

Introduction to QPA

Part 3

Øystein Skartsæterhagen Øyvind Solberg

Department of Mathematical Sciences
Norwegian University of Science and Technology

Third GAP Days

Outline

1 Homological algebra

2 AR-theory

Original aim

What we are not going to talk about:

- One original aim for QPA: Do projective resolutions using Gröbner basis.

Original aim

What we are not going to talk about:

- One original aim for QPA: Do projective resolutions using Gröbner basis.
- Based on a non-minimal projective resolution - Green-S-Zacharia.
- The resolution is available in QPA,
`ProjectiveResolutionOfPathAlgebraModule(N, 3);`

Original aim

What we are not going to talk about:

- One original aim for QPA: Do projective resolutions using Gröbner basis.
- Based on a non-minimal projective resolution - Green-S-Zacharia.
- The resolution is available in QPA,
`ProjectiveResolutionOfPathAlgebraModule(N, 3);`
- But nothing is developed around it.

Projective cover

Applies to: Only finitely generated modules in QPA.

Algorithm:

- ① Find minimal set of generators
- ② Find maps from indecomposable projective covering the generators
- ③ Sum up the maps

Projective cover

Applies to: Only finitely generated modules in QPA.

Algorithm:

- ① Find minimal set of generators
- ② Find maps from indecomposable projective covering the generators
- ③ Sum up the maps

```
gap> Q:= Quiver(3,[[1,2,"a"],[1,2,"b"],[2,2,"c"],
    [2,3,"d"],[3,1,"e"]]);;
gap> KQ:= PathAlgebra(Rationals, Q);;
gap> AssignGeneratorVariables(KQ);;
gap> A:= KQ/[d*e,c^2,a*c*d-b*d,e*a];;
gap> S := SimpleModules(A)[1];;
gap> f := ProjectiveCover(S);
<<[ 1, 4, 3 ]> ---> <[ 1, 0, 0 ]>>
```

Injective envelope

Injective envelope not implemented directly.

Must use DIY:

```
gap> S := SimpleModules(A)[1];  
gap> Sop := DualOfModule(S);  
<[ 1, 0, 0 ]>  
gap> fop := ProjectiveCover(Sop);  
<<[ 1, 0, 1 ]> ---> <[ 1, 0, 0 ]>>  
gap> f := DualOfModuleHomomorphism(fop);  
<<[ 1, 0, 0 ]> ---> <[ 1, 0, 1 ]>>
```


Syzygies

Finding syzygies, that is, the kernels in a projective resolution, is based on finding kernels of `ProjectiveCovers` and this uses linear algebra.

The command: `NthSyzygy` – two arguments, one module M and a positive integer n .

```
gap> NthSyzygy(S, 3);  
Computing syzygy number: 1 ...  
Dimension vector for syzygy: [ 0, 4, 3 ]  
Top of the 1th syzygy: [ 0, 2, 0 ]  
Computing syzygy number: 2 ...  
Dimension vector for syzygy: [ 0, 0, 1 ]  
Top of the 2th syzygy: [ 0, 0, 1 ]
```

Syzygies

```
Computing syzygy number: 3 ...  
Dimension vector for syzygy: [ 1, 2, 1 ]  
Top of the 3th syzygy: [ 1, 0, 0 ]  
<[ 1, 2, 1 ]>  
gap> NthSyzygy(S,20);  
Computing syzygy number: 1 ...  
Top of the 1th syzygy: [ 0, 2, 0 ]  
.....  
Top of the 3th syzygy: [ 1, 0, 0 ]  
Computing syzygy number: 4 ...  
Dimension vector for syzygy: [ 0, 2, 2 ]  
Top of the 4th syzygy: [ 0, 1, 0 ]  
The module has projective dimension 4.  
<[ 0, 2, 2 ]>
```

Injective and projective dimension

- Again based on using `ProjectiveCover`.
- Injective and projective dimension can be infinite!
- Can only check if they are less or equal to a number in general.
- In some situations, we know that they are finite.
- In general, no algorithm exists.
- In some cases (work in progress): Can prove infinite projective dimension. Using modules over kQ when $A = kQ/I$.

