The **orders** Package

(Version 1.0)

July 2018

Florian Eisele

Florian Eisele Email: `florian.eisele@city.ac.uk`
Homepage: `https://feisele.github.io`

Copyright

Contents

Chapter 1

Introduction

The orders-package deals with $\mathbb{Z}_p$-orders Λ in semisimple $\mathbb{Q}_p$-algebras A, where $\mathbb{Z}_p$ denotes the p-adic integers. Its original purpose was to compute the projective indecomposable lattices of such orders, and from those a basic algebra of Λ. In addition, the package offers some functionality to deal with arbitrary lattices over Λ. In particular, it can compute representatives for the isomorphism classes of all Λ-lattices inside a given A-module V, compute homomorphism spaces betwen lattices and check whether two Λ-lattices L_1 and L_2 are isomorphic.

1.1 General Design & Limitations of this Package

An order Λ can be created using the function `ZpOrderByMultiMatrices` (2.1.1). The following data needs to be specified:

- A prime p
- A list $[g_1, \ldots, g_k]$ where each g_i is a list of matrices over $\mathbb{Z}$. The list g_i represents a generator of the order. More precisely, $g_i[l]$ is supposed to be the image of the i-th generator under the l-th irreducible $\mathbb{Q}_p$-representation of $\mathbb{Q}_p \otimes \Lambda$.

A few remarks are in order here:

- We assume that we know the Wedderburn decomposition of $\mathbb{Q}_p \otimes \Lambda$. When dealing with group algebras, this means that we need to know the irreducible representations of the group over $\mathbb{Q}_p$, which is a non-trivial problem. For group algebras of finite groups, we might be able to use `IrreducibleRepresentations`, but there is no guarantee that these representations end up being defined over a small enough field. The condition that the images of the generators of Λ are integral means in particular that we single out one particular irreducible lattice as a "standard lattice" in the corresponding simple $\mathbb{Q}_p \otimes \Lambda$-module. For the purposes of this package it does not matter though which lattice we choose.
- While we consider A as a $\mathbb{Q}_p$-algebra, we actually ask for the images of the generators of A under the irreducible representations of A to be matrices over $\mathbb{Q}$. In the case of group algebras of finite groups, it often happens that the irreducible representations over $\mathbb{Q}$ stay irreducible over $\mathbb{Q}_p$, in which case we get generators of the desired form. Integral spinning will the allow us to turn this collection of matrices over the rationals into matrices over the integers.

None of the issues mentioned above arise for $\Lambda = \mathbb{Z}_p S_n$, where S_n denotes the symmetric group on n letters. Therefore symmetric groups are particularly easy amenable to the methods in this package, and we provide some functionality specific to symmetric groups.

Once we have defined an order in the sense of this package, we can define modules over it. This package supports modules that are either Λ-lattices (i. e. free as $\mathbb{Z}_p$-modules) or p-torsion modules (i. e. $\mathbb{F}_p$-vector spaces with a Λ-action). Other torsion types are not supported. Modules are internally represented by the images of the generators $g_1, ..., g_k$ of Λ in some matrix ring. For lattices we also store (if known) an embedding in a direct sum of irreducible lattices.

1.2 Basic Data Structures

This package handles two types of objects: orders and modules over orders. Although you can define modules "by hand", the general idea is that you define an order (which essentially means that you provide a set of irreducible integral representations) and use the functions provided in this package to construct further modules (e. g. simple modules, projectives, radicals of modules) and homomorphism spaces between them.

1.2.1 IsZpOrder

A $\mathbb{Z}_p$-order in a semisimple $\mathbb{Q}_p$-algebra is represented by an oject in the category `IsZpOrder`. You may create such an object using `ZpOrderByMultiMatrices` (2.1.1). Such an object (representing an order Λ) is used to store information about:

- The irreducible representations of $\mathbb{Q}_p \otimes \Lambda$.
- The endomorphism rings of the irreducible lattices (these are crucial for efficient computation). At present, we cannot compute these efficiently in GAP itself, unless the generators of Λ generate a finite group (in which case we can use `CalculateEndomorphismRingsByReynoldsNC` (2.2.3)). If you know that $\mathbb{Q}_p$ is a splitting field for $\mathbb{Q}_p \otimes \Lambda$ you can use `InstallTrivialEndomorphismRings` (2.2.1). If you have a recent version of Magma available, you can also compute these endomorphism rings using `CalculateEndomorphismRingsWithMAGMA` (2.2.4). If you have calculated these endomorphism rings by other means, you can use `InstallEndomorphismRingsNC` (2.2.2) to install them.
- The simple modules and the decomposition matrix of Λ.

