Introduction to QPA

Part 3

Øystein Skartsæterhagen Øyvind Solberg

Department of Mathematical Sciences
Norwegian University of Science and Technology

Third GAP Days

Outline

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 `ProjectiveCover`s 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.

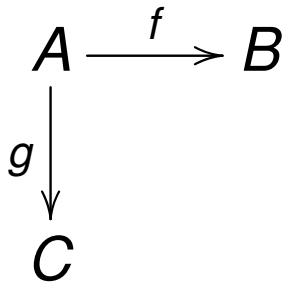
Global dimension

```
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,
              [3,3,3,3,3,3,3,2,1]);;
gap> GlobalDimensionOfAlgebra(A,7);
5
```

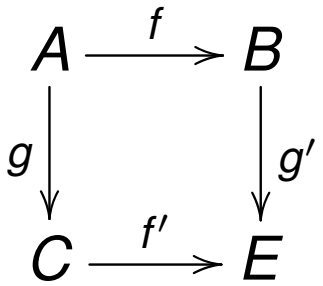
Known bound for monomial algebras, not implemented.

Pullback and pushout

Given



can construct pushout



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

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

we can construct the pullback

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

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

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

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

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

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

$$\mathrm{Ext}^1_\Lambda(M,N) \simeq \mathrm{Hom}_\Lambda(\Omega^1_\Lambda(M),N)/\{\Omega^1_\Lambda(M) \to P_0 \xrightarrow{\forall f} N\}.$$

Extensions

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

1. the inclusion $\Omega^1_\Lambda(M) \to P_0$,
2. a basis $\mathcal{B}$ over the ground field k of $\mathrm{Ext}^1_\Lambda(M,N)$ as homomorhisms from $\Omega^1_\Lambda(M) \to N$, and
3. a function $\varphi\colon\ \mathrm{Hom}_\Lambda(\Omega^1_\Lambda(M),N) \to k^{|\mathcal{B}|}$, which computes the coordinates of any element in $\mathrm{Hom}_\Lambda(\Omega^1_\Lambda(M),N)$ as an element in $\mathrm{Ext}^1_\Lambda(M,N)$.

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 $\mathrm{Ext}^1_\Lambda(M,N) \longleftrightarrow \{f : \Omega^1_\Lambda(M) \to N\}$
Representation by short exact sequences:

$$\begin{array}{ccccccccc} 0 & \longrightarrow & \Omega^1_\Lambda(M) & \longrightarrow & P_0 & \longrightarrow & M & \longrightarrow & 0 \\ & & \downarrow{\scriptstyle f} & & \downarrow & & \| & & \\ 0 & \longrightarrow & N & \longrightarrow & E & \longrightarrow & M & \longrightarrow & 0 \end{array}$$

```
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,

$$H^*(G,k) = \oplus_{i\geqslant 0} \operatorname{Ext}^i_{kG}(k,k).$$

- Hochschild cohomology ring,

$$\operatorname{HH}^*(\Lambda) = \oplus_{i\geqslant 0} \operatorname{Ext}^i_{\Lambda^{\mathrm{op}}\otimes_k\Lambda}(\Lambda,\Lambda).$$

- Hopf algebra H, cohomology ring, $\oplus_{i\geqslant 0} \operatorname{Ext}^i_H(k,k)$.
- Λ Koszul algebra, Koszul dual = $\oplus_{i\geqslant 0} \operatorname{Ext}^i_\Lambda(\Lambda_0,\Lambda_0)$.
- In general, $\oplus_{i\geqslant 0} \operatorname{Ext}^i_\Lambda(M,M)$.

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 $\mathrm{Ext}^i_\Lambda(M, M)$ for $i = 0, 1, \ldots, n$,
- the second element is the number of minimal generators in the degrees $[0, \ldots, n]$,
- 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

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

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

(i) for any not splittable epimorphism $t\colon X \to C$ there is a homomorphism $t'\colon X \to B$ such that $gt' = t$,

(ii) for any not splittable monomorphism $s\colon A \to Y$ there is a homomorphism $s'\colon A \to Y$ such that $s'f = s$.

AR-theory

Facts:

- C and A are indecomposable modules.
- $A \simeq D\,\mathrm{Tr}\,C$ and $C \simeq \mathrm{Tr}\,D(A)$.
- 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 $\mathrm{Ext}^1_\Lambda(C, D\,\mathrm{Tr}(C))$ as an $\mathrm{End}_\Lambda(C)$-module.

AR-theory

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

```
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:

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

- 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 $I \to I/\operatorname{soc} I$.
- All irreducible homomorphisms ending in an indecomposable projective P module occur in $\operatorname{rad} P \to P$.
- 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

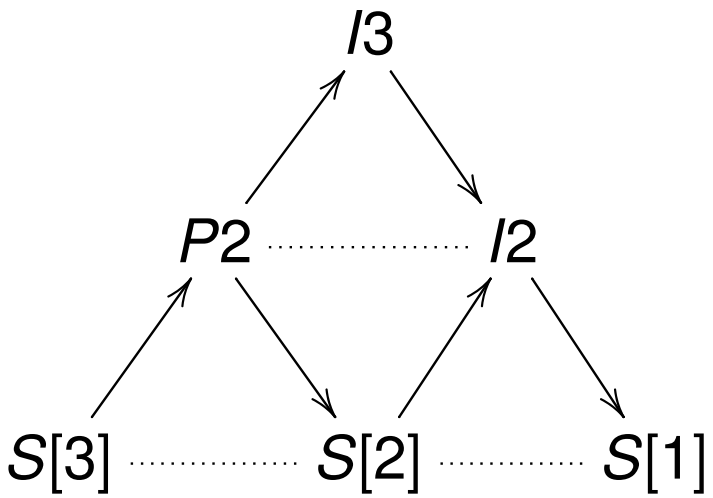
```
gap> ass2 := AlmostSplitSequence(I2);
[ <<[ 0, 1, 1 ]> ---> <[ 1, 2, 1 ]>>,
  <<[ 1, 2, 1 ]> ---> <[ 1, 1, 0 ]>> ]
gap> U := Range(ass2[1]);
<[ 1, 2, 1 ]>
gap> IsIndecomposableModule(U);
false
gap> decompU := DecomposeModule(U);
[ <[ 0, 1, 0 ]>, <[ 1, 1, 1 ]> ]
gap> I3 := decompU[2];
<[ 1, 1, 1 ]>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