1 Running The Code

First, make sure you have downloaded the file `KoszulCohomologyAndKodairaSpencerMap - GAPCode`. It should contain three things, the file `MasterFile.g`, and the folders `CodeFiles` and `ObsoleteCode`. To get the code running, simply ask GAP to read `MasterFile.g`. The terminal should then display that the GAP packages GBNP and QPA have been loaded. The GNBP package is loaded by QPA and mostly used in the background, so we do not have to mind it much. However, QPA is central to all of my code, and so it would be a good idea to familiarize yourself with its basic functions.

2 Creating Algebras

My code uses the QPA package to construct finitely-presented graded algebras. Say we want to construct a k-algebra A with generating set $\{x_1, \ldots, x_n\}$ and (homogeneous) defining relations $\{r_1, \ldots, r_m\}$, where each x_i has degree 1. Mathematically, we would construct A by taking the quotient $k\langle x_1, \ldots, x_n\rangle/R$, where $k\langle x_1, \ldots, x_n\rangle$ is the free k-algebra on n generators and R the ideal generated by the r_i. The process in GAP is similar. First, we construct a free k-algebra `kQ` on n generators by calling the function `FreeKAlgebra(K, n, "x")`. The first input specifies the underlying field, the second the amount of generators, and the final input is a string used to label the generators. For instance, were we to choose k to be the field of rationals and $n = 4$, then in the terminal we would see

```
gap> kQ:= FreeKAlgebra( Rationals, 4, "x" );
#I  Assigned the global variables [ x0, x1, x2, x3, x4 ]
<Rationals[<quiver with 1 vertices and 4 arrows>]>
```

The function `FreeKAlgebra` constructs the free k-algebra `kQ` as a path algebra on the quiver with one vertex and n edges. Note that `FreeKAlgebra` asks for a name for the generators because it calls the QPA function `AssignGeneratorVariables`, which allows us to freely manipulate elements of `kQ` in the terminal, as follows

```
gap> x1*x2 + x1*x4;
(1)*x1*x2+(1)*x1*x4
gap> x1*x2 = x2*1;
false
gap> x1*(2*x2*x4 - 8*x3*x1);
(2)*x1*x2*x4+(-8)*x1*x3*x1
```

The element `x0` denotes the identity element of `kQ`. We can then use these manipulations to define the relations r_i and place them in a list `rels`. For example,

```
gap> r1:= x1*x2 + x2*x1;
(1)*x1*x2+(1)*x2*x1
gap> r2:= x4*x3 - x3*x4;
(-1)*x3*x4+(1)*x4*x3
gap> r3:= x3*x1 + kQ.x1*kQ.x3 ;
(1)*x1*x3+(1)*x3*x1
gap> r4:= x3*x2 + x2*x3 - x1*x4 ;
(-1)*x1*x4+(1)*x2*x3+(1)*x3*x2
gap> r5:= x4*x1 - x1*x4;
(-1)*x1*x4+(1)*x4*x1
gap> r6:= x4*x2 - x1*x3 - x2*x4 ;
(-1)*x1*x3+(-1)*x2*x4+(1)*x4*x2
gap> rels:= [ r1, r2, r3, r4, r5, r6 ];;
```

It is important to note that, inside a function, writing `x1` to get x_1 won't work, and instead we must use `kQ.x1`.

```
gap> kQ.x1*kQ.x2 + kQ.x2*kQ.x1 ;
(1)*x1*x2+(1)*x2*x1
```

Finally, we can get A as a GAP object by calling `A = kQ/rels`. In our example, we get

```
gap> A:= kQ/rels;
<Rationals[<quiver with 1 vertices and 4 arrows>]/
<two-sided ideal in <Rationals[<quiver with 1 vertices and 4 arrows>]>, (6 generators)>>
```

Note that calling `AssignGeneratorVariables(A)` would allow us to freely manipulate the elements of `A` in the terminal, using the same names as for `kQ`. However, doing so will overwrite the variable names for `kQ`.