The command `InjDimensionOfModule` and `ProjDimensionOfModule` takes two arguments, one module and one non-negative integer:

Injective and projective dimension

```
gap> ProjDimensionOfModule(S, 3);  
false  
gap> ProjDimensionOfModule(S, 4);  
4  
gap> ProjDimension(S);  
4  
gap> InjDimensionOfModule(S, 1);  
false  
gap> InjDimensionOfModule(S, 2);  
false  
gap> InjDimensionOfModule(S, 3);  
3
```

TODO: Set ProjDimension and InjDimension when computing NthSyzygy.

Global dimension

- Enough to check the projective dimension of the simple modules.
- Global dimension can be infinite!
- Can only check if it is less or equal to a number in general.
- kQ = hereditary \longrightarrow global dimension set in QPA
- kQ/I and Q no oriented cycle \longrightarrow global dimension finite, not implemented.
- kQ/I selfinjective \longrightarrow infinite global dimension, not implemented.

Global dimension

```
gap> A := NakayamaAlgebra(Rationals, [3, 2, 1]);;  
gap> GlobalDimension(A);  
1  
gap> A := NakayamaAlgebra(Rationals, [2]);;  
gap> GlobalDimensionOfAlgebra(A, 3);  
infinity  
gap> A := NakayamaAlgebra(Rationals,  
                           [3, 3, 3, 3, 3, 3, 2, 1]);;  
gap> GlobalDimensionOfAlgebra(A, 7);  
5
```

Known bound for monomial algebras, not implemented.

Pullback and pushout

Given

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ g \downarrow & & \\ C & & \end{array}$$

can construct pushout

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ g \downarrow & & \downarrow g' \\ C & \xrightarrow{f'} & E \end{array}$$

Pullback and pushout

```
gap> A := NakayamaAlgebra(Rationals, [3,2,1]);;
gap> B := IndecProjectiveModules(A)[1];
<[ 1, 1, 1 ]>
gap> f := RadicalOfModuleInclusion(B);
<<[ 0, 1, 1 ]> ---> <[ 1, 1, 1 ]>>
gap> g := TopOfModuleProjection(Source(f));
<<[ 0, 1, 1 ]> ---> <[ 0, 1, 0 ]>>
gap> po := PushOut(f,g);
[ <<[ 0, 1, 0 ]> ---> <[ 1, 1, 0 ]>>,
  <<[ 1, 1, 1 ]> ---> <[ 1, 1, 0 ]>> ]
gap> Range(g) = Source(po[1]);
true
gap> Range(f) = Source(po[2]);
true
```


Pullback and pushout

Given

$$\begin{array}{ccc} & & C \\ & & \downarrow g \\ A & \xrightarrow{f} & B, \end{array}$$

we can construct the pullback

$$\begin{array}{ccc} E & \xrightarrow{f'} & C \\ g' \downarrow & & \downarrow g \\ A & \xrightarrow{f} & B. \end{array}$$

Pullback and pushout

```
gap> g := CoKernelProjection(  
      SocleOfModuleInclusion(C));  
<<[ 1, 1, 1 ]> ---> <[ 1, 1, 0 ]>>  
gap> f := RadicalOfModuleInclusion(Range(g));  
<<[ 0, 1, 0 ]> ---> <[ 1, 1, 0 ]>>  
gap> pb := PullBack(f,g);  
[ <<[ 0, 1, 1 ]> ---> <[ 1, 1, 1 ]>>,  
  <<[ 0, 1, 1 ]> ---> <[ 0, 1, 0 ]>> ]
```

Extensions

Given a projective resolution of a module M

$$\cdots \rightarrow P_2 \xrightarrow{d_2} P_1 \xrightarrow{d_1} P_0 \xrightarrow{d_0} M \rightarrow 0,$$

the extension group $\text{Ext}_\Lambda^1(M, N)$ is the homology of

$$\text{Hom}_\Lambda(P_0, N) \xrightarrow{d_1^*} \text{Hom}_\Lambda(P_1, N) \xrightarrow{d_2^*} \text{Hom}_\Lambda(P_2, N)$$

in the middle term. The kernel of d_2^* can be identified with $\text{Hom}_\Lambda(\Omega_\Lambda^1(M), N)$, so that

$$\text{Ext}_\Lambda^1(M, N) \simeq \text{Hom}_\Lambda(\Omega_\Lambda^1(M), N) / \{\Omega_\Lambda^1(M) \rightarrow P_0 \xrightarrow{\forall f} N\}.$$