When we refer to an "order", we will always mean an object of this type.

1.2.2 IsRModuleOverZpOrder

An object on the category `IsRModuleOverZpOrder` represents a right module over an order. All modules in this package are right modules, so we will usually just call them modules. Modules are represented by a linear representation. The only exception is the zero module. This package supports three types of modules:

- *Lattices*: Lattices are represented by a linear representation over $\mathbb{Z}_p$. For practical purposes the package does expect the images of the generators of the order over which the lattice is defined to lie in $\mathbb{Z}^{n \times n}$ (this constitutes no further theoretical restriction). Lattices have the property

`IsRLatticeOverZpOrder`. Lattices in this package may also store an embedding into a direct sum of irreducible lattices, which is crucial for efficient computation. If lattices are constructed from other lattices using functions from this package, the functions will always compute and store such an embeding for all lattices they create.

- *p-torsion modules*: These are given by the images of the generators of the order in some matrix ring over $\mathbb{F}_p$. p-torsion modules have the property `IsRModuleOverZpOrderModp`.
- *The zero module*: This one has the properties `IsZeroRModuleRep` and `IsZero`.

1.3 Examples

An example session can be found here.

Chapter 2

Provided Functions

2.1 Basics

2.1.1 ZpOrderByMultiMatrices

▷ `ZpOrderByMultiMatrices(`*`p, gens`*`)` (function)

This function creates an order over $\mathbb{Z}_p$ generated by the elements of *`gens`*. *`gens`* is supposed to be a list of lists of matrices with integer entries.

It is highly recommended (and necessary for many functions), that you install endomorphism rings for the irreducible lattices after defining an order. See Section 2.2 for details.

2.1.2 IrreducibleLattices

▷ `IrreducibleLattices(`*`lambda`*`)` (method)

This returns list of irreducible lattices, one for each Wedderburn component. (Note that you have fixed an ordering of the Wedderburn components upon creation of *`lambda`*. This ordering will be used here.)

2.1.3 SimpleModules

▷ `SimpleModules(`*`lambda`*`)` (method)

Returns a list containing the simple modules of the order *`lambda`*. Once called, the ordering of the simple modules will remain fixed.

2.1.4 DecompositionMatrix

▷ `DecompositionMatrix(`*`lambda`*`)` (method)

Return the decomposition matrix of the order *`lambda`*. Rows are sorted as `IrreducibleLattices` (2.1.2), columns as `SimpleModules` (2.1.3).

2.1.5 NameSimpleModulesByDims

▷ `NameSimpleModulesByDims(lambda)` (method)

Gives the simple modules of `lambda` names according to their dimension (constisting of the dimension and some dashes to make the names unique). The names will (for instance) be carried over to a basic algebra by `BasicOrder` (2.4.3).

2.1.6 NameSimpleModules

▷ `NameSimpleModules(lambda, lst)` (method)

`lst` is supposed to be a list of strings of the same length as `SimpleModules(lambda)`. This will assign names to the simple modules of `lambda` accordingly.

2.1.7 SimpleNames

▷ `SimpleNames(lambda)` (method)

Returns a list containing the names of the simple modules of `lambda`. If no names are set, this will produce an error.

2.1.8 NameWedderburnComponentsByDims

▷ `NameWedderburnComponentsByDims(lambda)` (method)

Gives the Wedderburn components of `lambda` names according to their dimension (constisting of the dimension of the corresponding $\mathbb{Q}_p \otimes \Lambda$-module and some dashes to make the names unique). The names will (for instance) be carried over to a basic algebra by `BasicOrder` (2.4.3)..

2.1.9 NameWedderburnComponents

▷ `NameWedderburnComponents(lambda, lst)` (method)

`lst` is supposed to be a list of strings of the same length as `IrreducibleLattices(lambda)`. This will assign names to the Wedderburn components modules of `lambda` accordingly.

2.1.10 ComponentNames

▷ `ComponentNames(lambda)` (method)

Returns a list containing the names of the Wedderburn components of `lambda`. If no names are set, this will produce an error.