In some instances, it is possible that the operation `kQ/rels` does not terminate. This happens when the algebra A does not admit a finite Groebner basis. In this case, one must manually terminate the operation, and it is impossible to directly get the algebra A in GAP. Luckily, we can often work around this issue.

Unless there is chance for confusion, we will stop differentiating between the algebra A as a mathematical object and the algebra `A` as an object in GAP. If A does not have a finite Groebner basis, we will simply say that A cannot be represented in GAP. Moreover, we will say that the list `[kQ, rels]` is a GAP-presentation of A, or a presentation of A in GAP.

2.1 The `Algebras.g` file

The file `Algebras.g` in `CodeFiles` contains functions for the families of algebras we have sourced for the project from the literature. These algebras input the chosen base field `K` of the algebra and values, in `K`, for some parameters, with some exceptions; certain families have pre-chosen base fields for convenience. All of these functions output a list of three objects: the first is the quotient algebra `A`, if it admits a finite Groebner basis; otherwise, it is 0. The second object is the free algebra `kQ` used to define A, as explained above, and the third is the defining list of relations of A, as elements of `kQ`. As an example,

```
gap> U:= AlgebraA( Rationals, 1 );
[ 0, <Rationals[<quiver with 1 vertices and 4 arrows>]>,
[ (1)*x1*x2+(-1)*x2*x1, (-1)*x3^2+(-1)*x3*x4+(1)*x4*x3, (-1)*x1*x3+(1)*x3*x1, (-1)*x1*x4+(-1)*x2*x3+(1)*x3*x2,
(-1)*x1*x4+(1)*x4*x1, (1)*x1*x4+(2)*x2*x3+(-1)*x2*x4+(1)*x4*x2 ] ]
gap> V:= AlgebraB( -5 + E(4));
[ <GaussianRationals[<quiver with 1 vertices and 4 arrows>]/
<two-sided ideal in <GaussianRationals[<quiver with 1 vertices and 4 arrows>]>, (6 generators)>>,
<GaussianRationals[<quiver with 1 vertices and 4 arrows>]>,
[ (-E(4))*x1*x2+(1)*x2*x1, (-E(4))*x3*x4+(1)*x4*x3, (5-E(4))*x2*x4+(1)*x3*x1, (5-E(4))*x1*x4+(1)*x3*x2,
(-5+E(4))*x2*x3+(1)*x4*x1, (5-E(4))*x1*x3+(1)*x4*x2 ] ]
```

So `AlgebraA` has a choice of field as an input, but since it does not have a finite Groebner basis its first output is 0. On the other hand, `AlgebraB` has a fixed base field of `GaussianRationals`, but the algebra can be represented in GAP.

2.2 The `ParameterizedAlgebras.g` file

The goal of `ParameterizedAlgebras.g` is to represent the families of algebras themselves in GAP as algebras over rational function fields. Instead of asking for specific values for the parameters of an algebra, the parameters are taken to be the indeterminates of the rational function fields.

If `Foo` is a function in `Algebras.g` for a family of algebras, then there will be a corresponding function `ParameterizedFoo` in `ParametrizedAlgebras.g` which will have one or no inputs: if `Foo` asks for a base field, then so will `ParameterizedFoo`; otherwise, the base field of the algebra generated by `ParameterizedFoo` will be the same as the one generated by `Foo`, and so `ParameterizedFoo` will take in no input. The outputs of functions in `ParametrizedAlgebras.g` have the same form as those in `Algebras.g`.

3 The Hochschild Cohomology of Koszul Algebras

The functions used to compute the Hochschild cohomology groups of a Koszul algebras are found in the files `KoszulComplexByRels.g` and `GradingsForKoszulCohomology.g`. Most of them work in the background, so there is no need to know them, except maybe for the functions in the first two sections of `GradingsForKoszulCohomology.g`.