Extensions

The function `ExtOverAlgebra` takes two arguments, two modules M and N , and returns a list of three elements:

- ① the inclusion $\Omega_\Lambda^1(M) \rightarrow P_0$,
- ② a basis \mathcal{B} over the ground field k of $\text{Ext}_\Lambda^1(M, N)$ as homomorphisms from $\Omega_\Lambda^1(M) \rightarrow N$, and
- ③ a function $\varphi: \text{Hom}_\Lambda(\Omega_\Lambda^1(M), N) \rightarrow k^{|\mathcal{B}|}$, which computes the coordinates of any element in $\text{Hom}_\Lambda(\Omega_\Lambda^1(M), N)$ as an element in $\text{Ext}_\Lambda^1(M, N)$.

Extensions

```
gap> Q:= Quiver(3,[[1,2,"a"],[1,2,"b"],[2,2,"c"],  
                  [2,3,"d"],[3,1,"e"]]);;  
gap> KQ:= PathAlgebra(Rationals, Q);;  
gap> AssignGeneratorVariables(KQ);;  
#I Assigned the global variables [ v1, v2, v3,  
    a, b, c, d, e ]  
gap> A:= KQ/[d*e,c^2,a*c*d-b*d,e*a];;  
gap> S := SimpleModules(A)[1];;  
gap> M := Kernel(ProjectiveCover(S));;
```

Extensions

```
gap> ext := ExtOverAlgebra(S,M);  
[ <<[ 0, 4, 3 ]> ---> <[ 1, 4, 3 ]>>,  
  [ <<[ 0, 4, 3 ]> ---> <[ 0, 4, 3 ]>>,  
    <<[ 0, 4, 3 ]> ---> <[ 0, 4, 3 ]>>,  
    <<[ 0, 4, 3 ]> ---> <[ 0, 4, 3 ]>>,  
    <<[ 0, 4, 3 ]> ---> <[ 0, 4, 3 ]>>,  
    <<[ 0, 4, 3 ]> ---> <[ 0, 4, 3 ]>> ],  
function( map ) ... end ]  
gap> U := Source(ext[1]);;
```

Extensions

```
gap> homUM := HomOverAlgebra(U,M);  
[ <<[ 0, 4, 3 ]> ----> <[ 0, 4, 3 ]>>,  
  <<[ 0, 4, 3 ]> ----> <[ 0, 4, 3 ]>>,  
  <<[ 0, 4, 3 ]> ----> <[ 0, 4, 3 ]>>,  
  <<[ 0, 4, 3 ]> ----> <[ 0, 4, 3 ]>>,  
  <<[ 0, 4, 3 ]> ----> <[ 0, 4, 3 ]>> ]  
gap> ext[3](homUM[4]);  
[ -1, 0, 0, 1, 0 ]
```

Extensions

Elements in $\text{Ext}_\Lambda^1(M, N) \longleftrightarrow \{f: \Omega_\Lambda^1(M) \rightarrow N\}$

Representation by short exact sequences:

$$\begin{array}{ccccccc}
 0 & \longrightarrow & \Omega_\Lambda^1(M) & \longrightarrow & P_0 & \longrightarrow & M \longrightarrow 0 \\
 & & \downarrow f & & \downarrow & & \parallel \\
 0 & \longrightarrow & N & \longrightarrow & E & \longrightarrow & M \longrightarrow 0
 \end{array}$$

```

gap> pushout := PushOut(ext[1], ext[2][1]);
[ <<[ 0, 4, 3 ]> ---> <[ 1, 4, 3 ]>>,
  <<[ 1, 4, 3 ]> ---> <[ 1, 4, 3 ]>> ]
gap> fprime := pushout[1];
<<[ 0, 4, 3 ]> ---> <[ 1, 4, 3 ]>>
gap> gprime := CoKernelProjection(fprime);
<<[ 1, 4, 3 ]> ---> <[ 1, 0, 0 ]>>

```


Yoneda algebras

Interesting examples

- Group cohomology ring,

$$H^*(G, k) = \bigoplus_{i \geq 0} \operatorname{Ext}_{kG}^i(k, k).$$

- Hochschild cohomology ring,

$$\operatorname{HH}^*(\Lambda) = \bigoplus_{i \geq 0} \operatorname{Ext}_{\Lambda^{\operatorname{op}} \otimes_k \Lambda}^i(\Lambda, \Lambda).$$

- Hopf algebra H , cohomology ring, $\bigoplus_{i \geq 0} \operatorname{Ext}_H^i(k, k)$.
- Λ Koszul algebra, Koszul dual $= \bigoplus_{i \geq 0} \operatorname{Ext}_\Lambda^i(\Lambda_0, \Lambda_0)$.
- In general, $\bigoplus_{i \geq 0} \operatorname{Ext}_\Lambda^i(M, M)$.

