SingerAlg — A GAP 4 Package

(Version 1.0.1)

Thomas Breuer

Thomas Breuer Email: sam@Math.RWTH-Aachen.De
Homepage: http://www.math.rwth-aachen.de/~Thomas.Breuer

Copyright

Contents

Chapter 1

Introduction to the SingerAlg Package

1.1 Aims of the SingerAlg Package

The *mathematical aims* of the package are

- to document the computational results used in the papers [BHHK20] and [BHHK21],
- to give access to GAP and Julia [BEKS17] implementations of the relevant functions (see Chapter 3), and
- to give access to the known data about the Singer algebras $A[q,z]$ with $1 \leq z \leq 10000$ (see Section 1.2 for the background and Chapter 4 for the database).

From the viewpoint of *programming*, the package provides examples how the GAP/Julia integration via the JuliaInterface package [BGH20] can be used (see Section 1.3). From the viewpoint of *teaching*, some aspects of this package and of the papers mentioned above would be suitable for a course in elementary number theory, with accompanying computational experiments. Note that one can neglect the representation theoretic background, and restrict the needed theory to knowledge of the group of prime residues modulo an integer, g.c.d. computations, and linear algebra; for examples, see Chapter 2, in particular Section 2.2, and Chapter 4.

1.2 Theoretical Background: Singer algebras

Let z be a positive integer, and $q \in \{2,3,\ldots,z+1\}$ be coprime to z. Let $n = \mathrm{ord}_z(q)$ denote the multiplicative order of q modulo z, and let $e = (q^n-1)/z$. We define the *Singer algebra* $A[q,z]$ as the free $\mathbb{Z}$-module spanned by $(b_0, b_1, \ldots, b_z)$ where the multiplication is given by $b_i \cdot b_j = b_{i+j}$ if the pointwise sum of the q-adic expansions of ie and je does not exceed $q-1$ (that is, there is no carry in the q-adic addition of ie and je), and $b_i \cdot b_j = 0$ otherwise.

Another interpretation of $A[q,z]$ and the basis of b_i is as follows (see [BHHK21, Section 1]). Consider the factor of the polynomial ring $R = \mathbb{Z}[X_1, X_2, \ldots, X_n]$ modulo the ideal I spanned by $X_1^q, X_2^q, \ldots, X_n^q$. Then $A[q,z]$ is isomorphic to a subalgebra of this algebra, via mapping b_k to the monomial $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$, where $i_1 + qi_2 + \cdots + q^{n-1} i_n = ke$ and $x_i = X_i + I$. This interpretation motivates the terms "monomials" for the b_i and "degree" for the sum $i_1 + i_2 + \cdots i_n$; these terms occur in the names and descriptions of several functions of this package.

The algebra $A[q,z]$ is called $A(q,n,e)$ and $A[q,n,z]$ in [BHHK20] and [BHHK21], where n is required to be a multiple of $\mathrm{ord}_e(q)$ or $\mathrm{ord}_z(q)$, respectively, and q can be any integer larger than 1

that is coprime to e or z, respectively. For the purposes of this GAP package, it is more suitable to consider the parameters q and z, as introduced above. In particular, these numbers are small for low dimensional algebras, whereas the number e can be quite large, see Section 2.2.1.

Note that $A[q,n,z]$ is isomorphic to $A[q,\mathrm{ord}_z(q),z]$ for each multiple n of $\mathrm{ord}_z(q)$, by [BHHK21, Lemma 7.1], and $A[q,z]$ is isomorphic to $A[q',z]$ whenever q and q' generate the same subgroup of prime residues modulo z, by [BHHK21, Lemma 7.5].

For any prime integer p, $A[q,z]_p = \bigoplus_{0\le i\le z} \mathbb{F}_p b_i$ denotes the reduction of $A[q,z]$ modulo p. We call $(b_0,b_1,\ldots,b_z)$ the *canonical basis* of $A[q,z]$ and $A[q,z]_p$, and denote it by $B(A[q,z])$ and $B(A[q,z]_p)$, respectively. Note that the i-th basis vector is $B(A[q,z])_i = b_{i-1}$.

The name "Singer algebra" was chosen because these algebras occur in the following context.

Let p be a prime, n be a positive integer, F be the field with p^n elements, and F_1 be the prime field of F.

We choose an element σ of order p^n-1 in the group $GL(n,F_1)$, a so-called *Singer cycle*, let e be a divisor of p^n-1, and set $z=(p^n-1)/e$.

The cyclic group $H=\langle\sigma^z\rangle$ of order e acts naturally on the elementary abelian group $P=\langle x_1,x_2,\ldots,x_n\rangle$ of order p^n. This action extends to the group algebra FP, and we may consider the algebra $(FP)^H$ of fixed points in FP under this action. As is described in [BHHK21, Section 2], we can construct an F-basis of FP that consists of eigenvectors for the action of H. This yields a basis of $(FP)^H$, for which the multiplication rules stated above for $A[q,z]$ can be derived. In other words, $(FP)^H$ is isomorphic with $F\otimes_{F_1} A[q,z]_p$.

1.3 GAP-Julia Integration in the SingerAlg Package

If the SingerAlg package is used together with the GAP package JuliaInterface [BGH20] then both GAP and Julia implementations of most of the package's functions are available. For example, there are the functions `LoewyLengthGAP` (3.1.6) and `LoewyLengthJulia` (3.1.6) for computing the Loewy length of a Singer algebra from its defining parameters. There is also the function `LoewyLength` (3.1.6), which uses one of the two implementations. By default, the Julia code is used for computations if JuliaInterface is available, and the GAP code is used by default if not.

Note that some of the Julia functions call GAP functions. For example, the factorization of large integers in GAP and its package FactInt [Koh19] are (currently) faster than the corresponding functions from Julia's `Primes` package. In order to achieve a "fair" comparison of the runtimes in GAP and Julia, we call the GAP function `Factors` (**Reference: Factors**) in both situations.

Note also that Julia objects are stored as the values of GAP attributes such as `LoewyStructureInfoJulia` (3.1.11), and some Julia functions take a GAP object as an argument, in order to benefit from its attribute values if applicable.

One can also use the Julia functions in a Julia session, by loading the file `julia/SingerAlg.jl` into Julia with `include`; afterwards the Julia functions are accessible in the Julia module `"SingerAlg"`. This approach requires the Julia package `GAP.jl` (and thus a Julia compatible installation of GAP).

1.4 Installation of the SingerAlg Package

The SingerAlg package itself consists only of GAP (and Julia) code, it is in principle enough to unpack the archive in a `pkg` directory of a GAP installation, and then to load the package into the GAP session.

However, if one wants to use the Julia implementations of the package's functions then the GAP package JuliaInterface [BGH20] must be available. In particular, the GAP installation must be done using Julia's garbage collector in this case; one can check this by looking at the value of `GAPInfo.KernelInfo.GC`: If it is `"Julia GC"` then JuliaInterface can be loaded (provided it is installed), otherwise Julia features are not available.

The easiest way to install GAP with Julia's garbage collector is to install Julia (see `https://julialang.org/downloads/`), then ask Julia's package manager to download and install GAP (by entering `using Pkg; Pkg.add( "GAP" )` at the Julia prompt), and then to use the SingerAlg package with this version of GAP.

For one function of the package (`SingerAlg.ProposedPermutationIsomorphism` (3.4.2)), the interface to [McK90] provided by GAP's GraPe package [Soi19] is needed; if one wants to use this function –see for example Section 4.3.4– then GraPe must be installed.

1.5 Acknowledgements

The development of this GAP package has been supported by the SFB-TRR 195 "Symbolic Tools in Mathematics and their Applications" (from 2017 until 2020). Thanks to the coauthors of the papers [BHHK20] and [BHHK21], Erzsébet Horváth, László Héthelyi, and Burkhard Külshammer, for many discussions and suggestions that contributed to this package. Thanks to Bettina Eick for hints about invariants, see Section 3.5.

Chapter 2

Tutorial for the SingerAlg package

This chapter shows small introductory computations with the functions of the package. More examples can be found in Section 4.3.

In order to force that the examples in this manual consist only of ASCII characters, we set the user preference `DisplayFunction` of the package (see Section 3.2.3) to the value `"Print"`. This is necessary because the LaTeX and HTML versions of GAPDoc documents do not support non-ASCII characters.

Example

```
gap> origpref:= UserPreference( "SingerAlg", "DisplayFunction" );;
gap> SetUserPreference( "SingerAlg", "DisplayFunction", "Print" );
```

2.1 How to Study Singer Algebras

The definition of Singer algebras given in Section 1.2 suggests that GAP's tools for algebras defined by structure constants (see Section (**Reference: Constructing Algebras by Structure Constants**)) might be suitable for them, and Section 3.1 takes this approach.

However, the fact that the canonical bases of Singer algebras have a very special structure –the product of two basis elements is either another basis element or zero– implies that many interesting subalgebras and subspaces can be described in terms of subsets of the canonical basis. Thus many questions can be answered using combinatorial computations, without the need to add or multiply or even create elements of the algebra. Section 3.3 lists functions where this approach is taken.

Most of the functions of the SingerAlg package do not involve objects that represent algebraic structures. In particular, the Julia code does not introduce such objects.

2.2 Number Theoretic Caveats

When one deals with Singer algebras $A[q,z] = A(q,n,e)$, with $q^n - 1 = ez$, seemingly trivial computations can become expensive even if the dimensions (equal to $z+1$) and the parameters q and n are small, because the number e can still be huge; see Section 2.2.1 for examples. The point is that there are situations where one can (and then should) avoid dealing with large numbers, but there are also situations where this is not possible. Since GAP knows just one type of integers, there is no need to write different GAP code for computations with small or large integers. This is different in Julia, where one can (and wants to) write special code for computing with small integers whenever one knows in advance that there will be no overflow.

Other computational problems arise from factorization questions. One instance is the computation of the multiplicative order of q modulo z or e; an example where calling `OrderMod( q, e )` runs into problems is shown in the documentation of `OrderModExt` (3.6.2). Here the point is that one should enter a known multiple of the desired multiplicative order as the third argument of `OrderModExt` (3.6.2) whenever this is possible.

Also primality tests in the context of natural questions about Singer algebras may run into problems, see Section 2.2.2.

2.2.1 Large values of e

Computational examples in the study of $A(q,n,e)$ often avoid dealing with e because this number can be very large when the algebra itself has small dimension. Let us look at the database of $A[q,z]$ with $z \leq 10000$.

Example

```
gap> cand:= AllSingerAlgebraInfos( "e", e -> e > 2^64 );;
gap> Length( cand );
543989
gap> cand:= AllSingerAlgebraInfos( "e", e -> e > 10^10000 );;
gap> Length( cand );
12
gap> SortParallel( List( cand, r -> r.e ), cand );
gap> cand[ Length( cand ) ];
rec( LL := 3, d := [ 1, 9438 ], dec := 0, diff := 0,
  e := <integer 646...617 (12666 digits)>, isom := [ 9439, 2 ],
  m := 99099, n := 9438, q := 22, vprime := [ 1, 9438, 1 ], z := 9439
 )
```

We see that for the majority of entries in the database, the value of e cannot be represented by a 64-bit integer, and that the largest value of e in the database is bigger than 10^{12666}.

When one deals with $A[q,z]$, one of the basic tasks is to compute the q-adic coefficients of some multiple of e, i. e., to write $ke = \sum_{i=0}^{n} a_i q^i$, with $0 \leq a_i < q$. By [BHHK21, Remark 2.23 (iv)], one can compute the a_i without dealing explicitly with numbers of the magnitude of e.

Example

```
gap> q:= 22;;  n:= 9438;;  z:= 9439;;
gap> e:= (q^n-1)/z;;
gap> coeffs1:= CoefficientsQadic( e, q );;
gap> coeffs2:= CoefficientsQadicReversed( 1, z, q, n );;
gap> Length( coeffs1 );
9436
gap> Length( coeffs2 );
9438
gap> coeffs1 = Reversed( coeffs2 ){[1..9436]};
true
```

Note that `CoefficientsQadicReversed` (3.1.12) is several times faster than `CoefficientsQadic` (**Reference: CoefficientsQadic**); the two functions in question are interpreted GAP functions. The Julia variant `Julia.SingerAlg.CoefficientsQadicReversed` is faster than the GAP function `CoefficientsQadicReversed` (3.1.12), whereas the Julia variant `Julia.SingerAlg.CoefficientsQadic` needs about the same time as `CoefficientsQadic`

(**Reference: CoefficientsQadic**). Note that computations with small integers are much faster in Julia than in GAP, but that the Julia data type of big integers is not supported well.

2.2.2 Primality tests

Suppose we want to check whether [BHHK21, Thm. 4.3 (i)] yields that the Loewy length of a given Singer algebra $A(q,n,e)$ is equal to the upper bound $\lfloor n(q-1)/m(q,e) \rfloor + 1$; for that, we have to decide whether $e/2$ is a prime power.

In the case of $(q,z) = (8390,21)$, GAP prints a message and then nothing happens for a long time.

Example

```
ap> z:= 8390;
8390
gap> r:= OneSingerAlgebraInfo( "z", z, "q", 21 );;
gap> LogInt( r.e, 10 );
550
gap> IsEvenInt( r.e );
true
gap> IsPrimePowerInt( r.e/2 );
#I  Straightforward Fermat-Lucas primality proof failed on 6096...
[...]
```

(Fortunately, we need not check whether $e/2$ is a prime power; if we look into the proof of [BHHK21, Thm. 4.3 (i)], we find out that a cheaper criterion can be used.)

2.3 Some Examples from the Papers

We show the examples of Singer algebras $A(q,n,e)$ that appear in [BHHK20]. See Section 1.2 for the meaning of the term "monomial", and Section `DisplaySingerMonomials` (3.2.1) for the meaning of the output that is shown.