2.1.11 DirectSumOfOrders

▷ `DirectSumOfOrders(lst)` (method)

`lst` is supposed to be a list of orders. This will return the direct sum of those orders.

2.1.12 DirectSumOfOrders

▷ `DirectSumOfOrders(lambda1, lambda2)` (method)

As above, for just two orders.

2.1.13 BlocksOfZpOrder

▷ `BlocksOfZpOrder(lambda)` (method)

This method will return a list of lists of integers. Each list of integers corresponds to a block of Λ, the integers indexing the Wedderburn components belonging to that block.

2.1.14 ExtractWedderburnComponents

▷ `ExtractWedderburnComponents(lambda, lst)` (method)

Given a list of integers in `lst` indexing Wedderburn componnents, this method will calculate the projection of Λ onto those components (simply by projecting the generators). Information on simple and ordinary representations (if present) will be carried over to the order that is returned.

2.1.15 RModuleOverZpOrder

▷ `RModuleOverZpOrder(lambda, rep)` (function)

This method is used to create modules from representations. `lambda` is supposed to be an order, and `rep` the images of the generators of `lambda` in either $\mathbb{F}_p^{n\times n}$ or $\mathbb{Z}^{n\times n} \subset \mathbb{Z}_p^{n\times n}$, i. e. `rep` is a list of matrices. This method returns the corresponding module.

2.1.16 ZeroRModule

▷ `ZeroRModule(lambda)` (function)

Returns the zero-module over the order `lambda`.

2.1.17 Dimension

▷ `Dimension(M)` (method)

Returns the dimension of the module `M`.

2.1.18 Generators

▷ `Generators(lambda)` (method)

This returns the generators of `lambda` (i. e. a list of lists of matrices, representing elements in the Wedderburn decomposition).

2.1.19 Rep

▷ `Rep(M)` (method)

M is supposed to be a (non-zero) module defined over an order Λ. This method returns the images of the generators Λ under the representation associated to *M*.

2.1.20 ReduceModP

▷ `ReduceModP(M)` (method)

M is supposed to be a module. This returns M/pM.

2.1.21 SubmoduleByBasisNC

▷ `SubmoduleByBasisNC(M, B)` (method)

Given a module *M* of dimension n and an $n \times n$ matrix *B* whose rowspace is a submodule of *M* (this is not checked), this method will return the corresponding submodule (as a module in the sense of this package).

2.1.22 MaximalSubmoduleBases

▷ `MaximalSubmoduleBases(M[, S])` (method)

If *S* is not specified, this will return a list consisting of tuples `[B,T]` (one for every maximal submodule of *M*), where `B` is a basis matrix for a maximal submodule N of *M*, and `T` is a simple module isomorphic to the quotient M/N.

If *S* is given (a simple module is expected here), this will return a list of basis matrices for all maximal submodules N with $M/N \cong S$.

Whenever the zero module is a maximal submodule, its basis matrix will be given by an empty list.

2.1.23 MaximalSubmoduleBasesMTX

▷ `MaximalSubmoduleBasesMTX(M)` (method)

This returns a list of bases of all maximal submodules of the module *M* using the MeatAxe that comes with GAP. This will only work if the generators of Λ act as automorphisms on *M*. Unsing this function is therefore not recommended, use `MaximalSubmoduleBases` (2.1.22) instead.

2.1.24 MaximalSubmodules

▷ `MaximalSubmodules(M[, S])` (method)

Like `MaximalSubmoduleBases` (2.1.22), but instead of basis matrices, this function will return modules in the sense of this package.

2.1.25 Hom

▷ `Hom(M, N)` (method)

Calculates a basis for the homomorphism space between the modules *M* and *N*. As any module in this package is canonically embedded in either some $\mathbb{Z}_p^{1\times n}$ or some $\mathbb{F}_p^{1\times n}$, this method will always return a list of matrices either defined over the integers or over `GF(p)`.

2.1.26 HomToSimpleNC

▷ `HomToSimpleNC(M, S)` (method)

This is basically equivalent to `Hom` (2.1.25), but it is assumed that *S* is simple. This function uses the built-in MeatAxe of GAP, rendering it much faster than `Hom` (2.1.25).

