolution (for IsBoundedBelowCochainComplex)

▷ `QuasiIsomorphismInInjectiveResolution(C)` (attribute)
▷ `QuasiIsomorphismInInjectiveResolution(C)` (attribute)

Returns: a (co)chain epimorphism

The input is a below bounded cochain complex $C^\bullet$. The output is a quasi-isomorphism $q : C^\bullet \to I^\bullet$ such that $I^\bullet$ is below bounded and all its objects are injective in the underlying abelian category. In the second command the input is an above bounded chain complex $C_\bullet$. The output is a quasi-isomorphism $q : C_\bullet \to I_\bullet$ such that $I_\bullet$ is below bounded and all its objects are injective in the underlying abelian category.

6.3 Examples

Let

Example
```
gap> LoadPackage( "ModulePresentations" );;
gap> LoadPackage( "ComplexesForCAP" );;
gap> Z6 := HomalgRingOfIntegers( )/6;
Z/( 6 )
gap> cat := LeftPresentations( Z6:FinalizeCategory := false );
Category of left presentations of Z/( 6 )
gap> AddEpimorphismFromSomeProjectiveObject( cat, CoverByFreeModule );
gap> Finalize( cat );
true
gap> SetIsAbelianCategoryWithEnoughProjectives( cat, true );
gap> m := HomalgMatrix( "[ [ 3 ] ]", 1, 1, Z6 );
<A 1 x 1 matrix over a residue class ring>
gap> Z2 := AsLeftPresentation( m );
<An object in Category of left presentations of Z/( 6 )>
gap> proj_Z2 := ProjectiveResolution( Z2 );
<A bounded from above object in cochain complexes category
over category of left presentations of Z/( 6 ) with active upper bound 1>
gap> Display( proj_Z2^-1 );
[ [ 3 ] ]

modulo [ 6 ]
```

```
A morphism in Category of left presentations of Z/( 6 )
gap> Display( proj_Z2^-2 );
[ [ 2 ] ]

modulo [ 6 ]

A morphism in Category of left presentations of Z/( 6 )
gap> Display( proj_Z2^-300 );
[ [ 2 ] ]

modulo [ 6 ]

A morphism in Category of left presentations of Z/( 6 )
gap> Display( proj_Z2^-301 );
[ [ 3 ] ]

modulo [ 6 ]

A morphism in Category of left presentations of Z/( 6 )
```

Index