$(q,n,e) = (3,4,5)$, [BHHK20, Example 3.2]:

Example

```
gap> DisplaySingerMonomials( 3, 4, 5 );
A[3,4,16]

0 | 1 |  0
1 | 6 |  1  2  3  6  9 11
2 | 7 |  4  5  7  8 12 13 15
3 | 2 | 10 14
4 | 1 | 16
gap> DisplaySingerMonomials( 3, 4, 5 : m );
A[3,4,16]

0 | 1 | (0,0,0,0)
1 | 6 | (0,0,2,1) (0,1,0,1) (0,2,1,0) (1,0,0,2) (1,0,1,0) (2,1,0,0)
2 | 7 | (0,1,2,2) (0,2,0,2) (1,1,1,1) (1,2,2,0) (2,0,1,2) (2,0,2,0)
  |   | (2,2,0,1)
3 | 2 | (1,2,1,2) (2,1,2,1)
4 | 1 | (2,2,2,2)
```

$(q,n,e) = (13,2,8)$, [BHHK20, Example 3.3]:

Example

```
gap> DisplaySingerMonomials( 13, 2, 8 );
A[13,2,21]

0 | 1 |  0
1 | 4 |  1  2  5 13
2 | 5 |  3  4  7 10 18
3 | 4 |  6  9 12 15
4 | 5 |  8 11 14 17 20
5 | 2 | 16 19
6 | 1 | 21
gap> DisplaySingerMonomials( 13, 2, 8 : m );
A[13,2,21]

0 | 1 | ( 0, 0)
1 | 4 | ( 0, 8) ( 1, 3) ( 3, 1) ( 8, 0)
2 | 5 | ( 1,11) ( 2, 6) ( 4, 4) ( 6, 2) (11, 1)
3 | 4 | ( 3, 9) ( 5, 7) ( 7, 5) ( 9, 3)
4 | 5 | ( 4,12) ( 6,10) ( 8, 8) (10, 6) (12, 4)
5 | 2 | ( 9,11) (11, 9)
6 | 1 | (12,12)
```

$(q,n,e) = (5,4,78)$, [BHHK20, Example 5.1]:

Example

```
gap> DisplaySingerMonomials( 5, 4, 78 );
A[5,4,8]

0 | 1 | 0
1 | 3 | 1 2 5
2 | 3 | 3 4 7
3 | 1 | 6
4 | 1 | 8
gap> DisplaySingerMonomials( 5, 4, 78 : m );
A[5,4,8]

0 | 1 | (0,0,0,0)
1 | 3 | (0,3,0,3) (1,1,1,1) (3,0,3,0)
2 | 3 | (1,4,1,4) (2,2,2,2) (4,1,4,1)
3 | 1 | (3,3,3,3)
4 | 1 | (4,4,4,4)
```

$(q,n,e) = (11,2,4)$, [BHHK20, Example 5.2]:

Example

```
gap> DisplaySingerMonomials( 11, 2, 4 );
A[11,2,30]

 0 | 1 |  0
 1 | 3 |  1  3 11
 2 | 5 |  2  4  6 14 22
 3 | 5 |  5  7  9 17 25
 4 | 5 |  8 10 12 20 28
 5 | 3 | 13 15 23
```

```
 6 | 3 | 16 18 26
 7 | 3 | 19 21 29
 8 | 1 | 24
 9 | 1 | 27
10 | 1 | 30
gap> DisplaySingerMonomials( 11, 2, 4 : m );
A[11,2,30]

 0 | 1 | ( 0, 0)
 1 | 3 | ( 0, 4) ( 1, 1) ( 4, 0)
 2 | 5 | ( 0, 8) ( 1, 5) ( 2, 2) ( 5, 1) ( 8, 0)
 3 | 5 | ( 1, 9) ( 2, 6) ( 3, 3) ( 6, 2) ( 9, 1)
 4 | 5 | ( 2,10) ( 3, 7) ( 4, 4) ( 7, 3) (10, 2)
 5 | 3 | ( 4, 8) ( 5, 5) ( 8, 4)
 6 | 3 | ( 5, 9) ( 6, 6) ( 9, 5)
 7 | 3 | ( 6,10) ( 7, 7) (10, 6)
 8 | 1 | ( 8, 8)
 9 | 1 | ( 9, 9)
10 | 1 | (10,10)
```

2.4 Example: The case $n = 4$

For fixed (small) n, we are interested in the question for which values of q and z the upper bound $\lfloor n(q-1)/m(q,e) \rfloor + 1$ on the Loewy length of $A[q,n,z]$ is not attained.

If $n \leq 3$ holds then we know by [BHHK20, Cor. 7.1] that the upper bound is always attained. For $n = 5$, the database of Singer algebras contains a few examples where the bound is not attained (cf. [BHHK21, Remark 7.13]).

Example

```
gap> expls:= AllSingerAlgebraInfos( "n", 5,
>                r -> r.LL = Int( r.n * ( r.q-1 ) / r.m ) + 1, false );;
gap> Length( expls );
13
gap> expls[1];
rec( LL := 7, d := [ 43, 408 ], dec := 0, diff := 1, e := 407592814,
  isom := [ 1353, 223 ], m := 148, n := 5, q := 223,
  vprime := [ 1, 261, 375, 395, 260, 61, 1 ], z := 1353 )
```

From now on, we fix $n = 4$. The upper bound is attained for all relevant entries in the database.

Example

```
gap> OneSingerAlgebraInfo( "n", 4,
>        r -> r.LL = Int( r.n * ( r.q-1 ) / r.m ) + 1, false );
fail
```

We comute the Loewy lengths of the algebras $A[q,4,z]$, for $2 \leq q \leq 40$ and for all divisors z of $q^4 - 1$, and compare them with the upper bound.

Example

```
gap> n:= 4;;
gap> for q in [ 2 .. 40 ] do
>      for z in DivisorsInt( q^n-1 ) do
```

```
>        if z > SingerAlg.MaxZ then
>          A:= SingerAlgebra( q, n, z );
>          m:= MinimalDegreeOfSingerAlgebra( A );
>          if LoewyLength( A ) <> Int( n * (q-1) / m ) + 1 then
>            Print( "found an example\n" );
>          fi;
>        fi;
>      od;
>    od;
```

The above computations should need less than a minute, provided that the Julia code can be used; much more time is needed if only GAP can be used. [BHHK21, Section 1] states that the bound is always attained for $q \leq 100$; the computations for that need several hours (using Julia).

In some of the examples, such as for $(q,e) = (29,48)$, the computation of $m(q,e)$ without calling `LoewyStructureInfoJulia` (3.1.11) is more expensive than calling this function directly and then reading off the Loewy length (and $m(q,e)$).

Chapter 3

Functions for Singer algebras

3.1 Singer Algebras as Algebraic Structures

3.1.1 SingerAlgebra

▷ `SingerAlgebra(q[, n], z[, R])` (function)
▷ `SingerAlgebra(arec[, R])` (function)

For nonnegative integers `q`, `z` with $q > 1$, and a field `R` (with default the field of Rationals, see `Rationals` (**Reference: Rationals**)), let n be the multiplicative order of `q` modulo `z`, set $e = (q^n - 1)/z$, and define $A[q,z]$ as the free `R`-module with basis $(b_0, b_1, \ldots, b_z)$ and multiplication defined as follows. If there is no carry in the addition of the `q`-adic expansions of ie and je then $b_i b_j = b_{i+j}$ holds, and otherwise $b_i b_j$ is zero.

This function returns the algebra $A[q,z]$.

Alternatively, a record `arec` can be given, which must have the components `q` and `z` or the components `q`, `n`, `e`.

The idea is to use the algebra object first of all as a container for the attributes that belong to the parameters `q` and `z`, see `LoewyLength` (3.1.6), `MinimalDegreeOfSingerAlgebra` (3.1.8), and `LoewyStructureInfo` (3.1.11). This works well also for high dimensional algebras.

If one really wants to do computations beyond this context, for example compute with elements of the algebra, then special methods for `CanonicalBasis` (**Reference: CanonicalBasis**), `Representative` (**Reference: Representative**), `GeneratorsOfAlgebra` (**Reference: GeneratorsOfAlgebra**), or `GeneratorsOfAlgebraWithOne` (**Reference: GeneratorsOfAlgebraWithOne**) will trigger the computation of a structure constants table, and afterwards the algebra behaves like other algebras in GAP that are defined via structure constants.

Example

```
gap> A:= SingerAlgebra( 6, 2, 7 );
A[6,2,7]
gap> Dimension( A );   # is always z+1
8
gap> LeftActingDomain( A );
Rationals
gap> A2:= SingerAlgebra( 6, 2, 7, GF(2) );
A[6,2,7,GF(2)]
gap> Print( A2, "\n" );
SingerAlgebra( 6, 2, 7, GF(2) )
```

```
gap> String( A2 );
"SingerAlgebra( 6, 2, 7, GF(2) )"
gap> SingerAlgebra( rec( q:= 6, n:= 2, e:= 5 ) );
A[6,2,7]
```

3.1.2 IsSingerAlgebra

▷ `IsSingerAlgebra(A)` (Category)

This filter is set in all algebras constructed with `SingerAlgebra` (3.1.1).

Example

```
gap> A:= SingerAlgebra( 6, 7 );
A[6,2,7]
gap> IsSingerAlgebra( A );
true
gap> AA:= AlgebraByStructureConstants( Rationals,
>             StructureConstantsTable( Basis( A ) ) );
<algebra of dimension 8 over Rationals>
gap> IsSingerAlgebra( AA );
false
```

3.1.3 ParametersOfSingerAlgebra

▷ `ParametersOfSingerAlgebra(A)` (attribute)

For a Singer algebra $A = A[q,n,z]$ (see `SingerAlgebra` (3.1.1)), the value is the list $[q,n,z]$.

Example

```
gap> A:= SingerAlgebra( 6, 7 );
A[6,2,7]
gap> ParametersOfSingerAlgebra( A );
[ 6, 2, 7 ]
```

3.1.4 DimensionsLoewyFactors

▷ `DimensionsLoewyFactors(A)` (attribute)
▷ `LoewyVector(A)` (attribute)

For a Singer algebra A (see `SingerAlgebra` (3.1.1)), this function returns the Loewy vector of A, that is, the list of nonzero dimensions J^{i-1}/J^i, for $i \geq 1$, where J is the Jacobson radical of A and $J^0 = A$.

`LoewyVector` is a synonym of `DimensionsLoewyFactors`.

Example

```
gap> A:= SingerAlgebra( 6, 2, 7 );
A[6,2,7]
gap> DimensionsLoewyFactors( A );
[ 1, 6, 1 ]
```

In the GAP Reference Manual, this attribute is declared for groups, see DimensionsLoewyFactors (**Reference: DimensionsLoewyFactors**). In that context, it means the dimensions of the Loewy factors of the group algebra of its argument over the field with p elements, where the argument is required to be a finite p-group. Note that this value can be computed just from the group, without constructing a group algebra.

3.1.5 LoewyVectorAbbreviated and LoewyVectorExpanded

▷ LoewyVectorAbbreviated(*v*) (function)
▷ LoewyVectorExpanded(*v*) (function)

For a dense list *v* of positive integers, LoewyVectorAbbreviated returns a new list in which each maximal sublist of at least two consecutive equal entries is replaced by the list containing this element and its multiplicity.

For a dense list *v* whose entries are non-lists and pairs whose second entries are positive integers, LoewyVectorExpanded returns a new list in which each such pair is replaced by a sublist that contains the first entry with multiplicity given by the second entry.

Example

```
gap> LoewyVectorAbbreviated( [ 1, 1, 1, 1 ] );
[ [ 1, 4 ] ]
gap> LoewyVectorAbbreviated( [ 1, 7, 7, 3, 3, 1, 1, 1 ] );
[ 1, [ 7, 2 ], [ 3, 2 ], [ 1, 3 ] ]
gap> LoewyVectorExpanded( [ [ 1, 4 ] ] );
[ 1, 1, 1, 1 ]
gap> LoewyVectorExpanded( [ 1, [ 7, 2 ], [ 3, 2 ], [ 1, 3 ] ] );
[ 1, 7, 7, 3, 3, 1, 1, 1 ]
```

3.1.6 LoewyLength

▷ LoewyLength(*A*) (attribute)
▷ LoewyLength(*q[, n], z[, m]*) (operation)
▷ LoewyLengthGAP(*A*) (attribute)
▷ LoewyLengthGAP(*q[, n], z[, m]*) (operation)
▷ LoewyLengthJulia(*A*) (attribute)
▷ LoewyLengthJulia(*q[, n], z[, m]*) (operation)

Let *q*, *n*, *z* be positive integers such that *z* divides *q*^*n* - 1; the default for *n* is the multiplicative order of *q* modulo *z*. These functions return the Loewy length of the Singer algebra $A[q,n,z]$, see SingerAlgebra (3.1.1).

Alternatively, also a Singer algebra *A* can be given as an argument.

Note that it may be cheap to compute the Loewy length of this algebra, using criteria from [BHHK20] and [BHHK21], even if computing its Loewy vector (see DimensionsLoewyFactors (3.1.4)) would be hard.

If Julia is available then LoewyLength uses LoewyLengthJulia, otherwise it uses LoewyLengthGAP.

Example

```
gap> A:= SingerAlgebra( 6, 2, 7 );
A[6,2,7]
```

```
gap> LoewyLength( A );
3
```

3.1.7 SufficientCriterionForLoewyBoundAttained

▷ SufficientCriterionForLoewyBoundAttained(*q, n, z, m*) (function)

Returns: a string.

Let *q*, *n*, *z*, *m* be positive integers, where *q* and *z* are coprime. SufficientCriterionForLoewyBoundAttained returns a string that describes the criterion from [BHHK20] or [BHHK21] from which it follows that the Loewy length of the algebra $A[q,z]$ is equal to the upper bound $\lfloor n(q-1)/m \rfloor + 1$, where n=OrderMod(*q*, *z*), $e = (q^n - 1)/z$, and $m = m(q,e)$; if no such criterion was found then the returned string is empty.