2.1.27 HomForLattices

▷ `HomForLattices(M, N, opt)` (function)

M and *N* are supposed to be lattices defined over an order `lambda`. *opt* is supposed to be one of `[ ]` (useless), `[ 1 ]` `[ 2 ]` or `[ 1, 2 ]` (although the ordering of the list is irrelevant). This function computes a basis of $\mathrm{Hom}(M,N)$ and will return a list containing the following:

- If $1 \in$ *opt*: A basis of $\mathrm{Hom}(M,N)$, as it would be returned by `Hom` (2.1.25).
- If $2 \in$ *opt*: A basis of $\mathrm{Hom}(M,N)$ embedded in

  $$\bigoplus_i \mathrm{End}_{\mathbb{Q}_p\otimes\Lambda}(\mathbb{Q}_p \otimes L_i)^{m_i\times n_i}$$

  where L_i is the i-th irreducible lattice of the order Λ (over which *M* and *N* are defined), as returned by `IrreducibleLattices` (2.1.2). The elements of $\mathrm{End}_{\mathbb{Q}_p\otimes\Lambda}(\mathbb{Q}_p \otimes L_i)$ are given by a (fixed) representation in some matrix ring over $\mathbb{Q}_p$. m_i resp. n_i denote the multiplicity of $\mathbb{Q}_p \otimes L_i$ as a summand of $\mathbb{Q}_p \otimes M$ resp. $\mathbb{Q}_p \otimes N$.

This function requires that some additional information about the lattices is known, namely an embedding into a direct sum of irreducible lattices, as well as the endomorphism rings of the latter (which is usually the case). If $1 \in$ *opt*, you may check that this information is present via `IsBound(M!.embedding_into_irr_lats)` and `IsBound(M!.order!.Qp_end_bases)`. If $2 \in$ *opt* we additionally require `IsBound(M!.Qp_end_bases_smallmats)`.

2.1.28 RadicalOfModule

▷ `RadicalOfModule(M)` (method)

Returns a tuple `[R, v]`, where `R` is a module isomorphic to $\mathrm{Rad}(M)$. `v` will be a list of non-negative integers, each `v[i]` indicating the multiplicity of `SimpleModules(...)[i]` in $M/\mathrm{Rad}(M)$

2.1.29 TopEpimorphism

▷ `TopEpimorphism(M)` (method)

M is supposed to be a module. This returns a tuple $[Q,\phi,v,h]$, where Q is a module isomorphic to $M/\mathrm{Rad}M$ and $\phi : M \to Q$ is an epimorphism. Furthermore v is a list of the multiplicities of the different simple modules in Q. h is a list of $\mathrm{End}(S)$-bases of $\mathrm{Hom}(M,S)$ (for all simple modules S).

2.1.30 RadicalSeries

▷ `RadicalSeries(M, k)` (method)

Calculates the first *k* radical layers of *M*. Returns a matrix `X` such that `X[j][i]` is equal to the multiplicity of `SimpleModules(...)[i]` in $\mathrm{Rad}^{j-1}(M)/\mathrm{Rad}^{j}(M)$

2.1.31 GramMatrixOfTrace

▷ `GramMatrixOfTrace(lambda, u)` (method)

u is supposed to be a list of rationals, the same numer of entries as *lambda* has Wedderburn components (i. e. *u* represents an element in $Z(\mathbb{Q}_p \otimes \mathit{lambda})$). Return the Gram-matrix of *lambda* with respect to the trace bilinear form $T_u : (a,b) \mapsto \mathrm{tr}(uab)$ and the basis of *lambda* returned by `Generators` (2.1.18).

2.2 Endomorphism Rings of the Irreducible Lattices

Many functions in this package require that the endomorphism rings of the irreducible lattices of an order are known. Unfortunately, at this point we cannot effectively compute these in GAP.

2.2.1 InstallTrivialEndomorphismRings

▷ `InstallTrivialEndomorphismRings(lambda)` (method)

Call this function if you *know* that $\mathbb{Q}_p$ is a splittiong field for $\mathbb{Q}_p \otimes \mathit{lambda}$

2.2.2 InstallEndomorphismRingsNC

▷ `InstallEndomorphismRingsNC(lambda, lst)` (method)

lst is supposed to be a list of bases of endomorphism rings (i. e. a list of lists of matrices). You may for instance call `InstallEndomorphismRingsNC(lambda, List(IrreducibleLattices(lambda), L -> Hom(L,L)))` to compute the endomorphism rings in GAP itself (this will however be horribly slow and excessive in memory usage).

