Finitely Presented Modules over the Steenrod Algebra in Sage

by

Michael J. Catanzaro

AN ESSAY

Submitted to the College of Liberal Arts and Sciences,
Wayne State Uiversity,
Detroit, Michigan,
in partial fulfillment of the requirements
for the degree of

MASTER OF ARTS

December of 2011

MAJOR: Mathematics

APPROVED BY: ______________________________

Adviser Date

____

2^{nd} Reader Date
(if necessary)

Acknowledgements

I'm indebted to many people for their support throughout the completion of this essay. I would first like to thank my family. Without the support of my parents and sisters (including the endless lunches), I could not have completed this essay. I would also like to thank Jessica Selweski for all of her helpful ideas, advice, and encouragement.

I'd like to thank my MRC friends for their moral support and positive attitudes, especially Carolyn, Clayton, Nick, Luis, Sara, Sharmin, (serious) Mike, Araz, Emmett, Chad and Heather. I'm especially grateful to Sean Tilson for countless discussions and helpful criticisms. I thank Steve Reddmann for sharing his expertise in programming and many great ideas.

My most sincere thanks are to my adviser Robert Bruner. His guidance, not only as a mathematician but as a mentor, has forever changed the person I am. In no way would this work have been possible without his thoughtful insight and dedication.

Finally, I'd like to dedicate this essay to my dear friend Jim Veneri. His enthusiasm and passion for learning inspired me to never give up on my dreams in mathematics and physics. Thank you Jimbo.

Contents

1 Introduction

In this essay, we outline the functionality and algorithms of the software package `FPMods` written for Sage. The package is based on three main object classes, FP_Module, FP_Element, and FP_Hom, their methods and attributes. From these classes, we've defined more complex functions, allowing the user to perform higher level calculations. Altogether, our current package provides the user with computational tools for attacking problems in algebraic topology in a language and notation closer to mathematical notation than computer language.

Each class has its own section in which we discuss the algorithms associated to it. These sections include a Specification subsection, in which we give a complete list of all attributes and methods defined. The practical user will find this list very useful for working with such an object. The remaining subsections discuss the algorithms of the actual implementation, as well as mathematical proofs and justification. Someone wishing to work with the actual code and expand this work is encouraged to read these subsequent subsections.

1.1 Motivation

Our primary objects of study throughout this essay are finitely presented modules over the Steenrod Algebra. Each Steenrod Algebra (indexed over primes p) is the algebra of all stable mod p cohomology operations, so that the cohomology of any space (or spectrum) is a module over the Steenrod Algebra.

Calculations in modules over the Steenrod Algebra are useful in algebraic topology but can become quite tedious. Automating these runs up against the fact that the Steenrod Algebra is not Noetherian, so that most calculations will be infinite and therefore pose problems for computer implementation. Past solutions have involved simply calculating low degree terms. Here, we solve the problem by restricting attention to finitely presented modules, so that calculations become finite. This is possible because the Steenrod Algebra is the union of finite sub-algebras. One of the tasks we must accomplish is to find convenient sub-algebras over which to work.

We stress that the modules we define are infinite. Finitely presented modules over a sub-Hopf algebra of $\mathcal{A}$ have a finite presentation, but once tensored back up to $\mathcal{A}$-modules, they are infinite. Figure 1 illuminates the complexity of modules tensored up from $\mathcal{A}(1)$ as compared to $\mathcal{A}(2)$. This complexity is only compounded as when we tensor to $\mathcal{A}$, where our modules naturally lie. Even though our package defines modules over some sub-Hopf algebra of $\mathcal{A}$, we emphasize that our results are valid over all of $\mathcal{A}$. The module of Figure 1 is the first szygy of $\mathcal{A}/\mathcal{A}(\mathrm{Sq}^2, \mathrm{Sq}^2\mathrm{Sq}^1)$. In Section 2, this module is defined as M. We use this example to illustrate the role of the profile used in Section 5.3.

1.2 Sage

Our package is written to be part of the mathematics program Sage. Sage is a free, open-source mathematics software system designed to be an alternative to, and interface between, Magma, Maple, Mathematica, GAP, Pari, etc. Sage can be used to study elementary and advanced, pure and applied mathematics.

The primary computing language used in Sage is Python. Python is a dynamic programming language, so computations are done on the fly with an interactive command line. This contrasts to other languages like C, C++, or Fortran. Additionally, Python has very clear and readable syntax, which makes our code more understandable to the non-computer scientist.

A key feature of Sage is the Steenrod Algebra package, written by John Palmieri. This allows us to work with sub-algebras (and their profile functions), the Adem relations, the Milnor and admissible bases, and many other already defined programs and properties. This is one of the main reasons for choosing to implement this in Sage.