Example

```
gap> expls:= [ [3,2], [2,5], [3,70], [13,70], [19,70], [5,72],
>              [2,73], [5,76], [11,80], [13,80], [2,85], [4,123],
>              [3,164], [4,369], [2,771], [15,791] ];;
gap> for l in expls do
>      q:= l[1];  z:= l[2];  n:= OrderMod( q, z );  e:= (q^n-1)/z;
>      nn:= OrderModExt( q, e, n );
>      m:= MinimalDegreeOfSingerAlgebra( q, nn, e );
>      Print( "q = ", String( q, 2 ), ", z = ", String( z, 3 ), ":  ",
>      SufficientCriterionForLoewyBoundAttained( q, n, z, m ), "\n" );
>    od;
q =  3, z =   2:  Cor. I.7.1 (n <= 3)
q =  2, z =   5:  z < 70
q =  3, z =  70:
q = 13, z =  70:  Thm. I.7.1
q = 19, z =  70:  La. I.7.1 (iii)
q =  5, z =  72:  La. II.4.1 for r = 1
q =  2, z =  73:  Prop. II.6.1 (e <= 32)
q =  5, z =  76:  Prop. II.5.1 (ii)
q = 11, z =  80:  Prop. II.5.3, II.5.6 (e | (q^n-1)/(q-1), n <= 5)
q = 13, z =  80:  Prop. II.3.15
q =  2, z =  85:  La. I.6.3
q =  4, z = 123:  Prop. II.5.1 (iii)
q =  3, z = 164:  La. II.5.2 (ii)
q =  4, z = 369:  Prop. II.5.1 (i)
q =  2, z = 771:  La. II.4.1
q = 15, z = 791:  Thm. II.4.3 (iii)
```

3.1.8 MinimalDegreeOfSingerAlgebra

▷ MinimalDegreeOfSingerAlgebra(*A*) (attribute)
▷ MinimalDegreeOfSingerAlgebra(*q, e*) (operation)
▷ MinimalDegreeOfSingerAlgebraGAP(*A*) (attribute)
▷ MinimalDegreeOfSingerAlgebraGAP(*q, e*) (operation)
▷ MinimalDegreeOfSingerAlgebraJulia(*A*) (attribute)
▷ MinimalDegreeOfSingerAlgebraJulia(*q, e*) (operation)

Returns: a positive integer.

For two coprime positive integers `q` and `e`, `MinimalDegreeOfSingerAlgebra` computes the minimal number of powers of `q` such that `e` divides the sum of these powers.

If a Singer algebra $A[q,z]$ is given as the argument `A` (see `SingerAlgebra` (3.1.1)) then the value for the parameters q and $e = (q^n - 1)/z$ is returned, where n is the multiplicate order of q modulo z, see `OrderMod` (**Reference: OrderMod**); note that the minimal degree does not depend on n.

The same value is returned by `MinimalDegreeOfSingerAlgebraGAP` and `MinimalDegreeOfSingerAlgebraJulia`, which use implementations in GAP and Julia, respectively. `MinimalDegreeOfSingerAlgebra` delegates to the Julia variant if the package JuliaInterface is available, and to the GAP variant otherwise.

Example

```
gap> A:= SingerAlgebra( 6, 2, 7 );
A[6,2,7]
gap> MinimalDegreeOfSingerAlgebra( A );
5
gap> MinimalDegreeOfSingerAlgebra( 6, 5 );
5
```

3.1.9 RadicalSeriesOfAlgebra

▷ `RadicalSeriesOfAlgebra(A)` (attribute)

For an algebra `A` with radical `J` (see `RadicalOfAlgebra` (**Reference: RadicalOfAlgebra**)), `RadicalSeriesOfAlgebra` returns the list $[J^0, J^1, J^2, J^3, \ldots, J^k]$, where k is the smallest index such that J^k is zero.

Example

```
gap> A:= SingerAlgebra( 2, 7 );
A[2,3,7]
gap> ser:= RadicalSeriesOfAlgebra( A );
[ A[2,3,7], <algebra of dimension 7 over Rationals>,
  <algebra of dimension 4 over Rationals>,
  <algebra of dimension 1 over Rationals>,
  <algebra of dimension 0 over Rationals> ]
```

3.1.10 SocleSeriesOfAlgebra

▷ `SocleSeriesOfAlgebra(A)` (attribute)

For an algebra `A`, `SocleSeriesOfAlgebra` returns the list $[S_0, S_1, S_2, S_3, \ldots, S_k]$, where S_0 is the trivial subalgebra of `A` and S_{i+1}/S_i is the socle of A/S_i and k is the smallest index such that $S_k = A$ holds. (Thus `S_1` is the socle of `A`.)

Example

```
gap> A:= SingerAlgebra( 2, 7 );
A[2,3,7]
gap> ser:= SocleSeriesOfAlgebra( A );
[ <algebra of dimension 0 over Rationals>,
  <algebra of dimension 1 over Rationals>,
  <algebra of dimension 4 over Rationals>,
  <algebra of dimension 7 over Rationals>, A[2,3,7] ]
```

3.1.11 LoewyStructureInfo

▷ LoewyStructureInfo(A) (attribute)
▷ LoewyStructureInfoGAP(A) (attribute)
▷ LoewyStructureInfoJulia(A) (attribute)
▷ LoewyStructureInfo(q[, n], z) (operation)
▷ LoewyStructureInfoGAP(q[, n], z) (operation)
▷ LoewyStructureInfoJulia(q[, n], z) (operation)

For a Singer algebra A (see SingerAlgebra (3.1.1)) with parameters q, n, z, or for these parameters themselves, these three operations compute the distribution of the canonical basis in $A[q,z]$ to Loewy layers, using the algorithm from [BHHK20, Proposition 3.2].

Let $e = (q^n - 1)/z$.

LoewyStructureInfoJulia returns a Julia dictionary whose keys are the following symbols.

:monomials
: the array of reversed (see CoefficientsQadicReversed (3.1.12)) q-adic expansions for multiples of e, each of length n,

:layers
: the array of the Loewy layers to which the monomials belong,

:chain
: an array of positions of monomials of a longest ascending chain,

:m
: the value $m(q,e)$ (see MinimalDegreeOfSingerAlgebra (3.1.8)

:LL
: the Loewy length of A (see LoewyLength (3.1.6)), equal to the length of the :layers value plus 1,

:parameters
: the array [q, n, z] of parameters of A.

LoewyStructureInfoGAP returns a GAP record whose components correspond to the keys listed above. LoewyStructureInfo returns the same; however, if Julia is available then this result is computed by converting the result of LoewyStructureInfoJulia to a GAP object.

Example

```
gap> LoewyStructureInfo( 6, 7 );
rec( LL := 3, chain := [ 8, 2, 1 ],
  layers := [ 0, 1, 1, 1, 1, 1, 1, 2 ], m := 5,
  monomials := [ [ 0, 0 ], [ 0, 5 ], [ 1, 4 ], [ 2, 3 ], [ 3, 2 ],
      [ 4, 1 ], [ 5, 0 ], [ 5, 5 ] ], parameters := [ 6, 2, 7 ] )
```

3.1.12 CoefficientsQadicReversed

▷ CoefficientsQadicReversed(k, z, q, n) (function)

Let k, z, q, n be positive integers such that $0 \le k \le z$, $q > 1$, and $n > 0$.

This function computes the coefficients of the q-adic expansion of $e = k\ (q\hat{}n-1)/z$, of length n, without creating this number, which may be a large integer although all arguments are small (see [BHHK21, Remark 2.23 (iv)] and section 2.2.1).

If $e = v_n\ q^0 + v_{n-1}\ q^1 + \cdots + v_1\ q^{n-1}$ then the returned array is $[v_1, v_2, \ldots, v_n]$.

Example

```
gap> CoefficientsQadicReversed( 2, 7, 6, 2 );
[ 1, 4 ]
gap> e:= (6^2-1)/7;
5
gap> CoefficientsQadic( 2*e, 6 );
[ 4, 1 ]
```

3.2 Visualization of Singer Algebras

The following functions can be used to show the canonical basis of a Singer algebra in a two-dimensional array, where the rows correspond to Loewy layers.

3.2.1 DisplaySingerMonomials

▷ `DisplaySingerMonomials(A)` (function)
▷ `DisplaySingerMonomials(q, z)` (function)
▷ `DisplaySingerMonomials(q, n, e)` (function)

Returns: nothing.

Let `A` be a Singer algebra $A[q,n,z]$, where n is the multiplicative order of q modulo z, or let q and z be two coprime integers that define such an algebra $A[q,n,z]$, or let q, n, e be parameters such that $z = (q^n - 1)/e$ holds.

`DisplaySingerMonomials` prints a table showing three column parts: The first column contains the positions $0, 1, \ldots$ of the Loewy layers of `A`, the second column contains the dimensions of the Loewy layers, and the remaining columns describe the elements of the canonical basis of `A` in each layer; by default, the element b_i is represented by i, but when the global option `m` is present (which stands for "monomials") then B_i is represented by the coefficients of the q-adic expansion of ie.

Example

```
gap> DisplaySingerMonomials( 2, 7 );  # e = 1
A[2,3,7]

0 | 1 | 0
1 | 3 | 1 2 4
2 | 3 | 3 5 6
3 | 1 | 7
gap> DisplaySingerMonomials( 2, 7 : m );  # same, show monomials
A[2,3,7]

0 | 1 | (0,0,0)
1 | 3 | (0,0,1) (0,1,0) (1,0,0)
2 | 3 | (0,1,1) (1,0,1) (1,1,0)
3 | 1 | (1,1,1)
gap> DisplaySingerMonomials( 9, 8 : m );  # uniserial case
A[9,1,8]
```

```
0 | 1 | (0)
1 | 1 | (1)
2 | 1 | (2)
3 | 1 | (3)
4 | 1 | (4)
5 | 1 | (5)
6 | 1 | (6)
7 | 1 | (7)
8 | 1 | (8)
gap> DisplaySingerMonomials( 6, 7 : m );  # Loewy length 3
A[6,2,7]

0 | 1 | (0,0)
1 | 6 | (0,5) (1,4) (2,3) (3,2) (4,1) (5,0)
2 | 1 | (5,5)
gap> DisplaySingerMonomials( 14, 15 : m );  # wrapped output
A[14,2,15]

0 |  1 | ( 0, 0)
1 | 14 | ( 0,13) ( 1,12) ( 2,11) ( 3,10) ( 4, 9) ( 5, 8) ( 6, 7)
  |    | ( 7, 6) ( 8, 5) ( 9, 4) (10, 3) (11, 2) (12, 1) (13, 0)
2 |  1 | (13,13)
```

3.2.2 BrowseSingerMonomials

▷ `BrowseSingerMonomials(A)` (function)
▷ `BrowseSingerMonomials(q, z)` (function)
▷ `BrowseSingerMonomials(q, n, e)` (function)

Returns: nothing.

Let `A` be a Singer algebra $A[q,n,z]$, where n is the multiplicative order of `q` modulo `z`, or let `q` and `z` be two coprime integers that define such an algebra $A[q,n,z]$, or let `q`, `n`, `e` be parameters such that $z = (q^n - 1)/e$ holds.

`BrowseSingerMonomials` opens a Browse table showing three column parts: The first column contains the positions $0, 1, \ldots$ of the Loewy layers of `A`, the second column contains the dimensions of the Loewy layers, and the remaining columns describe the elements of the canonical basis of `A` in each layer; by default, the element b_i is represented by i, but when the global option `m` is present (which stands for "monomials") then B_i is represented by the coefficients of the q-adic expansion of ie. The first two columns are regarded as row labels, that is, their horizontal position is fixed on the screen when one scrolls to the left or right in the third column

`BrowseSingerMonomials` is based on the `NCurses.BrowseDenseList` (**Browse: NCurses.BrowseDenseList**) function from the Browse package [BL20]; it is available only if this package is available.

Example

```
gap> if IsBound( BrowseSingerMonomials ) then
>   n:= [ 14, 14, 14 ];;  # ‘‘do nothing’’ input
>   BrowseData.SetReplay( Concatenation( n, n, "Q" ) );
>   BrowseSingerMonomials( 2, 7 );
>   BrowseData.SetReplay( false );
> fi;
```

3.2.3 Changing the display format: User preference DisplayFunction

The way how the function DisplaySingerMonomials (3.2.1) shows tabular information can be customized via the user preference "DisplayFunction" of the SingerAlg package. The value must be a string that evaluates to a GAP function. Useful values are "Print" (see Print (**Reference: Print**)), "PrintFormattedString" (see PrintFormattedString (**GAPDoc: PrintFormattedString**) in [LN19]), and "SingerAlg.Pager", which means that Pager (**Reference: Pager**) is called with the formatted option, which is necessary for switching off GAP's automatic line breaking. The default value is "SingerAlg.Pager" if GAPInfo.TermEncoding has the value "UTF-8", and "Print" otherwise.

3.3 Singer Algebras as Combinatorial Structures

Many interesting subalgebras and subspaces U, say, of a Singer algebra $A = A[q,z]$ or $A = A[q,z]_p$ have vector space bases that are subsets of the canonical basis $B(A)$, that is, U is spanned by the set $U \cap B(A)$. We call these subspaces *combinatorial*, and write $B(U) = U \cap B(A)$, the canonical basis of U. The examples of combinatorial subspaces for which implementations are provided are listed below.

Each of the GAP functions in question returns the set of indices i such that the basis vectors $B(A[q,z])_i$ are members of $B(U)$. The idea is that this set can be computed combinatorially, without performing computations with elements of the algebra A, and that for example the product space of two combinatorial subspaces is again combinatorial. The availability of the attribute GeneratingSubsetOfCanonicalBasisOfSingerAlgebra (3.3.1) in a subspace U marks U as combinatorial, and the attribute value is the set of indices of $B(U)$.

3.3.1 GeneratingSubsetOfCanonicalBasisOfSingerAlgebra

▷ GeneratingSubsetOfCanonicalBasisOfSingerAlgebra(*U*) (attribute)

Let *U* be a subspace of a Singer algebra A, say. If this attribute is set in *U* then the value is a strictly sorted list of nonnegative integers such that the corresponding subset of the canonical basis of A is a basis of *U*. In particular, *U* is a combinatorial subspace of A, see the introduction of Section 3.3.

There is no default method for computing the value of this attribute. On the other hand, if the value is known then there are efficient methods to compute annihilators, product spaces, etc.

Example

```
gap> A:= SingerAlgebra( 3, 7 );
A[3,6,7]
gap> J:= RadicalOfAlgebra( A );;
gap> HasGeneratingSubsetOfCanonicalBasisOfSingerAlgebra( J );
true
gap> GeneratingSubsetOfCanonicalBasisOfSingerAlgebra( J );
[ 2 .. 8 ]
gap> P:= ProductSpace( J, J );
<vector space of dimension 1 over Rationals>
gap> HasGeneratingSubsetOfCanonicalBasisOfSingerAlgebra( P );
true
gap> GeneratingSubsetOfCanonicalBasisOfSingerAlgebra( P );
[ 8 ]
```

```
gap> V:= Subspace( A, Basis( A ){ [ 2, 3 ] } );;
gap> HasGeneratingSubsetOfCanonicalBasisOfSingerAlgebra( V );
false
```

3.3.2 SingerAlg.MultTable

▷ `SingerAlg.MultTable(`*data*`)` (function)

Let *data* be the record returned by `LoewyStructureInfoGAP` (3.1.11) that describes a Singer algebra $A[q,z]$, and let $B = B(A)$. `SingerAlg.MultTable` returns the $(z+1) \times (z+1)$ matrix that contains at the position (i,j) the value $i+j-1$ if the product $B_i \cdot B_j$ is nonzero (hence equal to B_{i+j-1}, the $i+j-1$-th basis vector), and 0 otherwise.