2.2.3 CalculateEndomorphismRingsByReynoldsNC

▷ `CalculateEndomorphismRingsByReynoldsNC(lambda)` (method)

This calculates endomorphism rings using the Reynolds `HomByReynoldsNC` (**??**). This only works if the generators of `lambda` generate a finite group (e.g. if `lambda` is a group algebra).

2.2.4 CalculateEndomorphismRingsWithMAGMA

▷ `CalculateEndomorphismRingsWithMAGMA(`*`lambda`*`)` (method)

This calculates endomorphism rings using MAGMA. Use `SetMAGMAExecutable` (2.2.5) to specify the location of the MAGMA-executable on your system.

2.2.5 SetMAGMAExecutable

▷ `SetMAGMAExecutable(`*`cmd`*`)` (method)

Use `cmd` to run MAGMA.

2.2.6 SetDebugOutput

▷ `SetDebugOutput(`*`b`*`)` (method)

`b` is expected to be a boolean. This turn debugging output on or off (as some computations take a long time, it can be useful to have some indication of how the computation is progressing).

2.3 Condensation for Group Algebras

It is assumed throughout this section that the order Λ we are dealing with is a group algebra (or a block of) $\mathbb{Z}_pG$ for some finite group G. It is furthermore assumed that the generators of Λ are elements of G.

2.3.1 CondensationData

▷ `CondensationData(`*`G, H[, chi]`*`)` (method)

`H` is supposed to be a p'-subgroup of *`G`*. *`chi`* is supposed to be a linear (i. e. one-dimensional) character of *`H`*. If *`chi`* is not given, it will be taken as the trivial character. This method then computes a condensation idempotent e_χ belonging to the pair (H,χ) and a generating system for $e_\chi\mathbb{Z}_pGe_\chi$ (cf. [Noe05]).

2.3.2 CondenseGroupRingNC

▷ `CondenseGroupRingNC(`*`lambda[, gens], data`*`)` (method)

Returns an order that is a condensation of *`lambda`* with respect to the condensation idempotent and system of generators fixed in *`data`*. *`data`* should have been fixed via `CondensationData(...)`. The argument *`gens`* should contain the images of the generators of G (as returned by `GeneratorsOfGroup(G)`) in Λ (i. e., *`gens`* will be a list of lists of matrices). If *`gens`* is not provided, it will be assumed that the images will be the generators of Λ in the same order as they were given upon creation of *`lambda`*.

2.3.3 CondensationProperties

▷ `CondensationProperties(lambda[, gens], data)` (method)

This function is analogous to `CondenseGroupRingNC` (2.3.2), except that it actually just calculates which simple torsion-modules and which irreducible ordinary representations are annihilated by the selected condensation idempotent. You may use this to check wether the condensation will be faithful.

2.3.4 CondenseMatricesNC

▷ `CondenseMatricesNC(Ggens, Greps, data)` (function)

Ggens is supposed to be a list of generators of a group G, *Greps* is supposed to be a list of matrices (defined over the integers) of the same length as *Ggens*. The first to arguments should define a representation of the group in question, i. e., the map sending *Ggens*`[i]` to *Greps*`[i]` extends to a representation of G. *data* is supposed to be a record generated by `CondensationData` (2.3.1). This will return a list `[cgens, base, baseR]`. `cgens` will contain the images of the generators of the condesed algebra (as fixed in *data*) under the condensed representation. `base` will be an embedding of the condensed representation into the uncondensed, and `baseR` will be a right inverse of `base`.

2.3.5 CondenseMatricesWithEvalMapNC

▷ `CondenseMatricesWithEvalMapNC(hom, data)` (function)

Just like `CondenseMatricesNC` (2.3.4), but instead of the arguments `Ggens` and `Greps` you specify a function *hom* which will return for any element in the group its image under the representation you wish to condense.

2.3.6 CondenseTorsionRepNC

▷ `CondenseTorsionRepNC(Ggens, Greps, p, data)` (method)

Just like `CondenseMatricesNC` (2.3.4), but for *p*-modular representations instead of integral ones.

2.4 Projective Modules & Basic Algebras

2.4.1 ProjectiveIndecomposableLattices

▷ `ProjectiveIndecomposableLattices(lambda)` (method)

Calculates the projective indecomposable lattices for *lambda*. It will return a list cof tuples, where the first entry is a projective indecomposable lattice, and the second one a simple torsion module (its head). Depending on the dimensions of the projective indecomposables, running this method may take some time.