Suppose $A = kQ/R$ is a Koszul k-algebra, whose Hochschild cohomology $HH^p(A)$ we are interested in. We will first need a GAP-presentation `[kQ, rels]` of A. There are three cases to consider:

• **Case 1**: A can be reprensented in GAP and is infinite-dimensional. In this case, it is impossible for us to compute the full Hochschild cohomology groups $HH^p(A)$. However, the Hochschild cohomology groups inherit a natural grading $HH^p(A) = \bigoplus_i HH^p(A)_i$, and it is possible for us to compute the individual $HH^p(A)_i$. First, we must compute the differentials

$$\mathrm{Hom}_k(W_{n-1}, A)_i \xrightarrow{d_i^n} \mathrm{Hom}_k(W_{n-1}, A)_i \xrightarrow{d_i^{n+1}} \mathrm{Hom}_k(W_{n+1}, A)_i$$

In the terminal, call the function

```
gap> maps:= GradedKoszulCohomologyMapsByRels( A, kQ, rels, p, i );;
```

(It is important to add the two semicolons at the end to stop GAP from printing the output, as it is massive.) Then, `maps` will be a list of three element, the first one being d_i^{n+1}, the second being d_i^n and the third being `A`.

Before explaining how to get GAP to compute $HH^p(A)_i = \ker d^{n+1}/\mathrm{im}\, d_i^n$, we must discuss **The Ghost Error**. To get GAP to compute the kernel of a linear map `f`, we simply `Kernel(f)`. However, it is often the case that asking GAP for the kernel of a map constructed by `GradedKoszulCohomologyMapsByRels` will return a “no method found” error. But, if we exit the break loop and call for the kernel once more, GAP will gladly output the result, as demonstrated below

```
gap> U:= AlgebraB( -5 + E(4));
[ <GaussianRationals[<quiver with 1 vertices and 4 arrows>]/
<two-sided ideal in <GaussianRationals[<quiver with 1 vertices and 4 arrows>]>,
(6 generators)>>, <GaussianRationals[<quiver with 1 vertices and 4 arrows>]>,
[ (-E(4))*x1*x2+(1)*x2*x1, (-E(4))*x3*x4+(1)*x4*x3, (5-E(4))*x2*x4+(1)*x3*x1,
(5-E(4))*x1*x4+(1)*x3*x2, (-5+E(4))*x2*x3+(1)*x4*x1, (5-E(4))*x1*x3+(1)*x4*x2 ] ]
gap> maps:= GradedKoszulCohomologyMapsByRels( U[1], U[2], U[3], 2, 0 );;
gap> Kernel( maps[1] );
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for `*' on 2 arguments at /proc/cygdrive/C/gap-4.11.0/lib/m
ethsel2.g:249 called from
coeff[i] * B[i] at /proc/cygdrive/C/gap-4.11.0/lib/basis.gi:360 called from
LinearCombination( row, mapi[1] ) at /proc/cygdrive/C/gap-4.11.0/lib/vspchom.gi:374 called fr
om
func( C[i] ) at /proc/cygdrive/C/gap-4.11.0/lib/coll.gi:665 called from
List( ech.coeffs, function ( row )
return LinearCombination( row, mapi[1] );
end ) at /proc/cygdrive/C/gap-4.11.0/lib/vspchom.gi:374 called from
MakePreImagesInfoLinearGeneralMappingByImages( map
); at /proc/cygdrive/C/gap-4.11.0/lib/vspchom.gi:436 called from
...  at *stdin*:71
type 'quit;' to quit to outer loop
brk> quit;
gap> Kernel( maps[1] );
<vector space over GaussianRationals, with 15 generators>
```

Subsequently asking GAP for the kernel will cause no errors. At moment of writing, I do not know what causes this mysterious error, nor how to stop it.