Example

```
gap> data:= LoewyStructureInfoGAP( 2, 3, 7 );;
gap> Display( SingerAlg.MultTable( data ) );
[ [  1,  2,  3,  4,  5,  6,  7,  8 ],
  [  2,  0,  4,  0,  6,  0,  8,  0 ],
  [  3,  4,  0,  0,  7,  8,  0,  0 ],
  [  4,  0,  0,  0,  8,  0,  0,  0 ],
  [  5,  6,  7,  8,  0,  0,  0,  0 ],
  [  6,  0,  8,  0,  0,  0,  0,  0 ],
  [  7,  8,  0,  0,  0,  0,  0,  0 ],
  [  8,  0,  0,  0,  0,  0,  0,  0 ] ]
```

The multiplication table of a Singer algebra of dimension N has the value N on the antidiagonal ($i+j = N+1$), is zero below the antidiagonal, and contains only $N-i$ and zero on the i parallel above the antidiagonal.

3.3.3 SingerAlg.BasisOfSum and SingerAlg.BasisOfIntersection

▷ `SingerAlg.BasisOfSum(`*data, I, J*`)` (function)
▷ `SingerAlg.BasisOfIntersection(`*data, I, J*`)` (function)

For two subsets *I*, *J* of $\{1,2,\ldots,z+1\}$, these functions just return the union and the intersection, respectively, of *I* and *J*.

I and *J* describe subsets of a basis, which generate the spaces U and V, say, then the result describes the subset of this basis that generates the sum and the intersection, respectively, of U and V.

Example

```
gap> SingerAlg.BasisOfSum( data, [ 1, 2, 3 ], [ 2, 4, 6 ] );
[ 1, 2, 3, 4, 6 ]
gap> SingerAlg.BasisOfIntersection( data, [ 1, 2, 3 ], [ 2, 4, 6 ] );
[ 2 ]
```

3.3.4 SingerAlg.BasisOfProductSpace

▷ `SingerAlg.BasisOfProductSpace(`*data, I, J*`)` (function)

Let *data* be the record returned by LoewyStructureInfoGAP (3.1.11) that describes a Singer algebra $A = A[q,z]$, let $B = B(A)$, and let *I*, *J* be subsets of $\{1,2,\ldots,z+1\}$, describing subspaces U, V of A with bases $(B_i; i \in I)$ and $(B_i; i \in J)$, respectively. SingerAlg.BasisOfProductSpace returns the subset K of $\{1,2,\ldots,z+1\}$ such that $(B_i; i \in K)$ is a basis of the product space $U \cdot V$.

Example

```
gap> data:= LoewyStructureInfoGAP( 2, 7 );;
gap> radser:= SingerAlg.BasesOfRadicalSeries( data );
[ [ 2 .. 8 ], [ 4, 6, 7, 8 ], [ 8 ] ]
gap> SingerAlg.BasisOfProductSpace( data, radser[1], radser[1] );
[ 4, 6, 7, 8 ]
gap> SingerAlg.BasisOfProductSpace( data, radser[1], radser[2] );
[ 8 ]
gap> SingerAlg.BasisOfProductSpace( data, radser[2], radser[2] );
[  ]
```

3.3.5 SingerAlg.BasisOfIdeal

▷ SingerAlg.BasisOfIdeal(*data, I*) (function)

Let *data* be the record returned by LoewyStructureInfoGAP (3.1.11) that describes a Singer algebra $A = A[q,z]$, let $B = B(A)$, and let *I* be a subset of $\{1,2,\ldots,z+1\}$, describing a subspace U of A with basis $(B_i; i \in I)$. SingerAlg.BasisOfIdeal returns the subset J of $\{1,2,\ldots,z+1\}$ such that $(B_i; i \in J)$ is a basis of the ideal $U \cdot A$.

Example

```
gap> data:= LoewyStructureInfoGAP( 2, 7 );;
gap> SingerAlg.BasisOfIdeal( data, [ 4 ] );
[ 4, 8 ]
```

3.3.6 SingerAlg.BasisOfAnnihilator

▷ SingerAlg.BasisOfAnnihilator(*data, I*) (function)

Let *data* be the record returned by LoewyStructureInfoGAP (3.1.11) that describes a Singer algebra $A = A[q,z]$, let $B = B(A)$, and let *I* be a subset of $\{1,2,\ldots,z+1\}$, describing a subspace U of A with basis $(B_i; i \in I)$. SingerAlg.BasisOfAnnihilator returns the subset J of $\{1,2,\ldots,z+1\}$ such that $(B_i; i \in J)$ is a basis of the annihilator $\{x \in A; x \cdot U = 0\}$ of U in A.

Example

```
gap> data:= LoewyStructureInfoGAP( 2, 7 );;
gap> radser:= SingerAlg.BasesOfRadicalSeries( data );
[ [ 2 .. 8 ], [ 4, 6, 7, 8 ], [ 8 ] ]
gap> List( radser, I -> SingerAlg.BasisOfAnnihilator( data, I ) );
[ [ 8 ], [ 4, 6, 7, 8 ], [ 2, 3, 4, 5, 6, 7, 8 ] ]
```

3.3.7 SingerAlg.BasesOfRadicalSeries

▷ SingerAlg.BasesOfRadicalSeries(*data*) (function)

Let *data* be the record returned by LoewyStructureInfoGAP (3.1.11) that describes a Singer algebra $A = A[q,z]$, and let $B = B(A)$. SingerAlg.BasesOfRadicalSeries returns the list $[I_1, I_2, \ldots, I_l]$ of subsets of $\{1,2,\ldots,z+1\}$ such that $(B_i; i \in I_j)$ is a basis of the j-th power of the Jacobson radical J of A, and such that J^l is nonzero and J^{l+1} is zero.

Example

```
gap> data:= LoewyStructureInfoGAP( 2, 7 );;
gap> radser:= SingerAlg.BasesOfRadicalSeries( data );
[ [ 2 .. 8 ], [ 4, 6, 7, 8 ], [ 8 ] ]
```

3.3.8 SingerAlg.BasesOfSocleSeries

▷ SingerAlg.BasesOfSocleSeries(*data*) (function)

Let *data* be the record returned by LoewyStructureInfoGAP (3.1.11) that describes a Singer algebra $A = A[q,z]$, and let $B = B(A)$. SingerAlg.BasesOfSocleSeries returns the list $[I_1, I_2, \ldots, I_l]$ of subsets of $\{1,2,\ldots,z+1\}$ such that $(B_i; i \in I_j)$ is a basis of S_j, where S_1 is the socle of A, S_{j+1}/S_j is the socle of A/S_j, and A/S_l is nonzero and its own socle.

Example

```
gap> socser:= SingerAlg.BasesOfSocleSeries( data );
[ [ 8 ], [ 4, 6, 7, 8 ], [ 2 .. 8 ] ]
```

3.3.9 SingerAlg.BasisOfPowers

▷ SingerAlg.BasisOfPowers(*data, I, p, m*) (function)

Let p be a prime integer, let *data* be the record returned by LoewyStructureInfoGAP (3.1.11) that describes a Singer algebra $A = A[q,z]$, let $B = B(A_p)$, let I be a subset of $\{1,2,\ldots,z+1\}$, describing a subspace U of A_p with basis $(B_i; i \in I)$, and let m be a positive integer. SingerAlg.BasisOfPowers returns the subset J of $\{1,2,\ldots,z+1\}$ such that $(B_i; i \in J)$ is a basis of the subspace $\{x^{p^m}; x \in U\}$ of A_p.

Example

```
gap> data:= LoewyStructureInfoGAP( 3, 8 );;
gap> SingerAlg.BasisOfPowers( data, [ 1 .. 9 ], 2, 1 );
[ 1, 3, 7, 9 ]
gap> SingerAlg.BasisOfPowers( data, [ 1 .. 9 ], 2, 2 );
[ 1 ]
gap> SingerAlg.BasisOfPowers( data, [ 1 .. 9 ], 3, 1 );
[ 1 ]
```

3.3.10 SingerAlg.BasisOfPMRoots

▷ SingerAlg.BasisOfPMRoots(*data, I, p, m*) (function)

Let p be a prime integer, let *data* be the record returned by LoewyStructureInfoGAP (3.1.11) that describes a Singer algebra $A = A[q,z]$, let $B = B(A_p)$, let I be a subset of $\{1,2,\ldots,z+1\}$, describing a subspace U of A_p with basis $(B_i; i \in I)$, and let m be a positive integer. (See Section 1.2 for the definition of A_p.)

`SingerAlg.BasisOfPMRoots` returns the subset J of $\{1,2,\ldots,z+1\}$ such that $(B_i; i \in J)$ is a basis of the subspace $\{x \in A_p; x^{p^m} \in U\}$ of A_p.

Example

```
gap> data:= LoewyStructureInfoGAP( 3, 8 );;
gap> SingerAlg.BasisOfPMRoots( data, [], 2, 1 );
[ 3, 6, 7, 8, 9 ]
gap> SingerAlg.BasisOfPMRoots( data, [], 2, 2 );
[ 2, 3, 4, 5, 6, 7, 8, 9 ]
gap> SingerAlg.BasisOfPMRoots( data, [ 3 ], 2, 1 );
[ 2, 3, 6, 7, 8, 9 ]
```

3.3.11 SingerAlg.BasisOfPC

▷ `SingerAlg.BasisOfPC(`*`data, I, J`*`)` (function)

Let *`data`* be the record returned by `LoewyStructureInfoGAP` (3.1.11) that describes a Singer algebra $A = A[q,z]$, let $B = B(A)$, let *`I`* and J be subsets of $\{1,2,\ldots,z+1\}$, describing subspaces U and V of A with bases $(B_i; i \in I)$ and $(B_i; i \in J)$, respectively. `SingerAlg.BasisOfPC` returns the subset K of $\{2,3,\ldots,z+1\}$ such that $(B_i; i \in K)$ is a basis of the subspace $\{x \in J(A); x \cdot U \subseteq V\}$ of $J(A)$.

(The perhaps strange name "BasisOfPC" was chosen because the result contains the indices of those basis vectors such that the *P*roduct with the space U is *C*ontained in the space V.)

Example

```
gap> data:= LoewyStructureInfoGAP( 23, 585 );;
gap> soc:= SingerAlg.BasesOfSocleSeries( data );;
gap> rad:= SingerAlg.BasesOfRadicalSeries( data );;
gap> I1:= SingerAlg.BasisOfPC( data, soc[3], rad[3] );;
gap> Length( I1 );
581
gap> data:= LoewyStructureInfoGAP( 212, 585 );;
gap> soc:= SingerAlg.BasesOfSocleSeries( data );;
gap> rad:= SingerAlg.BasesOfRadicalSeries( data );;
gap> I2:= SingerAlg.BasisOfPC( data, soc[3], rad[3] );;
gap> Length( I2 );
545
```

3.4 Permutation Isomorphism of Singer Algebras

Proving that two given Singer algebras are isomorphic is in general hard. An easier question is whether an algebra isomorphism exists that maps the canonical basis of the first to the canonical basis of the second. (And fortunately this situation occurs in quite a few cases, see 4.3.4.)

The following functions can be used to check for such permutation isomorphisms.

3.4.1 SingerAlg.IsInducedAlgebraIsomorphism

▷ `SingerAlg.IsInducedAlgebraIsomorphism(`*`t1, t2, perm`*`)` (function)

Returns: `true` or `false`.

Let *t1* and *t2* be the multiplication tables of two Singer algebras A_1, A_2, respectively, of the same dimension $z+1$, say, as returned by `SingerAlg.MultTable` (3.3.2). Let *perm* be a permutation with largest moved point at most z.

This function returns `true` if mapping the i-th vector of the canonical basis of A_1 to the i^*perm*-th vector of the canonical basis of A_2 defines an algebra isomorphism, and `false` otherwise.

Example

```
gap> # Loewy length 3, naturally isomorphic
gap> t1:= SingerAlg.MultTable( LoewyStructureInfo( 3, 7 ) );;
gap> t2:= SingerAlg.MultTable( LoewyStructureInfo( 6, 7 ) );;
gap> SingerAlg.IsInducedAlgebraIsomorphism( t1, t2, () );
true
gap> # q1 and q2 generate the same group of residues modulo z,
gap> # naturally isomorphic
gap> t1:= SingerAlg.MultTable( LoewyStructureInfo( 2, 7 ) );;
gap> t2:= SingerAlg.MultTable( LoewyStructureInfo( 4, 7 ) );;
gap> SingerAlg.IsInducedAlgebraIsomorphism( t1, t2, () );
true
gap> # one of the explicitly computed natural isomorphisms
gap> t1:= SingerAlg.MultTable( LoewyStructureInfo( 3, 65 ) );;
gap> t2:= SingerAlg.MultTable( LoewyStructureInfo( 9, 65 ) );;
gap> SingerAlg.IsInducedAlgebraIsomorphism( t1, t2, () );
true
gap> # one of the explicitly computed permutation isomorphisms
gap> # that are not natural isomorphisms
gap> t1:= SingerAlg.MultTable( LoewyStructureInfo(  41, 275 ) );;
gap> t2:= SingerAlg.MultTable( LoewyStructureInfo( 116, 275 ) );;
gap> SingerAlg.IsInducedAlgebraIsomorphism( t1, t2, () );
false
gap> pi:= (2,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3)
> (28,29,30,31,32)(38,39,40,41,42)(43,44,45,46,47,48,49,50)
> (54,55,57,79,77,94,91,89,87,83,82,81,80,78,76,75,74,73,72,64,65,63,62,
> 61,60)
> (85,99,97,93,90,88,86)(92,108,107,106,105,98,96,95)
> (109,110,112,113,115)
> (114,116,117,120,118,121,124,127,119,122,125,129,132,149,146,144,143,
> 142,141,140,154,151,147,163,161,160,157,159,156,153,150,158,155,152,
> 148,145,128,131,133,134,135,136,137,123,126,130)(162,168,167,165,164)
> (169,170,171,172,179,181,182,185)(178,180,184,187,189,191,192)
> (183,186,188,190,194,195,196,197,199,201,202,203,204,205,213,212,214,
> 215,216,217,223,222,220,198,200)(227,234,233,232,231,230,229,228)
> (235,239,238,237,236)(245,249,248,247,246)
> (253,254,255,256,257,258,259,260,261,262,263,264,265,266,267,268,269,
> 270,271,272,273,274,275);;
gap> SingerAlg.IsInducedAlgebraIsomorphism( t1, t2, pi );
true
gap> SingerAlg.IsInducedAlgebraIsomorphism( t1, t2, pi^-1 );
false
```

3.4.2 SingerAlg.ProposedPermutationIsomorphism

▷ `SingerAlg.ProposedPermutationIsomorphism(`*data1, data2*`)` (function)

Returns: a string, a permutation, or `fail`.