2.4.2 BasicOrder

▷ `BasicOrder(lambda)` (method)

This calculates a basic algebra for *lambda*. This method calls `ProjectiveIndecomposableLattices` (2.4.1), and thus may take some time. The generators of the result will form a basis of the order. Names given to the simple modules and Wedderburn components will be carried over accordingly.

2.4.3 BasicOrder

▷ `BasicOrder(lst)` (method)

lst should be a list of projective indecomposable lattices. It will return an order isomorphic to

$$\bigoplus_{P,Q\in \mathtt{lst}} \mathrm{Hom}(P,Q)$$

and carry over names accrodingly. If *lst* is a list of all projective indecomposables, this is just equivalent to `BasicOrder(lambda)`.

2.4.4 ProjectiveIndecomposableForBasicOrder

▷ `ProjectiveIndecomposableForBasicOrder(lambda, k)` (method)

This will construct the projective indecomposable lattive with head `SimpleModules(lambda)[k]`. This method just uses (in contrast to `ProjectiveIndecomposableLattices` (2.4.1)) the regular representation of *lambda*. *lambda* should be an order constructed via `BasicOrder` (2.4.3) (this is not checked though).

2.4.5 ProjectiveIndecomposableLatticesForBasicOrder

▷ `ProjectiveIndecomposableLatticesForBasicOrder(lambda)` (method)

This behaves like `ProjectiveIndecomposableLattices` (2.4.1), but will make use of `ProjectiveIndecomposableForBasicOrder` (2.4.4). In particular, *lambda* should be an order constructed via `BasicOrder` (2.4.3).

2.4.6 GeneratorsForBasicOrder

▷ `GeneratorsForBasicOrder(lambda)` (method)

This will calculate a minimal (w.r.t. inclusion) generating system for *lambda* (that will be preimages of a basis of $\Lambda/\mathrm{Jac}(\Lambda)$ and of a basis of $\mathrm{Jac}(\Lambda)\mathrm{Jac}^2(\Lambda)+p\Lambda$). *lambda* should be an order constructed via `BasicOrder` (2.4.3) (this is not checked though). This method returns a list consisting of an order and a function. The order will in the mathematical sense be equal to *lambda*, but will have the calculated generating set as its set of generators. The function that is returned can be applied to modules defined over *lambda*, and will return the same module defined over the order that was returned.

2.5 Computing with Lattices

2.5.1 AllLattices

▷ `AllLattices(L)` (method)

If `L` is a lattice over an order, this function computes representatives for the isomorphism classes of full sublattices of `L`, or, equivalently, for all lattices in the $\mathbb{Q}_p$-span of `L`.

2.5.2 AllLatticesInd

▷ `AllLatticesInd(L1, L2)` (method)

If `L1` and `L2` are lattices over the same order, this function computes representatives for the isomorphism classes of full sublattices of `DirectSumOfModules(L1,L2)` whose projection down to `L1` and `L2` is all of `L1` and `L2`, respectively.

2.5.3 IsomorphismRModules

▷ `IsomorphismRModules(L1, L2)` (method)

If `L1` and `L2` are lattices over the same order, this function returns an isomorpism between `L1` and `L2` if the two lattices are isomorphic, and `fail` if they are not.

2.5.4 LatticeAlgorithm

▷ `LatticeAlgorithm(M, b)` (function)

M is supposed to be an irreducible lattice (this will not be checked, but if it is not irreducible, this function will not terminate) and *b* should be a boolean. What is returned depends on *b*:

- *b* `= false`: In this case, this function calculates representatives (possibly with repetitions) of all isomorphism classes of irreducible lattices in $\mathbb{Q}_p \otimes M$. It return a list of basis matrices for these as sublattices of *M*.
- *b* `= true`: In this case, this function will return only representatives of lattices with simple top. The result will be a list of tuples, where the first entry is a basis matrix, and the second entry is a simple module (the top of the lattice).

2.5.5 LatticesWithSimpleRadQuo

▷ `LatticesWithSimpleRadQuo(M)` (function)

M is supposed to be an irreducible lattice. This returns representatives (possibly with repititions) of all irreducible lattices with simple top in $\mathbb{Q}_p \otimes M$. It return a list of tuples, where the first entry is a lattice and the second one its top.