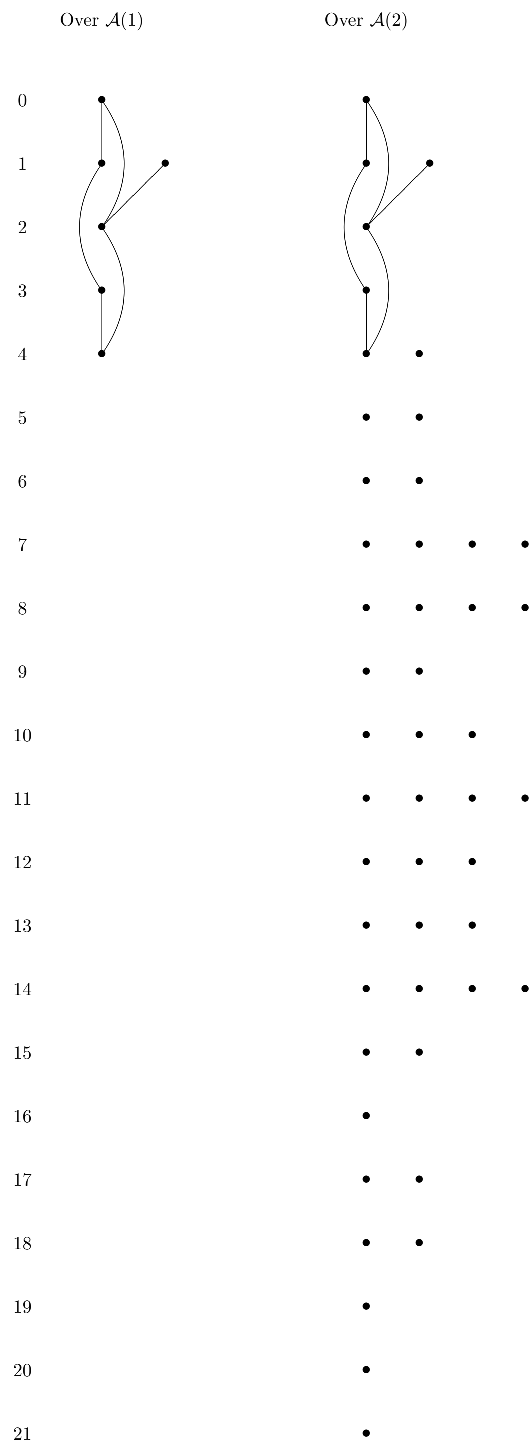

Figure 1: $\Sigma^{-2}\Omega\left(A/A(\mathrm{Sq}^2, \mathrm{Sq}^2\mathrm{Sq}^1)\right)$ as a module over $A = \mathcal{A}(1)$ and $\mathcal{A}(2)$

2 Sample Session

In this section, we show numerous examples of the use of this package. The Traceback portion of error messages has been suppresed for readability. The command prompt `sage:` is not a part of the command.

```
sage: load fpmods.py
sage: M = FP_Module([0,1],[[Sq(2),Sq(1)],[0,Sq(2)],[Sq(3),0]])
sage: x = M([1,0])
sage: x == M.gen(0)
True
sage: y = M.gen(1)
sage: z = x*Sq(2)*Sq(1)*Sq(2); z
[Sq(2,1), 0]
sage: z.nf()
[0, 0]
sage: y*Sq(2)*Sq(1)
[0, Sq(3)]
sage: (y*Sq(2)*Sq(1)).nf()
[0, 0]
sage: T = FP_Module([0,2,4],[[Sq(4),0,1],[Sq(1,1),Sq(2),1]])
sage: z = T.gen(0)*Sq(2) + T.gen(1); z
[Sq(2), 1, 0]
sage: TT,g,h = T.min_pres()
sage: TT.degs
[0, 2]
sage: TT.rels
[[Sq(1,1) + Sq(4), Sq(2)]]
sage: MM,g,h = M.min_pres()
sage: MM.rels
[[Sq(2), Sq(1)], [0, Sq(2)]]
sage: S,incl,proj = MM.submodule([y])
sage: S.degs
[1]
sage: S.rels
```

```
[[Sq(2)]]
sage: for i in range(7):
....:       print "basis for MM in dimension ", i, ": ", MM[i]
basis for MM in dimension  0 :  [[1, 0]]
basis for MM in dimension  1 :  [[Sq(1), 0], [0, 1]]
basis for MM in dimension  2 :  [[Sq(2), 0]]
basis for MM in dimension  3 :  [[Sq(0,1), 0]]
basis for MM in dimension  4 :  [[Sq(1,1), 0]]
basis for MM in dimension  5 :  []
basis for MM in dimension  6 :  []
sage: J = FP_Module([])
sage: J.conn()
+Infinity
sage: J.alg()
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile
    function []
sage: N = FP_Module([1],[[Sq(1)]])
sage: Z = N.suspension(4);Z.degs
[5]
sage: h = FP_Hom(N,M,[x*Sq(1)])
sage: g = FP_Hom(M,N,[0,N.gen(0)])
ValueError: Relation [0, Sq(2)] is not sent to 0
sage: K,i = h.kernel()
sage: K.degs
[6]
sage: i.values
[[Sq(2,1)]]
sage: C,p = h.cokernel()
sage: C.degs
[0, 1]
sage: C.rels
[[Sq(2), Sq(1)], [0, Sq(2)], [Sq(3), 0], [Sq(1), 0]]
sage: CC,pp = h.cokernel('minimal')
sage: CC.rels
[[Sq(1), 0], [Sq(2), Sq(1)], [0, Sq(2)]]
```

```
sage: I,e,m = h.image()
sage: I.degs
[1]
sage: I.rels
[[Sq(1)], [Sq(2,1)]]
sage: Hko = FP_Module([0],[[Sq(1)],[Sq(2)]])
sage: R = Hko.resolution(5,verbose=true)
sage: is_complex(R)
True
sage: is_exact(R)
True
sage: for i in range(6):
....:     print "Stage ", i, "\nDegrees: ", R[i].domain.degs, "\nValues of R[i]: ",R[i].values
Stage 0
Degrees: [0]
Values of R[i]: [[1]]
Stage 1
Degrees: [1, 2]
Values of R[i]: [[Sq(1)], [Sq(2)]]
Stage 2
Degrees: [2, 4]
Values of R[i]: [[Sq(1), 0], [Sq(0,1), Sq(2)]]
Stage 3
Degrees: [3, 7]
Values of R[i]: [[Sq(1), 0], [Sq(2,1), Sq(3)]]
Stage 4
Degrees: [4, 8, 12]
Values of R[i]: [[Sq(1), 0], [Sq(2,1), Sq(1)], [0, Sq(2,1)]]
Stage 5
Degrees: [5, 9, 13, 14]
Values of R[i]: [[Sq(1), 0, 0], [Sq(2,1), Sq(1), 0], [0, Sq(2,1), Sq(1)], [0, 0, Sq(2)]]
```