Let *data1* and *data2* be the `LoewyStructureInfo` (3.1.11) values of two Singer algebras A_1, A_2, respectively, of the same dimension $z+1$, say.

This function returns either a string that describes why there is no algebra isomorphism induced by mapping the canonical basis of A_1 to the permuted canonical basis of A_2, or `fail` (indicating that the relevant functionality of the GraPe package is not available), or a permutation π such that at least the following necessary conditions for such a mapping are satisfied.

- π maps pairs (i,j) where $b_i \cdot b_j$ is zero to pairs (k,l) where $B_k \cdot B_l$ is zero.
- The basis vectors b_i and $B_{\pi(i)}$ lie in the same Loewy layer and in the same socle layer.
- The basis vector b_i is a p^m-th power if and only if $B_{\pi(i)}$ is a p^m-th power.

The first condition is forced via a call to the function `GraphIsomorphism` (**GRAPE: GraphIsomorphism**) (which is based on [McK90]) from the GAP package GraPe [Soi19] with two graphs whose incidence relation is defined by this property, and the other conditions are translated into colorings of these graphs.

When a permutation is returned, one can check with `SingerAlg.IsInducedAlgebraIsomorphism` (3.4.1) whether it does in fact induce an algebra isomorphism. (For Singer algebras $A[q,z]$ with $z \leq 10000$, this is always the case, see Section 4.3.4.)

Example

```
gap> # a canonical isomorphism
gap> data1:= LoewyStructureInfo( 3, 7 );;
gap> data2:= LoewyStructureInfo( 6, 7 );;
gap> SingerAlg.ProposedPermutationIsomorphism( data1, data2 );
()
gap> # a proper permutation isomorphism
gap> data1:= LoewyStructureInfo(  41, 275 );;
gap> data2:= LoewyStructureInfo( 116, 275 );;
gap> pi:= SingerAlg.ProposedPermutationIsomorphism( data1, data2 );;
gap> t1:= SingerAlg.MultTable( data1 );;
gap> t2:= SingerAlg.MultTable( data2 );;
gap> SingerAlg.IsInducedAlgebraIsomorphism( t1, t2, pi );
true
gap> # no permutation isomorphism
gap> data1:= LoewyStructureInfo(  11, 171 );;
gap> data2:= LoewyStructureInfo(  68, 171 );;
gap> SingerAlg.ProposedPermutationIsomorphism( data1, data2 );
"different distributions of products"
```

3.5 Invariants of Singer Algebras

The following functions can be used to prove the nonisomorphism of given Singer algebras via invariants.

3.5.1 ConsiderInvariantsByParameters

▷ `ConsiderInvariantsByParameters(z, qs[, bounds])` (function)

Returns: a record.

Let $z \in \{1,2,\ldots,10000\}$, and `qs` be a list of prime residues modulo `z`. `ConsiderInvariantsByParameters` tries to find an invariant (under algebra isomorphisms) that is not equal for all Singer algebras $A[q,z]$, for $q \in$ `qs`. The invariants used here are the dimensions of suitable subspaces or the numbers of solutions of suitable equations.

The function returns a record with at least the components `success` (with value `true` if an invariant was found that distinguishes at least two algebras corresponding to `qs`, and `false` otherwise) and `comment` (a string). If the search was successful then the result contains also the components `label` (a string that decribes the distinguishing invariant) and `lists` (a partition of `qs` according to the values of this invariant).

If a record `bounds` is given then it controls how many checks are performed, as follows.

- If the component `maxnumber` is bound then the function returns a record with `success` value `false` as soon as at least `maxnumber` invariants have been checked without success; the default value of `maxnumber` is 100.
- If the components `RCdim` and `RCnum` are bound then the function `SingerAlg.NumberOfProductsInSubspace` (3.5.3) gets called after the combinatorial invariants have been computed, with the first two arguments taken from the first `RCnum` of these invariants and with third argument equal to the value of `RCdim`; by default, `SingerAlg.NumberOfProductsInSubspace` (3.5.3) is not called at all.

  If such a call is successful for two subspaces U and V then the `label` component of the result has the form `"RC(`U`,`V`)"`.
- If the component `LL4QuoDerMax` is bound then its value is taken as the `maxdim` argument of `SingerAlg.LL4QuoDerDim` (3.5.4), the default is 50.
- If the component `SubquoDerMax` is bound then its value is taken as the `maxdim` argument of `SingerAlg.SubquoDerDim` (3.5.5), the default is 50.

The following invariant subspaces of a Singer algebra A or its reduction A_p modulo some prime p, respectively, are considered.

- The members $J(A)^i$ of the radical series of A (denoted by `"J^1"`, `"J^2"`, ... in the `label` component; see `SingerAlg.BasesOfRadicalSeries` (3.3.7)),
- the members $S(A)_i$ of the socle series of A (denoted by `"S_1"`, `"S_2"`, ...; see `SingerAlg.BasesOfSocleSeries` (3.3.8)),
- the space of elements in A_p whose p^m-th powers are zero, for primes p and positive integers m (denoted by `"Roots(0,`p`,`m`)"`; see `SingerAlg.BasisOfPMRoots` (3.3.10)),
- the ideal in A or A_p that is spanned by an invariant subspace U described in this list (denoted by `"Ideal(`U`)"`; see `SingerAlg.BasisOfIdeal` (3.3.5)),
- the annihilator in A or A_p of an invariant subspace U described in this list (denoted by `"Annihilator(`U`)"`; see `SingerAlg.BasisOfAnnihilator` (3.3.6)),

- the space of p^m-th powers of elements in U, for primes p and positive integers m, where U is an invariant subspace of A or A_p described in this list (denoted by "Power(U,p,m)"; see SingerAlg.BasisOfPowers (3.3.9)),
- the space of elements in A_p whose p^m-th powers are in U, for primes p and positive integers m, where U is an invariant subspace of A or A_p described in this list (denoted by "Roots(U,p,m)"; see SingerAlg.BasisOfPMRoots (3.3.10)),
- the product space of two subspaces U, V in the same characteristic (denoted by "Prod(U,V)"; see SingerAlg.BasisOfProductSpace (3.3.4)),
- the sum of two subspaces U, V in the same characteristic (denoted by "Sum(U,V)"; see SingerAlg.BasisOfSum (3.3.3)),
- the intersection of two subspaces U, V in the same characteristic (denoted by "Intersection(U,V)"; see SingerAlg.BasisOfIntersection (3.3.3)),
- the space of elements $\{x \in J(A); x \cdot U \subseteq V\}$, for two subspaces U, V in the same characteristic (denoted by "PC(U,V)"; see SingerAlg.BasisOfPC (3.3.11)),

A basis for the invariant subspace can be recovered from the label string of the result, using the function SingerAlg.InfoFromInvariantString (3.5.2).

A different kind of invariant is given by the number of solutions of some equation, as computed with SingerAlg.NumberOfProductsInSubspace (3.5.3), the corresponding label has the form "RC(U,V)", and the function SingerAlg.InfoFromInvariantString (3.5.2) can be used to compute the number of solutions.

Example

```
gap> inv:= ConsiderInvariantsByParameters( 117, [ 29, 35 ] );
rec( comment := "total 12 invariants checked",
  label := "Prod(Annihilator(Power(J^1,2,1)),Roots(0,2,1))",
  lists := [ [ 29 ], [ 35 ] ], success := true )
gap> ConsiderInvariantsByParameters( 247, [ 37, 46 ] );
rec( comment := "total 4 invariants plus LL4QuoDerDim checked",
  label := "LL4QuoDerDim", lists := [ [ 46 ], [ 37 ] ],
  success := true )
gap> ConsiderInvariantsByParameters( 171, [ 11, 68 ] );
rec( comment := "no decision, checked 11 invariants",
  labels := [ "J^3", "Power(J^1,2,1)", "Annihilator(Roots(0,2,1))",
      "Sum(Power(J^1,2,1),J^3)",
      "Sum(Power(J^1,2,1),Annihilator(Roots(0,2,1)))", "J^2", "S_2",
      "Annihilator(Sum(Power(J^1,2,1),Annihilator(Roots(0,2,1))))",
      "Roots(0,2,1)", "Annihilator(Power(J^1,2,1))", "J^1" ],
  success := false )
gap> res:= ConsiderInvariantsByParameters( 171, [ 11, 68 ],
>             rec( SubquoDerMax:= 158 ) );;
gap> res.success;
true
```

3.5.2 SingerAlg.InfoFromInvariantString

▷ SingerAlg.InfoFromInvariantString(*data*, *str*) (function)

Returns: a record.

Let *data* be a GAP record as returned by `LoewyStructureInfoGAP` (3.1.11) or a Julia dictionary as returned by `LoewyStructureInfoJulia` (3.1.11), which describes a Singer algebra A, say, and let *str* be a string as in the `label` component of a record returned by `ConsiderInvariantsByParameters` (3.5.1).

`SingerAlg.InfoFromInvariantString` returns a record with one of the following components.

`basisIndices`:
: the list I of indices such that $\{B(A)_i; i \in I\}$ is a basis of the subspace defined by *str*,

`derivationsDim`:
: the dimension of the algebra of derivations (see `Derivations` (**Reference: Derivations**)) of the algebra,

`LL4QuoDerDim`:
: the dimension of the algebra of derivations of the algebra considered by the function `SingerAlg.LL4QuoDerDim` (3.5.4) or its Julia variant,

`SubquoDerDim`:
: the dimension of the algebra of derivations of the subquotient described by *str* that is considered by the function `SingerAlg.SubquoDerDim` (3.5.5) or its Julia variant,

`solutionCount`:
: the pair $[e,o]$ such that the number of solutions is $2^e \cdot o$, where o is an odd number.

If *data* is a GAP record then the invariants in question are computed with GAP functions, otherwise the Julia implementations are used.

Example

```
gap> SingerAlg.InfoFromInvariantString(
>        LoewyStructureInfoGAP( 29, 6, 117 ),
>        "Prod(Annihilator(Power(J^1,2,1)),Roots(0,2,1))" );
rec( basisIndices := [ 36, 64, 69, 73, 80, 93, 95, 100, 101, 118 ] )
gap> SingerAlg.InfoFromInvariantString(
>        LoewyStructureInfoGAP( 35, 6, 117 ),
>        "Prod(Annihilator(Power(J^1,2,1)),Roots(0,2,1))" );
rec( basisIndices := [ 37, 51, 60, 64, 73, 77, 86, 87, 91, 100, 109,
      113, 118 ] )
gap> SingerAlg.InfoFromInvariantString(
>        LoewyStructureInfoGAP( 2, 3, 7 ),
>        "Derivations" );
rec( derivationsDim := 24 )
gap> SingerAlg.InfoFromInvariantString(
>        LoewyStructureInfoGAP( 37, 12, 247 ),
>        "LL4QuoDerDim" );
rec( LL4QuoDerDim := 656 )
gap> SingerAlg.InfoFromInvariantString(
>        LoewyStructureInfoGAP( 68, 6, 171 ),
>        "SubquoDerDim(J^2,J^3)" );
rec( SubquoDerDim := 2025 )
gap> SingerAlg.InfoFromInvariantString(
>        LoewyStructureInfoGAP( 23, 12, 259 ),
>        "RC(J^3,J^1)" );
rec( solutionCount := [ 492, 51055 ] )
```

3.5.3 SingerAlg.NumberOfProductsInSubspace

▷ `SingerAlg.NumberOfProductsInSubspace(data, J, I[, bound])` (function)

Returns: a pair of nonnegative integers, or `fail`.

Let *data* be a record as returned by `LoewyStructureInfoGAP` (3.1.11) that describes a Singer algebra $A = A[q,z]$, let $B = B(A)$, and let J, I be subsets of $\{1,2,\ldots,z+1\}$, describing subspaces V and U of A with bases $(B_i; i \in J)$ and $(B_i; i \in I)$, respectively.

`SingerAlg.NumberOfProductsInSubspace` tries to compute the cardinality of the set

$$X = \{(x,y) \in U \times U; x \cdot y \in V\}.$$

For that, we set

$$\begin{array}{rcl} K & = & \{i \in I; B_i \cdot B_j \notin V \text{ for some } j \in I\}, \\ S & = & \{k; 0 \neq B_i \cdot B_j = B_k \text{ for some } i,j \in I\}, \\ D & = & \{k \in S; B_k \notin V\}. \end{array}$$

If $|K|$ is larger than *bound* then `fail` is returned, otherwise the list $[e,o]$ such that $|X| = 2^e \cdot o$ holds. The default value for *bound* is 15, the maximal possible value for *bound* is 59 (which is much too large for practical purposes).