2.5.6 GlueUpNC

▷ `GlueUpNC(P, Q, S)` (function)

`P` and `Q` are supposed to be lattices, and `S` is supposed to be a simple module. Under the assumption that `P` and `Q` both have simple top isomorphic to `S` and furthermore `P` and `Q` have no non-zero common $\mathbb{Z}_p$-torsionfree image this method will return a (full) lattice with simple top `S` in $P \oplus Q$. If the assumptions are not met, this method may not terminate.

2.6 Basic Homological Algebra

2.6.1 ProjectiveCover

▷ `ProjectiveCover(M)` (method)

Given the module `M`, this returns a tuple $[P, \phi]$. Here P is the projective cover of `M`, and ϕ is an epimorphism $P \to M$.

2.6.2 LiftHomomorphismNC

▷ `LiftHomomorphismNC(M, phi, N, psi, L)` (method)

`M`, `N` and `L` are supposed to be modules. `phi` is supposed to be a homomorphism $M \to L$. `psi` is supposed to be a homomorphism $N \to L$. This return a homomorphism $X : M \to N$ such that $X \cdot psi = phi$. If no such X exists, `fail` is returned. Note that all homomorphisms are represented by matrices, and it is not checked wether the given matrices actually are homomorphisms.

2.6.3 LiftHomomorphismToDirectSumNC

▷ `LiftHomomorphismToDirectSumNC(lst, svec, phi, N, psi, L)` (method)

`lst` is supposed to be a list of lattices. `svec` is supposed to be a list of indices. This is then equivalent to calling `LiftHomomorphismNC(DirectSumOfModules(List(svec, i -> lst[i])), phi, N, psi, L)` but it is actually a lot faster.

2.6.4 HellerTranslate

▷ `HellerTranslate(M)` (method)

Given a module `M`, this computes its Heller translate. Note that this is always going to return a lattice.

2.6.5 HellerTranslateModular

▷ `HellerTranslateModular(M)` (method)

Given a torsion module *M*, this computes its modular Heller translate. That is, the Heller translate in the module category of $\mathbb{F}_p \otimes \Lambda$, where Λ denotes the order over which *M* is defined. Note that this is always going to return a torsion module.

2.7 Input/Output

2.7.1 PrintAsFunction

▷ `PrintAsFunction(`*X*`)` (method)

Outputs a function that will create the module or order *X*. Usually you would use `SaveAsFunction` (2.7.2) directly.

2.7.2 SaveAsFunction

▷ `SaveAsFunction(`*filename, X*`)` (method)

Saves the module or order *X* to the file *filename* (which will be overwritten in case it exists). You can then read that file using `ReadOrder` (2.7.4) or `ReadModule` (2.7.5).

2.7.3 SaveAsRecord

▷ `SaveAsRecord(`*filename, lambda*`)` (method)

This writes the order *lambda* to the file *filename* (which will be overwritten in case it exists). The file will then contain a GAP-function that returns a record. The content of that file cannot be turned back into an order directly using this package, so usually you would want to use `SaveAsFunction` (2.7.2). However, the advantage of `SaveAsRecord` is that you do not need this package to read the file that it produces.

2.7.4 ReadOrder

▷ `ReadOrder(`*filename*`)` (method)

Returns an order that is identical to the one that was written to *filename* using `SaveAsFunction` (2.7.2).

2.7.5 ReadModule

▷ `ReadModule(`*filename, lambda*`)` (method)

Returns a module that is identical to the one that was written to *filename* using `SaveAsFunction` (2.7.2). That module will be defined over *lambda* (so when you save a module be sure to always save the order over which it is defined as well).

2.8 Symmetric Groups

2.8.1 ZpSn

▷ `ZpSn(p, n)` (function)

This returns the order $\mathbb{Z}_p\Sigma_n$. The Wedderburn components are named by partitions, and sorted descending with respect to the lexicographical ordering. The simple modules are named after their dimensions (it will not be too hard to find the corresponding `p`-regular partitions, though). The lattices returned by `IrreducibleLattices` (2.1.2) will be the Specht lattices.

2.8.2 ZpSnWedderburnComponentsNC

▷ `ZpSnWedderburnComponentsNC(p, part)` (function)

part is supposed to be a non-empty list of pairwise distinct partitions of the same number n (this will however not be checked). Returns the projection of $\mathbb{Z}_p\Sigma_n$ on the Wedderburn components labeled by the partitons in *part*. Otherwise this behaves like `ZpSn` (2.8.1).