Now, to compute $HH^p(A)_i$, you need only call `Kernel( maps[1] )/Image( maps[2] )`. However, due to The Ghost Error, you will need to exit the break loop and enter the command a second time in order to get the cohomology group. Due to the mysteries of GAP's internal machinery, doing this will return $HH^p(A)_i$ as a row space, which is not very useful, and so computing the full quotient is often not worth the effort. Additionally, we are most often interested in the dimension of the cohomology, so we may call `MapsDim( maps )` (again, twice) to get the dimension of $HH^p(A)_i$ without having to compute the quotient. If we are interested in the projection map $\ker d_i^{n+1} \to HH^p(A)_i$, we can obtain it through the function `MapsProj( maps )`.

• **Case 2**: A can be reprensented in GAP and is finite-dimensional. In this case, it is possible for us to compute the entire cohomology group $HH^p(A)$ by following the exact steps above, but replacing the integer `i` with the string `"full"`. Note that the Ghost Error still applies here.

• **Case 3**: A cannot be reprensented in GAP. In this case, we can instead consider the quadratic dual $A^!$ of A, which more often than not can be represented in GAP and is finite-dimensional. Calling `QuadraticDualByRels( kQ, rels )` in the terminal will output a GAP-presentation `[Abang, kP, perprels]` of $A^!$, exactly as the algebras in `Algebras.g`. Now, as explained in Berger, Lambre and Solotar's paper *Koszul Calculus*, there exists a cohomology $\tilde{HH}$ such that $HH^p(A)_i \cong \tilde{HH}^{p+i}(A^!)_{-i}$. We have code for this cohomology: you can compute $\tilde{HH}^p(A^!)_i$ via

```
gap> maps:= TildeGradedKoszulCohomologyMapsByRels( kP, perprels, p, i);
```

for `[kP, perprels]` a GAP-presentation of $A^!$. Note that **this function works only when $A^!$ is finite-dimensional**. This function works exactly as the non-tilde version: it outputs a list containing the two differentials and the algebra itself, and to get the dimension of the cohomology group, we must call `MapsDim( maps )`. Again, as with the non-tilde version, we can replace `i` with the string `"full"` in order to get the full cohomology group $\tilde{HH}^p(A^!)$.
If we are too lazy to call `QuadraticDualByRels` separately, we can do

```
gap> maps:= GradedKoszulCohomologyMapsByRelsViaTilde( kQ, rels, p, i );
```

which automatically computes the Koszul dual of A, and returns the differentials for $\tilde{HH}^{p+i}(A^!)_{-i}$.

Finally, in the off chance that $A^!$ is infinite-dimensional, we can use `TildeGradedKoszulCohomologyMapsByRels2` to get the differentials for $\tilde{HH}^p(A^!)_i$. However, I have not had much time to work on and double-check this code, so it may be unstable. As for its use, it takes in exactly the same inputs as `TildeGradedKoszulCohomologyMapsByRels`.

As hinted at by the names of our functions, the code does not compute the Hochschild cohomology of the algebras themselves, but instead the Koszul cohomology of the algebras. Luckily for us, the complex used for the Koszul cohomology is much smaller than the bar complex, which is too large to be practical, and importantly, the cohomology groups agree for Koszul algebras. More can be read about Koszul cohomology in the paper *Koszul Calculus*, which will be included in the files for the code.

4 The Kodaira-Spencer map

Due to the Ghost Error, computing the Kodaira-Spencer map of a family of algebras is a bit more complex than it should be, but not by much. Suppose we have a family $A(t_1, \ldots, t_m)$ of k-algebras, parameterized by variables $t_1, \ldots, t_m$. We can model this family as an algebra A over the rational function field $k(t_1, \ldots, t_m)$. Finally, suppose further that we want to compute the Kodaira-Spencer map of A at the point $t_1 = a_1 \in k, \ldots, t_m = a_m \in k$. First, start by forming a GAP-presentation `[PolyQ, PolyRels]` of A, where `PolyQ` is a free algebra over the field $k(t_1, \ldots, t_m)$. Then, create a list `pts` of length m in GAP, with the i^{th} element equaling the chosen value a_i. The order of the a_i in `pts` should correspond to the order of the t_i in `IndeterminatesOfFunctionField(PolyK)`, where `PolyK` is the function field $k(t_1, \ldots, t_m)$ in GAP. There are then three steps:

• (1) Call `L:= KSMapSetup1( PolyQ, PolyRels, pts);;`. The double semicolons are important, as the output is very large. This is a setup map, its output is a list of eight objects used in the next functions, none of which you need to manipulate directly.

• (2) Call `LL:= KSMapSetup2( PolyQ, PolyRels, pts);;`. This function involves computing the kernel of a differential, and due to the Ghost Error will probably give an error. Simply exiting the break loop and calling the function again solves the issue.

• (3) Call `f:= KSMap( LL );;`. Then, `f` will be the Kodaira-Spencer map of A at the chosen values a_i.

Some very important things to note:

1. For this to work, the algebra defined from $A(t_1, \dots, t_m)$ when taking $t_i = a_i$ must be representable in GAP, as we are computing its Hochschild cohomology.
2. You must choose your values a_i carefully. I could not find a way to compute in GAP the tangent space of the family at the point, so instead I had to model it as simply a vector space spanned by the indeterminates t_i. This means that the code works on the massive assumption that there is no singularity at the points a_i, so that the dimension of the tangent space doesn't jump.

5 Obsolete Code and Useless files

5.1 Obsolete Code

The code I have written contains the folder `Obsolete Code`. This is very old and very ugly code I wrote for the Hochschild cohomology via the bar complex at the very beginning of the project. None of the files in `Obsolete Code` are loaded when running the master file. However, I have included them in the off chance something written in them finds some use.

5.2 Useless Files

The folder `Useless Files` contains two files: `KoszulCupProductAndBracket.g` and `WorkInProgress.g`. The former serves as a way to construct in GAP the Koszul cup bracket, as defined in *Koszul Calculus*. To define the Koszul Cup bracket $[\,,] : HH^p(A)_0 \otimes HH^q(A)_0 \to HH^{p+q}(A)_0$ of an algebra A with GAP-presentation `[A, kQ, rels]`, simply write

```
gap> L:= KoszulCupBracketSetup1( A, kQ, rels, p, q );;
gap> LL:= KoszulCupBracketSetup2( L );;
gap> bra:= KoszulCupBracket( LL );;
```

This has been briefly glossed over, but you will have instances of the Ghost Error happening, and therefore you will have to call the second setup function multiple times before it outputs correctly. We have rushed over this section because if an algebra is Koszul, then the Koszul bracket is zero everywhere, which is why this code has landed in the `Useless Files` folder.

The latter file, `WorkInProgress.g`, contains the code I was more recently working on, namely an attempt of using the Homotopy Transfer Theorem and the quasi-isomorphism between the bar and Koszul complexes to define the Gerstenhaber bracket on the Koszul complex (only the last section of the file is relevant). The code is incomplete, obviously, with the last few functions having as goal to extend a degree 0 linear map $f : W_2 \to A$ to a degree 0 linear map $\tilde{f} : (A^{\otimes 2})_2 \oplus (A^{\otimes 2})_3 \to A$. This last section is surprisingly well-commented, although it is not explained why it is these functions we are creating. I have included the file in case someone is crazy enough to try and complete my code.

6 Final Notes

This code is the result of over a year and a half of work and research, written by a student who before this could barely program in Python. It is a product of love, anger, hard work and guess work, and as such is held together with string and masking tape. I might have also inadvertently forgotten to explain some parts of the code, some errors I've had, some specific steps to take, etc. If you have any questions, comments, criticisms or insults you want to send my way, you can email me at `laroche.felix1@gmail.com`.