Example

```
gap> data:= LoewyStructureInfoGAP( 23, 12, 259 );;
gap> radser:= SingerAlg.BasesOfRadicalSeries( data );;
gap> SingerAlg.NumberOfProductsInSubspace( data, radser[3], radser[1] );
[ 492, 51055 ]
gap> data:= LoewyStructureInfoGAP( 60, 12, 259 );;
gap> radser:= SingerAlg.BasesOfRadicalSeries( data );;
gap> SingerAlg.NumberOfProductsInSubspace( data, radser[3], radser[1] );
[ 493, 161051 ]
```

Note that for $x = \sum_{i \in I} x_i B_i$ and $y = \sum_{j \in I} y_j B_j$, we have

$$x \cdot y = \sum_{i,j \in I} x_i y_j (B_i \cdot B_j) = v + \sum_{i,j \in K} x_i y_j (B_i \cdot B_j) = v' + \sum_{k \in D} \Big(\sum_{i,j \in K, B_i \cdot B_j = B_k} x_i y_j \Big) B_k,$$

for some $v, v' \in V$, thus $x \cdot y \in V$ if and only if $x' M_k y' = 0$ holds for all $k \in D$, where $[M_k]_{i,j}$ is 1 if $B_i \cdot B_j = B_k$ holds, and 0 otherwise, for $i,j \in K$, and where x', y' are the restrictions of x, y to K. Let $M(x')$ be the matrix with rows $x' M_k$, $k \in D$. Then

$$|X| = 4^{|I|-|K|} \cdot \sum_{x' \in F_2^{|K|}} 2^{|K| - rank(M(x'))}.$$

3.5.4 SingerAlg.LL4QuoDerDim

▷ `SingerAlg.LL4QuoDerDim(data[, maxdim])` (function)

Returns: a positive integer, or `fail`.

Let *data* be a record as returned by `LoewyStructureInfoGAP` (3.1.11) that describes a Singer algebra $A = A[q,z]$ of Loewy length 4. We consider A as an algebra over the field F with two elements.

Let J denote the Jacobson radical of A, $S_2(A)$ denote the annihilator of J^2, and let $V(A)$ be the F-vector space generated by those elements of the canonical basis of A that lie in $J(A) \setminus S_2(A)$.

`SingerAlg.LL4QuoDerDim` tries to compute the dimension of the algebra of derivations of the algebra $(J(A)^2 \oplus V(A))/S_1(A)$, using `SingerAlg.RightDerivationsDimension` (3.6.5). If the return value is a positive integer then it is this dimension. The return value `fail` means that either A has Loewy length different from 4 or that the dimension of $V(A)$ is larger than the optional argument *maxdim* (the default is 50).

(Note that we have $\dim(J(A)^2 \oplus V(A)) = 2\dim(J(A)^2) - 1$.)

Example

```
gap> data:= LoewyStructureInfoGAP( 37, 247 );;
gap> SingerAlg.LL4QuoDerDim( data );
656
gap> data:= LoewyStructureInfoGAP( 46, 247 );;
gap> SingerAlg.LL4QuoDerDim( data );
585
```

3.5.5 SingerAlg.SubquoDerDim

▷ `SingerAlg.SubquoDerDim(`*datalist, labels, subspaces[, maxdim]*`)` (function)

Returns: a record or `fail`.

Let *datalist* be a list of length n, say, of records as returned by `LoewyStructureInfoGAP` (3.1.11), which belong to Singer algebras $A_1, A_2, \ldots, A_n$ of the same dimension. We consider the A_i as algebras over the field F with two elements.

Let *labels* be a list of length m, say, that consists of strings.

Let *subspaces* be a list of length m, where each entry is a list of length n such that *subspaces*$[i][j]$ is a list of positive integers that describes the basis of the subspace of A_j that corresponds to the label *labels*$[i]$.

Let *maxdim* be a positive integer; the default is 50.

`SingerAlg.SubquoDerDim` runs over the pairs (i, j) such that, for $1 \leq k \leq n$, both *subspaces*$[i][k]$ and *subspaces*$[j][k]$ describe bases of ideals $I_{i,k}$ and $I_{j,k}$, respectively, in A_k, such that $I_{j,k} \subset I_{i,k}$ holds and $I_{i,k}/I_{j,k}$ has dimension at most *maxdim*. Then it computes the dimensions of the algebras of derivations of the quotients $I_{i,k}/I_{j,k}$.

If not all of these dimensions are equal for some pair (i, j) then a record with the components `labels` (the list `[ i, j ]`) and `derdim` (the list of computed dimensions) is returned. Otherwise `fail` is returned.

Example

```
gap> z:= 171;;  qs:= [ 11, 68 ];;
gap> datalist:= List( qs, q -> LoewyStructureInfoGAP( q, z ) );;
gap> ids:= [ Difference( [ 2 .. z+1 ], [ 10, 37, 55, 64, 82, 100 ] ),
>            [ 73, 91, 109, 118, 136, 163, 172 ] ];;
gap> ideals:= [ [ ids[1], ids[1] ], [ ids[2], ids[2] ] ];;
gap> labels:= [ "I", "J" ];;
gap> SingerAlg.SubquoDerDim( datalist, labels, ideals, 150 );
fail
gap> SingerAlg.SubquoDerDim( datalist, labels, ideals, 158 );
rec( derdim := [ 14179, 14201 ], labels := [ "I", "J" ] )
```

3.6 Miscellaneous variables related to Singer Algebras

3.6.1 SingerAlg

▷ SingerAlg (global variable)

This global record is used to store information about the SingerAlg package. Some of its components are documented individually, see the manual index.

3.6.2 OrderModExt

▷ OrderModExt(*n*, *m*[, *bound*]) (function)

Returns: a nonnegative integer.

When called with two arguments *n* and *m*, OrderModExt returns the same as the GAP library function OrderMod (**Reference: OrderMod**), that is, the multiplicative order of the integer *n* modulo the positive integer *m*. If *n* and *m* are not coprime the order of *n* is not defined and OrderModExt will return 0.

If *n* and *m* are relatively prime the multiplicative order of *n* modulo *m* is the smallest positive integer *i* such that $n^i \equiv 1 \pmod{m}$.

If no a priori known multiple *bound* of the desired order is given, OrderModExt usually spends most of its time factoring *m* for computing a default for *bound*, and then factoring *bound*. Thus it is advisable to enter *bound* whenever one knows a reasonable bound.

If an incorrect *bound* is given then the result will be wrong.

Example

```
gap> OrderModExt( 2, 7 );
3
gap> OrderModExt( 3, 7 );  # 3 is a primitive root modulo 7
6
gap> m:= (5^166-1) / 167;;      # about 10^113
gap> OrderModExt( 5, m, 166 );  # needs minutes without third argument
166
```

3.6.3 SingerAlg.ContentsOfDataFile

▷ SingerAlg.ContentsOfDataFile(*filename*) (attribute)

Returns: the GAP object stored in the given file.

Let *filename* be a string that denotes the name of a file in the data subdirectory of the SingerAlg package directory. SingerAlg.ContentsOfDataFile evaluates the contents of this file (assuming that it is GAP readable) and returns the result.

Example

```
gap> val:= SingerAlg.ContentsOfDataFile( "mqe.json" );;
gap> val[1][2];
"This file contains the sorted list of values '[ e, q, m(q,e) ]', "
```

3.6.4 GrayCodeSwitchIndexIterator

▷ GrayCodeSwitchIndexIterator(*n*) (function)

Returns: an iterator (see (**Reference: Iterators**)).

For a nonnegative integer `n`, this function returns an iterator for the sequence of the positions of those bits in an `n`-bit Gray code where the next flip takes place.

For `n` tending to infinity, this is series `A001511` of the OEIS [Slo].

Example

```
gap> l:= [];;
gap> for i in GrayCodeSwitchIndexIterator( 4 ) do
>      Add( l, i );
>    od;
gap> l;
[ 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1 ]
```

The idea of a Gray code is to run over the vectors of a $GF(2)$-vector space in such a way that subsequent vectors differ in exactly one position.

Example

```
gap> F:= GF(2);;  one:= One( F );;  V:= F^4;;
gap> v:= ShallowCopy( Zero( V ) );;  l:= [ ShallowCopy( v ) ];;
gap> for i in GrayCodeSwitchIndexIterator( 4 ) do
>      v[i]:= v[i] + one;
>      Add( l, ShallowCopy( v ) );
>    od;
gap> Set( l ) = Elements( V );
true
```

3.6.5 SingerAlg.RightDerivationsDimension

▷ `SingerAlg.RightDerivationsDimension(T)` (function)

Returns: a positive integer.

Let T be a symmetric square matrix with n rows and columns, whose entries are integers in $[0..n]$. We interpret T as the multiplication table of a commutative algebra A over the field with two elements, as follows. Let $B = [b_1, b_2, \ldots, b_n]$ be a basis of A. If $T[i, j] = k$ holds then the product $b_i b_j$ is zero if $k = 0$, and $b_i b_j = b_k$ otherwise.

`SingerAlg.RightDerivationsDimension` returns the dimension of (right) derivations of A.

Let the structure constants of A w.r.t. the basis B be $c_{i,j,k}$, that is, we have $b_i b_j = \sum_{k=1}^{n} c_{i,j,k} b_k$. For each pair (i, j), we have thus $c_{i,j,k} = 1$ if $T[i, j] = k(\neq 0)$ holds, and $c_{i,j,k} = 0$ otherwise. A linear map $D: b_i \mapsto \sum_{j=1}^{n} d_{i,j} b_j$ is a right derivation if the equation $\sum_{k=1}^{n} (c_{i,j,k} d_{k,m} - c_{k,j,m} d_{i,k} - c_{i,k,m} d_{j,k}) = 0$ is satisfied for all $1 \leq i, j, m \leq n$. Thus the right derivations of A are given by the solution space of this linear equation system, in terms of the n^2 indeterminates $d_{i,j}, 1 \leq i, j \leq n$, and we can compute the dimension of this space by computing the rank of the matrix of the equation system.

Since the algebra A is commutative, we have $c_{i,j,k} = c_{j,i,k}$, thus we need only the equations with $i \leq j$.

Example

```
gap> q:= 3;;  z:= 40;;  a:= SingerAlgebra( q, z, GF(2) );;
gap> T:= SingerAlg.MultTable( LoewyStructureInfo( 3, 40 ) );;
gap> SingerAlg.RightDerivationsDimension( T );
118
gap> Dimension( Derivations( CanonicalBasis( a ) ) );
118
```

Chapter 4

The Database of Low Dimensional Singer Algebras

4.1 Overview

Fix a positive integer z. In order to describe all Singer algebras $A[q,z]$, it is sufficient to consider one representative q for each cyclic subgroup of the group of prime residues modulo z, see Section 1.2. The database of Singer algebras with $1 \leq z \leq 10000$ is built according to this observation. That is, there is one entry for each such parameter pair (q,z), where we choose the smallest q from the generators of the subgroup it generates, except that $q = z+1$ is chosen instead of $q = 1$.

This implies that the number of data records for given z is exactly $\sum_q 1/\texttt{Phi(}\ \mathrm{ord}_z(q)\ \texttt{)}$.

Example

```
gap> ForAll( [ 1 .. 10000 ],
>            z -> Length( AllSingerAlgebraInfos( "z", z ) ) =
>                 Sum( PrimeResidues( z ),
>                      q -> 1 / Phi( OrderMod( q, z ) ) ) );
true
```

Note that two algebras $A[q,z]$, $A[q',z]$ for different such representatives q, q' can be isomorphic, and in fact this happens in many cases in a "natural" way (see Section 4.3.3). As soon as new theoretical criteria become known that admit a reduction of the set of parameters to describe all Singer algebras of a given dimension, the setup of the database may be changed.

The database stores the following information for the algebra $A[q,z]$.

- $n = \mathrm{ord}_z(q)$, the multiplicative order of q modulo z (`OrderMod( q, z )`, see `OrderModExt` (3.6.2)),
- $m = m(q,e)$, the minimal number of powers of q whose sum is divisible by $e = (q^n - 1)/z$,
- l, the Loewy length of $A[q,z]$,
- the Loewy vector $(v_1, v_2, \ldots, v_l)$ of $A[q,z]$, where $v_i = \dim(J^{i-1}/J^i)$, and $J = J(A[q,z])$ is the Jacobson radical of $A[q,z]$; we encode v by an abbreviated vector v' (see `LoewyVectorAbbreviated` (3.1.5)) where subsequent equal entries i with multiplicity $j \geq 1$ are abbreviated as $[i,j]$; thus $v' = [[1,j]]$ stands for $v = [1,1,\ldots,1]$ of length j;

- "decomposition information" d, if available, as follows: Set $q' = q \bmod m$; a decomposition $(x_1 \cdots x_n)^{q'-1} = \prod_{i=1}^{N} (x_1^{a_{i,1}} x_2^{a_{i,2}} \cdots x_n^{a_{i,n}})$ of length $N = l - 1 - n(q-q')/m$ is encoded by $d = [k_1, k_2, \ldots, k_N]$, where $ek_i = \sum_{j=1}^{n} a_{i,j} q^{j-1}$ holds; if no such decomposition is known then $d = 0$ is stored,
- the flag *dec* is 0 if $(x_1 \cdots x_n)^{q'-1}$ can be written as a product of monomials as above, each of degree m except at most one monomial of larger degree, and 1 otherwise,
- $delta = \lfloor n(q-1)/m \rfloor + 1 - l$ is the difference between the upper bound from [BHHK20, Theorem 7.1] and the Loewy length l,
- the `IdSingerAlgebra` (4.2.4) value of $A[q,z]$.

4.2 Access to the Database of Singer Algebras

4.2.1 OneSingerAlgebraInfo and AllSingerAlgebraInfos

▷ `OneSingerAlgebraInfo(`*`cond1, val1, cond2, val2, ...`*`)` (function)
▷ `AllSingerAlgebraInfos(`*`cond1, val1, cond2, val2, ...`*`)` (function)

Let *`cond1`*, *`cond2`*, ... be strings that describe precomputed properties of Singer algebras from the database, that is, they occur in the set { `"z"`, `"q"`, `"n"`, `"m"`, `"e"`, `"vprime"`, `"diff"` } (see above). The two functions compute those database entries such that the value for *`cond1`* matches *`val1`*, the value for *`cond2`* matches *`val2`*, etc., where "matches" means one of the following.

- *`val`* is equal to the *`cond`* component of the entry,
- *`val`* is a list in which the *`cond`* component of the entry occurs, or
- *`val`* is a unary function that returns `true` for the *`cond`* component of the entry.

It is also possible to enter GAP functions as some of the *`cond`* arguments. Each such function must take a record as is returned by `OneSingerAlgebraInfo`, and the value returned by the function gets compared with the corresponding *`val`* argument in the same way as described above.

`OneSingerAlgebraInfo` returns the first matching entry if there is one, and `fail` otherwise. `AllSingerAlgebraInfos` returns the set of all matching entries.