2.8.3 NaturalSpechtRepresentation

▷ `NaturalSpechtRepresentation(lambda)` (function)

lambda is supposed to be a partition of some positive integer `n`. This function returns a group homomorphism from `SymmetricGroup(n)` into a general linear group over the integers. To be more precise, this function implements Young's "natural" representation for the Specht module S^λ. See Example 25.2 in [Jam78] for details.

2.8.4 PartitionAsString

▷ `PartitionAsString(mu)` (method)

Given a partition *mu*, this returns a string representing *mu*. For instance `PartitionAsString([3,2,1,1])` returns `"(3,2,1^{2})"`.

2.9 Experimental Functionality

The functions in this section are somewhat experimental and only work reasonably well in small examples. All functions in this section do furthermore require that the generators of the given orders form a basis.

2.9.1 SelfdualSuborders

▷ `SelfdualSuborders(lambda, u)` (method)

Given an element u in $Z(\mathbb{Q}_p \otimes$ *lambda*$)$, this returns all (ismomorphism classes of) orders contained in *lambda* that are selfdual with respect to the trace bilinear form T_u (same as in `GramMatrixOfTrace` (2.1.31)).

2.9.2 AreConjugate

▷ `AreConjugate(`*`lambda, gamma`*`)` (method)

Given two orders `lambda` and *`gamma`* in the same semisimple $\mathbb{Q}_p$-algebra A, this decides wether the two are conjugate (or, equivalently, isomorphic as algebras over the center of A).

2.10 Pretty Output

2.10.1 PrintDecompositionMatrixAsLatex

▷ `PrintDecompositionMatrixAsLatex(`*`A`*`)` (function)

Given an order `A`, this function prints the decomposition matrix of `A` as LaTeX-code (including information on names and dimensions of modules).

2.10.2 PrintBasisOfOrderAsMarkdown

▷ `PrintBasisOfOrderAsMarkdown(`*`A`*`)` (function)

Given a basic order `A`, this function prints a basis of `A` as Jupyter-markdown code (can easily be turned into LaTeX-code).

2.10.3 JupyterDisplayDecompositionMatrix

▷ `JupyterDisplayDecompositionMatrix(`*`A`*`)` (function)
▷ `JupyterDisplayBasisOfOrder(`*`A`*`)` (function)
▷ `JupyterDisplayMultiMatrix(`*`lst`*`)` (function)

If GAP is run in a Jupyter notebook, this will render the respective objects in the notebook.

2.11 Miscellaneous Other Functions

2.11.1 SpinningAlgorithmNC

▷ `SpinningAlgorithmNC(`*`lst`*`)` (function)

Given a list `lst` of rational square matrices that generate a $\mathbb{Z}$-order, this function return a list `ret` of integer square matrices such that $T^{-1} \cdot \texttt{lst}[i] \cdot T = \texttt{ret}[i]\ \forall i$ for an invertible rational matrix T.

2.11.2 UnFlattenMultiMatrixNC

▷ `UnFlattenMultiMatrixNC(`*`lst, dimvec`*`)` (function)

When $A = [A_1, \ldots, A_k]$ is a list of matrices of dimensions *`dimvec`* $= [n_1, \ldots, n_k]$, this method is inverse to `Flat(...)`, i. e. `UnFlattenMultiMatrixNC(Flat(A), dimvec) = A`.

2.11.3 DiagonalJoin

▷ `DiagonalJoin(lst)` (method)

`lst` is supposed to be a list $[A_1,\dots,A_k]$, and the method returns $\mathrm{Diag}(A_1,\dots,A_k)$

2.11.4 DiagonalJoin

▷ `DiagonalJoin(A, B)` (method)

The same as above, for just two matrices.

2.11.5 Valuation

▷ `Valuation(r, p)` (method)

Returns the `p`-valuation of the rational number `r` (the built-in method in GAP only covers integers).

2.11.6 RightInverse

▷ `RightInverse(M)` (method)

Computes a right inverse of the matrix `M` (over `DefaultFieldOfMatrix(M)`). If `M` has no right inverse, it returns `fail`.

References

[Jam78] G. D. James. *The Representation Theory of the Symmetric Group*. Number 682 in Lecture Notes in Mathematics. Springer, 1978. 19

[Noe05] Felix Noeske. *Morita-Äquivalenzen in der algorithmischen Darstellungstheorie*. PhD thesis, RWTH Aachen, 2005. 13