Yoneda algebras

The function `ExtAlgebraGenerators`, which takes two arguments, one module and one non-negative integer.

```
gap> ExtAlgebraGenerators(M,10);
[ [ 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
  [ [ <<[ 0, 1, 0 ]> ---> <[ 0, 1, 0 ]>> ],
  [ <<[ 0, 1, 2 ]> ---> <[ 0, 1, 0 ]>> ],
  [ ],
  [ <<[ 0, 3, 3 ]> ---> <[ 0, 1, 0 ]>> ],
  [ ], [ ], [ ], [ ], [ ], [ ], [ ] ] ]
```

Yoneda algebras

The output from this function is a list of three elements, where the

- first element is the dimensions of $\text{Ext}_\Lambda^i(M, M)$ for $i = 0, 1, \dots, n$,
- the second element is the number of minimal generators in the degrees $[0, \dots, n]$,
- the third element is the generators in these degrees.

Yoneda algebras

```
gap> Q := Quiver(1, [[1,1,"a"],[1,1,"b"]]);;
gap> KQ := PathAlgebra(Rationals, Q);;
gap> AssignGeneratorVariables(KQ);;
gap> A := KQ/[a*a,b*b,a*b + 2*b*a];;
gap> ExtAlgebraGenerators(biA, 5);
[ [ 2, 2, 1, 0, 0, 0 ], [ 2, 2, 0, 0, 0, 0 ],
  [
    [ <<[ 4 ]> ---> <[ 4 ]>>,
      <<[ 4 ]> ---> <[ 4 ]>> ],
    [ <<[ 12 ]> ---> <[ 4 ]>>,
      <<[ 12 ]> ---> <[ 4 ]>> ], [ ], [ ],
    [ ], [ ] ] ]
```

AR-theory

Recall that a short exact sequence

$$0 \rightarrow A \xrightarrow{f} B \xrightarrow{g} C \rightarrow 0$$

is *almost split exact* if it is not split exact and

- (i) for any not splittable epimorphism $t: X \rightarrow C$ there is a homomorphism $t': X \rightarrow B$ such that $gt' = t$,
- (ii) for any not splittable monomorphism $s: A \rightarrow Y$ there is a homomorphism $s': A \rightarrow Y$ such that $s'f = s$.

AR-theory

Facts:

- C and A are indecomposable modules.
- $A \simeq D \operatorname{Tr} C$ and $C \simeq \operatorname{Tr} D(A)$.
- For any indecomposable non-projective module C and for any indecomposable non-injective module A , there is an almost split sequence ending in C and starting in A .
- An almost split sequence is a generator of the socle of $\operatorname{Ext}_{\Lambda}^1(C, D \operatorname{Tr}(C))$ as an $\operatorname{End}_{\Lambda}(C)$ -module.

AR-theory

- ① Choose a non-zero element in $\text{Ext}_\Lambda^1(C, D \text{Tr}(C))$ (take a basis vector).
- ② Check if it is annihilated by all elements in the radical of $\text{End}_\Lambda(C)$.
- ③ If not annihilated by the radical of $\text{End}_\Lambda(C)$, multiply with an element in the radical of $\text{End}_\Lambda(C)$ which is not annihilating it. Go to (2). If it is annihilated by the radical of $\text{End}_\Lambda(C)$, it is in the socle of $\text{Ext}_\Lambda^1(C, D \text{Tr}(C))$ and therefore gives the almost split sequence. Jump to (4).
- ④ Done.

In special cases, other algorithms exist, but this is the only implemented in QPA.