Example

```
gap> OneSingerAlgebraInfo( "z", 8 );
rec( LL := 5, d := [  ], dec := 0, diff := 0, e := 1,
  isom := [ 8, 3 ], m := 1, n := 2, q := 3,
  vprime := [ 1, 2, 3, 2, 1 ], z := 8 )
gap> AllSingerAlgebraInfos( "diff", 1, "e", IsPrimeInt );
[ rec( LL := 3, d := fail, dec := 1, diff := 1,
      e := 380808546861411923, isom := [ 862, 3 ], m := 26, n := 43,
      q := 3, vprime := [ 1, 861, 1 ], z := 862 ) ]
gap> OneSingerAlgebraInfo( "z", 8, r -> r.m, 2 );
rec( LL := 5, d := [  ], dec := 0, diff := 0, e := 3,
  isom := [ 8, 5 ], m := 2, n := 2, q := 5,
  vprime := [ 1, [ 3, 2 ], [ 1, 2 ] ], z := 8 )
```

4.2.2 DisplaySingerAlgebras

▷ `DisplaySingerAlgebras(`*cond1, val1, cond2, val2, ...*`)` (function)

Returns: nothing.

This function shows the precomputed data about the Singer algebras of dimension up to 10000 in a table on the screen.

The rows of the table are determined by the arguments, which have the same meaning as in the function `AllSingerAlgebraInfos` (4.2.1). (It is advisable to restrict the contents of the database to a small number of rows.)

The columns of the table correspond to the parameters `z`, `q`, `n`, `LL` (the Loewy length), `m(q,e)`, `diff` (the difference $\lfloor n(q-1)/m(q,e) \rfloor + 1 - LL$), and the information whether the isomorphism type of the algebra with the given parameters is classified or not (an empty string or the flag `-`, respectively).

This function is available only if the Browse package is available.

Example

```
gap> DisplaySingerAlgebras( "z", 7 );
    z |     q |    n |    LL | m(q,e) | diff |  isom
------+-------+------+-------+--------+------+------
    7 |     2 |    3 |     4 |      1 |    0 |     2
    7 |     3 |    6 |     3 |      6 |    0 |     3
    7 |     6 |    2 |     3 |      5 |    0 |     3
    7 |     8 |    1 |     8 |      1 |    0 |     8

gap> DisplaySingerAlgebras( "z", [ 1 .. 200 ], "diff", IsPosInt );
    z |     q |    n |    LL | m(q,e) | diff |  isom
------+-------+------+-------+--------+------+------
   70 |     3 |   12 |     3 |      8 |    1 |     3
   91 |     5 |   12 |     3 |     16 |    1 |     5
   95 |     8 |   12 |     3 |     28 |    1 |     2
  104 |     7 |   12 |     3 |     24 |    1 |     7
  123 |     2 |   20 |     3 |      6 |    1 |     2
  143 |    32 |   12 |     3 |    124 |    1 |     2
  148 |    23 |   12 |     3 |     88 |    1 |     3
  155 |    37 |   12 |     3 |    144 |    1 |     3
  182 |     5 |   12 |     3 |     16 |    1 |     5
  182 |    41 |   12 |     3 |    160 |    1 |     5
  182 |    45 |   12 |     3 |    176 |    1 |     5
  185 |    14 |   12 |     3 |     52 |    1 |     2
  185 |    27 |   12 |     3 |    104 |    1 |     2
  190 |    27 |   12 |     3 |    104 |    1 |     3
  195 |     7 |   12 |     4 |     18 |    1 |     7
```

4.2.3 BrowseSingerAlgebras

▷ `BrowseSingerAlgebras(`*cond1, val1, cond2, val2, ...*`)` (function)

Returns: the list of data selected in visual mode.

This function shows the precomputed data about the Singer algebras of dimension up to 10000 in a Browse table. The columns of the table correspond to the parameters `z`, `q`, `n`, `LL` (the Loewy length), `m(q,e)`, `diff` (the difference $\lfloor n(q-1)/m(q,e) \rfloor + 1 - LL$), and the information whether the

isomorphism type of the algebra with the given parameters is classified or not (an empty string or the flag -, respectively).

This function is available only if the Browse package is available.

Example

```
gap> if IsBound( BrowseSingerAlgebras ) then
>   bsp:= [ NCurses.keys.BACKSPACE ];;  # hit the BACKSPACE key
>   d:= [ NCurses.keys.DOWN ];;  # hit the down arrow
>   l:= [ NCurses.keys.LEFT ];;  # hit the left arrow
>   BrowseData.SetReplay( Concatenation(
>         # select the 'isom?' column
>         "scrrrrrr",
>         # restrict the table to rows with unknown isom. type
>         "f-", [ NCurses.keys.ENTER ],
>         # clear the restriction
>         "!",
>         # select the 'diff' column
>         "scrrrrr",
>         # restrict the table to rows with nonzero 'diff'
>         "f", bsp, "0", d, d, d, d, l, [ NCurses.keys.ENTER ],
>         # clear the restriction
>         "!",
>         # select the first entry with 'z = 100'
>         "sc/100", [ NCurses.keys.ENTER ],
>         # add this entry to the result
>         [ NCurses.keys.ENTER ],
>         # and quit the applications
>         "Q" ) );
>   BrowseSingerAlgebras();;
>   BrowseData.SetReplay( false );
> fi;
```

4.2.4 IdSingerAlgebra

▷ `IdSingerAlgebra(q, z)` (operation)

▷ `IdSingerAlgebra(A)` (attribute)

Returns: a pair of positive integers, or `fail`.

For positive integers q and z, `IdSingerAlgebra` returns either `fail` (if the pair $[q,z]$ belongs to the set of those parameters for which the distribution to isomorphism types is not yet known) or the list $[z,q']$ such that q' is minimal with the property that the Singer algebra $A[q',z]$ is isomorphic with $A[q,z]$.

For a Singer algebra $A = A[q,z]$ of dimension $z+1 \leq 10001$ and with known `ParametersOfSingerAlgebra` (3.1.3) value, `IdSingerAlgebra` returns the value for the arguments q and z.

Example

```
gap> List( [ 2 .. 8 ], q -> IdSingerAlgebra( q, 7 ) );
[ [ 7, 2 ], [ 7, 3 ], [ 7, 2 ], [ 7, 3 ], [ 7, 3 ], fail, [ 7, 8 ] ]
gap> IdSingerAlgebra( 100, 259 );
fail
gap> IdSingerAlgebra( SingerAlgebra( 10, 11 ) );
[ 11, 2 ]
```

4.3 On the Classification of Singer Algebras by Isomorphism Type

Up to now, the algebras $A[q,z]$, for $z \leq 10000$, have not yet been fully classified up to isomorphism type. The following sections show how the current status of this classification can be obtained.

4.3.1 The Datastructure that Describes our Knowledge about the Distribution to Isomorphism Types

We introduce a global variable `KnownDistribution`, a list that stores at position z ($1 \leq z \leq 10000$) the currently known distribution of the relevant prime residues q modulo z (that is, the smallest representatives from cyclic groups of prime residues) into equivalence classes. Each equivalence class describes the smallest union of isomorphism classes of the algebras $A[q,z]$ that is currently known.

We encode each equivalence class by a list $[I_1, I_2, \ldots, I_n]$ where each I_i is a list of values q such that the $A[q,z]$ are known to be isomorphic; the algebras given by values in different sets I_j can be isomorphic or not. If $n = 1$ then the equivalence class is known to describe exactly one isomorphism class.

In the following sections, we will successively refine the underlying equivalence relation. Initially, we define it by equality of the Loewy vector of $A[q,z]$ –isomorphic algebras have the same Loewy vector– such that the class for the Loewy vector v, say, has the form $[[q_1],[q_2],\ldots,[q_n]]$, where $A[q_1,z], A[q_2,z], \ldots, A[q_n,z]$ are exactly the representatives of Singer algebras with Loewy vector v.

Example

```
gap> maxz:= 10000;;
gap> KnownDistribution:= [];;
gap> vectors:= "dummy";;
gap> for z in [ 1 .. maxz ] do
>      allforz:= AllSingerAlgebraInfos( "z", z );
>      vectors:= Set( allforz, r -> MakeImmutable( r.vprime ) );
>      positions:= List( allforz, r -> Position( vectors, r.vprime ) );
>      KnownDistribution[z]:= List( vectors, x -> [] );
>      for i in [ 1 .. Length( positions ) ] do
>        Add( KnownDistribution[z][ positions[i] ],
>             [ allforz[i].q ] );
>      od;
>    od;
```

We provide a small function that prints information about our current knowledge, and call it.

Example

```
gap> ShowDistributionStatus:= function()
>    Print( "#I  min. no. of isom. types: ",
>           Sum( KnownDistribution, Length ), "\n",
>           "#I  max. no. of isom. types: ",
>           Sum( List( KnownDistribution,
>                      l -> Sum( l, Length ) ) ), "\n",
>           "#I  no. of nontriv. classes: ",
>           Sum( KnownDistribution,
>                l -> Number( l, x -> Length( x ) > 1 ) ), "\n",
>           "#I  no. of entries in these classes: ",
>           Length( Flat( Filtered( Concatenation( KnownDistribution ),
>                       x -> Length( x ) > 1 ) ) ), "\n",
>           "#I  no. of dimensions with open questions: ",
```

```
>           Number( KnownDistribution,
>                   l -> Maximum( List( l, Length ) ) > 1 ), "\n" );
>    end;;
gap> ShowDistributionStatus();
#I  min. no. of isom. types: 475581
#I  max. no. of isom. types: 768512
#I  no. of nontriv. classes: 47834
#I  no. of entries in these classes: 340765
#I  no. of dimensions with open questions: 9963
```

4.3.2 Isomorphism of Algebras with Loewy Vector $(1,k,1,\ldots,1)$

By [BHHK20, Prop. 5.2], we know that two Singer algebras with the same Loewy vector of the form $(1,k,1,\ldots,1)$ are isomorphic; in particular, two Singer algebras of the same dimension and Loewy length 3 are isomorphic.

In the data records, a Loewy vector of the form $(1,k,1,\ldots,1)$ appears if and only if the `vprime` component has one of the forms `[ [ 1, z+1 ] ]` or `[ 1, z-1, 1 ]` or `[ 1, k, [ 1, z-k ] ]`.

Example

```
gap> for z in [ 1 .. maxz ] do
>      for i in [ 1 .. Length( KnownDistribution[z] ) ] do
>        C:= KnownDistribution[z][i];
>        vector:= OneSingerAlgebraInfo( "z", z, "q", C[1][1] ).vprime;
>        if Length( vector ) = 1 or
>           ( Length( vector ) = 3 and IsInt( vector[2] ) ) then
>          KnownDistribution[z][i]:= [ Concatenation( C ) ];
>        fi;
>      od;
>    od;
gap> ShowDistributionStatus();
#I  min. no. of isom. types: 475581
#I  max. no. of isom. types: 557645
#I  no. of nontriv. classes: 31852
#I  no. of entries in these classes: 113916
#I  no. of dimensions with open questions: 5714
```

4.3.3 Canonical Isomorphisms of Singer Algebras

We call $A[q,z]$ and $A[q',z]$ *canonically isomorphic* if the map $B(A[q,z])_i \mapsto B(A[q',z])_i$, for $1 \le i \le z+1$, induces an algebra isomorphism $A[q,z] \to A[q',z]$. By [BHHK21, Lemma 7.5], this holds whenever q and q' generate the same group of prime residues modulo z, and the database contains only one representative of each of the equivalence classes defined by this relation. However, it turns out that there are many more canonical isomorphisms.

The algebras $A[q,z]$ and $A[q',z]$ are canonically isomorphic if and only if their multiplication tables w.r.t. the canonical bases (see `SingerAlg.MultTable` (3.3.2)) are equal; equivalently, they are canonically isomorphic if their multiplication tables w.r.t. the canonical bases contain zero in the same places.

The parameters for which canonical isomorphisms occur have been computed and stored in the file `data/joinsCan.json` of the package; the file is in JSON format, and its contents can also be entered into GAP by applying `EvalString` (**Reference: EvalString**) to its contents. We use these

data for refining our equivalence relation. The file encodes a list of pairs; the first entry of each pair is the relevant value of z, and the second is the list of those subsets $\{q_1, q_2, \ldots\}$ such that there are canonical isomorphisms between $A[q_1, z]$, $A[q_2, z]$, ...

Example

```
gap> datadir:= DirectoriesPackageLibrary( "SingerAlg", "data" );;
gap> str:= StringFile( Filename( datadir, "joinsCan.json" ) );;
gap> joins:= EvalString( str );;
gap> L:= "dummy";;
gap> for ll in joins[2] do
>      # a pair of the form [ z, [ [ q1, q2, ... ], [ ... ], ... ] ]
>      z:= ll[1];
>      for i in [ 1 .. Length( KnownDistribution[z] ) ] do
>        L:= KnownDistribution[z][i];
>        for j in [ 1 .. Length( L ) ] do
>          for k in [ 1 .. j-1 ] do
>            if IsBound( L[k] ) and IsBound( L[j] ) and
>               ForAny( ll[2],
>                       l -> IsSubset( l, Set( [ L[j][1], L[k][1] ] ) ) ) then
>              # join the two classes
>              Append( L[k], L[j] );
>              Unbind( L[j] );
>            fi;
>          od;
>        od;
>        KnownDistribution[z][i]:= SortedList( Compacted( L ) );
>      od;
>    od;
gap> ShowDistributionStatus();
#I  min. no. of isom. types: 475581
#I  max. no. of isom. types: 484234
#I  no. of nontriv. classes: 7042
#I  no. of entries in these classes: 19924
#I  no. of dimensions with open questions: 2195
```

Many of the canonical isomorphisms concern algebras $A[q, kn, z]$ and $A[q^k, n, z]$. In fact, two such algebras are isomorphic whenever they have the same Loewy vector and $z \leq 10000$ holds, see Section 4.3.7 for details.

4.3.4 Permutation Isomorphisms of Singer Algebras

We call $A[q,z]$ and $A[q',z]$ *permutation isomorphic* if there is a permutation π of the set $\{1,2,\ldots,z+1\}$ such that the map $B(A[q,z])_i \mapsto B(A[q',z])_{\pi(i)}$, for $1 \leq i \leq z+1$, induces an algebra isomorphism $A[q,z] \to A[q',z]$. In the following, we consider only those permutation isomorphisms that are not canonical, i. e., where π is not the identity.