AR-theory

The correspondence between the end terms are given by $D \operatorname{Tr}$ and $\operatorname{Tr} D$.

```
gap> A := NakayamaAlgebra(GF(3), [3,2,1]);;
gap> S := SimpleModules(A);;
gap> DTr(S[1]);
<[ 0, 1, 0 ]>
gap> DTr(S[1],2);
Computing step 1...
Computing step 2...
<[ 0, 0, 1 ]>
```


AR-theory

```
gap> DTr(S[1],4);  
Computing step 1...  
Computing step 2...  
Computing step 3...  
Computing step 4...  
<[ 0, 0, 0 ]>  
gap> TrD(DTr(S[1])) = S[1];  
true
```

AR-theory

The function `AlmostSplitSequence` computes the almost split sequence ending in the argument, assuming the argument is an indecomposable module.

Given an almost split sequence:

$$0 \rightarrow A \xrightarrow{(f_i)} \bigoplus_{i=1}^t B_i \xrightarrow{(g_i)} C \rightarrow 0$$

- f_i and g_i are *irreducible homomorphisms*.
- An irreducible homomorphism is either a monomorphism or an epimorphism.
- All irreducible homomorphisms starting in an indecomposable injective module I occur in $I \rightarrow I / \text{soc } I$.
- All irreducible homomorphisms ending in an indecomposable projective P module occur in $\text{rad } P \rightarrow P$.
- The valuation we ignore here.

AR-theory

The AR-quiver:

- Each vertex correspond to an indecomposable module.
- Each arrow correspond to an irreducible homomorphism.
- Each arrow has an valuation (a, b) of pairs of positive integers.

AR-theory

```
gap> IsInjectiveModule(S[1]);  
true  
gap> ass1 := AlmostSplitSequence(S[1]);  
[ <<[ 0, 1, 0 ]> ---> <[ 1, 1, 0 ]>>,  
  <<[ 1, 1, 0 ]> ---> <[ 1, 0, 0 ]>> ]  
gap> I2 := Range(ass1[1]);  
<[ 1, 1, 0 ]>  
gap> IsIndecomposableModule(I2);  
true  
gap> IsInjectiveModule(I2);  
true
```

AR-theory

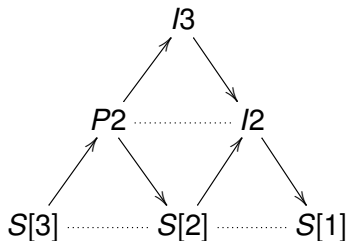
```
gap> ass2 := AlmostSplitSequence(I2);  
[ <<[ 0, 1, 1 ]> ---> <[ 1, 2, 1 ]>>,  
  <<[ 1, 2, 1 ]> ---> <[ 1, 1, 0 ]>> ]  
gap> U := Range(ass2[1]);  
<[ 1, 2, 1 ]>  
gap> IsIndecomposableModule(U);  
false  
gap> decompU := DecomposeModule(U);  
[ <[ 0, 1, 0 ]>, <[ 1, 1, 1 ]> ]  
gap> I3 := decompU[2];  
<[ 1, 1, 1 ]>  
gap> IsInjectiveModule(I3);  
true
```

AR-theory

```
gap> P2 := Source(ass2[1]);  
<[ 0, 1, 1 ]>  
gap> IsProjectiveModule(P2);  
true  
gap> RadicalOfModule(I3) = P2;  
true  
gap> IsProjectiveModule(I3);  
true  
gap> ass3 := AlmostSplitSequence(S[2]);  
[ <<[ 0, 0, 1 ]> ---> <[ 0, 1, 1 ]>>,  
  <<[ 0, 1, 1 ]> ---> <[ 0, 1, 0 ]>> ]  
gap> Range(ass3[1]) = P2;  
true  
gap> IsProjectiveModule(S[3]);  
true
```

AR-theory

From the above calculations we get that the AR-quiver is



where a dotted line means that modules in this mesh forms an almost split sequence.

AR-theory

- Λ an indecomposable finite dimensional algebra.
- A component \mathcal{C} of the AR-quiver is a collection of vertices/indecomposable modules closed under irreducible homomorphisms.
- If there is a component \mathcal{C} where the length of the indecomposable modules in \mathcal{C} is bounded (in particular when it is finite), then Λ is of finite representation type (only finitely many isomorphism classes of indecomposable modules) and \mathcal{C} consists of all isomorphism classes of indecomposable modules.
- Hence we can see from the above calculations that Λ is of finite representation type.

AR-theory

```
gap> IsFiniteTypeAlgebra(A);  
A_3  
Finite type!  
Quiver is a (union of) Dynkin quiver(s).  
true
```