A necessary condition on π to induce a permutation isomorphism is that the product $B(A[q',z])_{\pi(i)} \cdot B(A[q',z])_{\pi(j)}$ is zero if and only if the product $B(A[q,z])_i \cdot B(A[q,z])_j$ is zero, for all $i, j \in \{1,2,\ldots,z+1\}$. This means that π induces a graph isomorphism between the two simple undirected graphs $\Gamma(B(A[q,z]))$ and $\Gamma(B(A[q',z]))$, where $\Gamma(B(A[q,z]))$ has the vertex set $B(A[q,z])$ and there is an edge between $B(A[q,z])_i$ and $B(A[q,z])_j$ if and only if $B(A[q,z])_i \cdot B(A[q,z])_j$ is nonzero. We use the interface to [McK90] provided by GAP's GraPe package [Soi19] for computing such a graph isomorphism if

it exists, and then check whether it induces a permutation isomorphism of Singer algebras. In order to speed up the computations, we prescribe a partition of the vertices that must be respected by the desired graph isomorphism π; such a partition is given by the property that the numbers of zero entries in the i-th and $\pi(i)$-th row of the multiplication tables w.r.t. $B(A[q,z])$ and $B(A[q',z])$ must be equal.

The parameters for which permutation isomorphisms occur, which are not canonical isomorphisms, have been computed and stored in the file `data/joinsPerm.json` of the package; the file is in JSON format, and its contents can also be entered into GAP by applying `EvalString` (**Reference: EvalString**) to its contents. We use these data for refining our equivalence relation. If we are interested also in explicit permutation isomorphisms then we can use the file `joinsPermExt.json` instead. (Note that the size of this file is about 75 MB.)

Example

```
gap> datadir:= DirectoriesPackageLibrary( "SingerAlg", "data" );;
gap> str:= StringFile( Filename( datadir, "joinsPerm.json" ) );;
gap> joins:= EvalString( str );;
gap> applyjoin:= function( z, q1, q2 )
>      local i, L, j, k;
>      for i in [ 1 .. Length( KnownDistribution[z] ) ] do
>        L:= KnownDistribution[z][i];
>        for j in [ 1 .. Length( L ) ] do
>          for k in [ 1 .. j-1 ] do
>            if IsSubset( Set( [ q1, q2 ] ),
>                         Set( [ L[j][1], L[k][1] ] ) ) then
>              # join the two equivalence classes
>              Append( L[k], L[j] );
>              Unbind( L[j] );
>              KnownDistribution[z][i]:= SortedList( Compacted( L ) );
>              return;
>            fi;
>          od;
>        od;
>      od;
>      # This triple was not used at all.
>      Print( "#E  unnecessary join: ", [ z, q1, q2 ], "\n" );
>    end;;
gap> for l in joins[2] do
>      # 'l' is a triple of the form [ z, q1, q2 ]
>      CallFuncList( applyjoin, l );
>    od;
gap> ShowDistributionStatus();
#I  min. no. of isom. types: 475581
#I  max. no. of isom. types: 481744
#I  no. of nontriv. classes: 5174
#I  no. of entries in these classes: 15608
#I  no. of dimensions with open questions: 1754
```

Only 94 candidate pairs that satisfy the abovementioned necessary criterion for permutation isomorphism do not admit a graph automorphism of the graphs Γ, thus the criterion is quite good.

In all cases where a graph isomorphism is returned, the proposed permutation really induces an algebra isomorphism. This implies: Any two Singer algebras in our list for which we do not know yet whether they are isomorphic are definitely *not* permutation isomorphic.

4.3.5 Combinatorial Invariants Distinguishing Singer Algebras

We know several subspaces of Singer algebras that are *invariant* under algebra isomorphisms. Examples are the members of the radical and socle series, and sums and products of invariant subspaces, see Section 3.3. If we know an invariant subspace such that the dimension is different for two Singer algebras then these algebras are not isomorphic.

In the following, we consider the combinatorial invariants that are used in the function `ConsiderInvariantsByParameters` (3.5.1). The idea is to run over the nontrivial equivalence classes in `KnownDistribution`, and to split these classes whenever we find a distinguishing invariant. (There are cases where more than 500 combinatorial invariants exist. We had stopped the computations after at most 100 of them.)

The parameters for which such splits occur have been computed and stored in the file `data/splitsComb.json` of the package; the file is in JSON format, and its contents can also be entered into GAP by applying `EvalString` (**Reference: EvalString**) to its contents. We use these data for refining our equivalence relation.

Example

```
gap> datadir:= DirectoriesPackageLibrary( "SingerAlg", "data" );;
gap> str:= StringFile( Filename( datadir, "splitsComb.json" ) );;
gap> splits:= EvalString( str );;
gap> applysplit:= function( z, entry, entries, why )
>      local pos, len, elen, i;
>      pos:= Position( KnownDistribution[z], entry );
>      if pos = fail then
>        Print( "#E  did not find <entry> = ", entry,
>               " in KnownDistribution[", z, "]\n" );
>      elif Set( entry ) <> Union( entries ) then
>        Print( "#E  <entry> = ", entry,
>               " does not correspond to <entries> = ", entries, "\n" );
>      else
>        len:= Length( KnownDistribution[z] );
>        elen:= Length( entries ) - 1;
>        for i in [ len, len-1 .. pos+1 ] do
>          KnownDistribution[z][ i+elen ]:= KnownDistribution[z][i];
>        od;
>        KnownDistribution[z]{ [ pos .. pos + elen ] }:= entries;
>      fi;
>    end;;
gap> for l in splits[2] do
>      CallFuncList( applysplit, l );
>    od;
gap> ShowDistributionStatus();
#I  min. no. of isom. types: 479512
#I  max. no. of isom. types: 481744
#I  no. of nontriv. classes: 1934
#I  no. of entries in these classes: 5422
#I  no. of dimensions with open questions: 791
```

4.3.6 Other Invariants Distinguishing Singer Algebras

As soon as invariants are involved that are not combinatorial, in the sense of Section 3.3, computations are expected to get harder.

An example of such a non-combinatorial invariant is the dimension of the matrix Lie algebra of *derivations* (see `Derivations` (**Reference: Derivations**)). This works for low dimensional examples, for example the smallest one from the list in `data/splitsComb.json`, which states that $A[3,40]$ and $A[19,40]$ are not isomorphic.

Example

```
gap> splits[2][1];
[ 40, [ [ 3 ], [ 19 ] ], [ [ [ 3 ] ], [ [ 19 ] ] ], "Roots(0,2,1)" ]
gap> b:= CanonicalBasis( SingerAlgebra( 3, 40, GF(2) ) );;
gap> Dimension( Derivations( b ) );
118
gap> b:= CanonicalBasis( SingerAlgebra( 19, 40, GF(2) ) );;
gap> Dimension( Derivations( b ) );
112
```

This argument distiguishes also $A[23,182]$ and $A[25,182]$, where the dimensions of derivations are 9514 and 9502, respectively. For both $A[11,171]$ and $A[68,171]$, the algebra of derivations has dimension 8048. However, these computations run out of space for larger examples.

Another type of invariant concerns the number of solutions of an equation. A few open questions can be decided by computing the cardinality of the set

$$\{(x,y) \in V \times V; x \cdot y \in U\},$$

where U and V are combinatorial subspaces of the algebra in question, see `SingerAlg.NumberOfProductsInSubspace` (3.5.3).

These cases are collected in the file `data/splitsOther.json`. Computations of this kind are feasible only if the number of indeterminates is small, we call `SingerAlg.NumberOfProductsInSubspace` (3.5.3) with third argument 15 (which is the default value).

Example

```
gap> datadir:= DirectoriesPackageLibrary( "SingerAlg", "data" );;
gap> str:= StringFile( Filename( datadir, "splitsOther.json" ) );;
gap> splits:= EvalString( str );;
gap> for l in splits[2] do
>      CallFuncList( applysplit, l );
>    od;
gap> ShowDistributionStatus();
#I  min. no. of isom. types: 481069
#I  max. no. of isom. types: 481744
#I  no. of nontriv. classes: 649
#I  no. of entries in these classes: 1495
#I  no. of dimensions with open questions: 350
```

Finally, we check whether the currently stored information about the open cases (in the file `data/opencases.json` of the package) and about the distribution to isomorphism types (in the file `data/id.json`) coincides with the above data.

Example

```
gap> str:= Concatenation( "[",
>              SingerAlg.ComputedOpenCases(), "]" );;
gap> if EvalString( str ) <>
```

```
>       SingerAlg.ContentsOfDataFile( "opencases.json" )[2] then
>      Print( "#E  'data/opencases.json' is not up to date." );
>    fi;
gap> str:= Concatenation( "[",
>              SingerAlg.ComputedIdInfoForSingerAlgebras(), "]" );;
gap> if EvalString( str ) <> SingerAlg.IdData[2] then
>      Print( "#E  The stored isomorphism type data are not up to date." );
>    fi;
```

4.3.7 Inspect some of the Canonical Isomorphisms

By [BHHK21, proof of Prop. 3.6], the canonical bases $b_0, b_1, \ldots, b_z$ and $B_0, B_1, \ldots, B_z$ of the algebras $A[q, kn, z]$ and $A[q^k, n, z]$, respectively, have the property that $B_i B_j$ is nonzero whenever $b_i b_j$ is nonzero; the converse is in general not true; for example, consider the case $n = 1$.

However, it turns out that at least under the condition $z \leq 10000$, the converse holds as soon as $A[q, kn, z]$ and $A[q^k, n, z]$ have the same Loewy vector. Note that this implies that $A[q, kn, z]$ and $A[q^k, n, z]$ are then canonically isomorphic.

We can verify this observation as follows. First we collect, for all z, the relevant parameters q, q^k.

Example

```
gap> candidates:= [];;
gap> for z in [ 1 .. 10000 ] do
>      # Fetch the data for this z and sort them by decreasing n.
>      cand:= AllSingerAlgebraInfos( "z", z );
>      SortParallel( List( cand, x -> - x.n ), cand );
>      qs:= List( cand, x -> x.q );
>      # Run over those candidates for which we know
>      # that the Loewy vector determines the isomorphism type.
>      for r in Filtered( cand, r -> Length( r.vprime ) <> 3 ) do
>        q:= r.q;
>        n:= r.n;
>        result:= [ q ];
>        # Find parameters [ Q, m, z ] such that m divides n
>        # and Q is the minimal representative of the subgroup of prime
>        # residues modulo z that is generated by q^(n/m),
>        # and such that the Loewy vector is the same as for r.
>        for d in Difference( DivisorsInt( r.n ), [ 1 ] ) do
>          m:= n/d;
>          Q:= PowerModInt( q, d, z );
>          Q:= Minimum( List( PrimeResidues( m ),
>                             e -> PowerModInt( Q, e, z ) ) );
>          if Q = 1 then
>            Q:= z+1;
>          fi;
>          R:= cand[ Position( qs, Q ) ];
>          if r.vprime = R.vprime then
>            # The Loewy vectors for q and Q are equal.
>            AddSet( result, Q );
>          fi;
>        od;
>        if Length( result ) > 1 then
>          Add( candidates, [ z, result ] );
```

```
>        fi;
>      od;
>    od;
gap> Length( candidates );
22518
```

Then we test whether all these candidates occur in the file that lists the precomputed canonical isomorphisms.

Example

```
gap> datadir:= DirectoriesPackageLibrary( "SingerAlg", "data" );;
gap> str:= StringFile( Filename( datadir, "joinsCan.json" ) );;
gap> joins:= EvalString( str )[2];;
gap> for entry in candidates do
>      z:= entry[1];
>      joinsz:= First( joins, l -> l[1] = z );
>      if not ForAny( joinsz[2], l -> IsSubset( l, entry[2] ) ) then
>        Error( "did not find ", entry, " among the known canon. isom." );
>      fi;
>    od;
```

4.4 Files of the Database of Singer Algebras

The data files are stored in the `data` subdirectory of the package directory. Currently they are all valid JSON (JavaScript Object Notation) texts (see [JSO14]), and they are also valid GAP code. Thus they can be evaluated with `SingerAlg.ContentsOfDataFile` (3.6.3).

Each file contains a GAP list. Its first entry is a list of strings that describes the contents and the format of the remaining entries.

References

[BEKS17] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59:65–98, 2017. 4

[BGH20] T. Breuer, S. Gutsche, and M. Horn. JuliaInterface, GAP interface to Julia, Version 0.4.3. `https://github.com/oscar-system/GAP.jl`, Aug 2020. GAP package. 4, 5, 6

[BHHK20] T. Breuer, L. Héthelyi, E. Horváth, and B. Külshammer. The Loewy structure of certain fixpoint algebras, Part I. *J. Algebra*, 558:199–220, 2020. 4, 6, 9, 10, 11, 15, 16, 18, 36, 40

[BHHK21] T. Breuer, L. Héthelyi, E. Horváth, and B. Külshammer. The Loewy structure of certain fixpoint algebras, Part II. `http://arxiv.org/abs/1912.03065`, 2021. To appear in International Electronic Journal of Algebra. 4, 5, 6, 8, 9, 11, 12, 15, 16, 19, 40, 45

[BL20] T. Breuer and F. Lübeck. Browse, browsing applications and ncurses interface, Version 1.8.11. `http://www.math.rwth-aachen.de/~Browse`, Aug 2020. GAP package. 20

[JSO14] The javascript object notation (json) data interchange format. `http://www.rfc-editor.org/info/rfc7159`, Mar 2014. 46

[Koh19] S. Kohl. FactInt, advanced methods for factoring integers, Version 1.6.3. `https://gap-packages.github.io/FactInt`, Nov 2019. Refereed GAP package. 5

[LN19] F. Lübeck and M. Neunhöffer. GAPDoc, a meta package for GAP documentation, Version 1.6.3. `http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc`, Jul 2019. Refereed GAP package. 21

[McK90] B. D. McKay. *nauty user's guide (version 1.5), Technical report TR-CS-90-02*. Australian National University, Computer Science Department, ANU, 1990. 6, 27, 41

[Slo] N. J. A. Sloane. The on-line encyclopedia of integer sequences. `https://oeis.org`. 34

[Soi19] L. H. Soicher. GRAPE, graph algorithms using permutation groups, Version 4.8.3. `https://gap-packages.github.io/grape`, Dec 2019. Refereed GAP package. 6, 27, 41

Index