3 The Steenrod Algebra

The Steenrod Algebra, $\mathcal{A}$, is defined to be the algebra of stable cohomology operations [2, 8]. (This is a very brief description of $\mathcal{A}$ and is only meant to explain ideas which we'll explicitly use. A thorough description would be an essay by itself, and the interested reader should see the previously mentioned sources.) From an algebraic viewpoint, $\mathcal{A}$ is a graded algebra over $\mathbb{F}_p$. When $p = 2$, its generated by the *Steenrod squares*, Sq^i for each $i \geq 0$. For p odd, $\mathcal{A}$ is generated by the *Bockstein* β and the *Steenrod reduced* p^{th} *powers*, $\mathcal{P}^i$ for each $i \geq 0$. These are natural transformations in the cohomology of any toplogical space X.

$$\mathrm{Sq}^i : H^n(X; \mathbb{F}_2) \to H^{n+i}(X; \mathbb{F}_2) \tag{1}$$

$$\mathcal{P}^i : H^n(X; \mathbb{F}_p) \to H^{n+2i(p-1)}(X; \mathbb{F}_p) \tag{2}$$

The product structure of the Steenrod Algebra is given by composition. The unit is Sq^0 for $p = 2$, and $\mathcal{P}^0$ otherwise. These generators are subject to the Adem relations,

$$\mathrm{Sq}^a \mathrm{Sq}^b = \sum_{i=0}^{[a/2]} \binom{b-c-1}{a-2c} \mathrm{Sq}^{a+b-c} \mathrm{Sq}^c \qquad \text{if } a < 2b, \tag{3}$$

and all relations are implied by these. The situation is similar at odd primes. These allow one to write any Steenrod operation uniquely as a linear combination of admissable operations $\mathrm{Sq}^{a_1} \mathrm{Sq}^{a_2} \cdots \mathrm{Sq}^{a_i}$, where admissible means each $a_i \geq 2a_{i+1}$. This basis consisting of the admissable monomials modulo the Adem relations is known as the admissible or Serre-Cartan basis for $\mathcal{A}$.

The Steenrod Algebra $\mathcal{A}$ admits a comultiplication, as well as a multiplication. This coproduct $\psi : \mathcal{A} \longrightarrow \mathcal{A} \otimes \mathcal{A}$,

$$\psi(\mathrm{Sq}^k) = \sum_i \mathrm{Sq}^i \otimes \mathrm{Sq}^{k-i} \tag{4}$$

$$\psi(\mathcal{P}^k) = \sum_i \mathcal{P}^i \otimes \mathcal{P}^{k-i} \tag{5}$$

$$\psi(\beta) = 1 \otimes \beta + \beta \otimes 1 \tag{6}$$

gives $\mathcal{A}$ the structure of a Hopf algebra. It is induced by the Cartan formula, and is much easier to describe than the product. In particular, it is cocommutative. Furthermore, Milnor showed that the dual of $\mathcal{A}$, denoted $\mathcal{A}_*$, is a polynomial algebra, given by

$$\mathcal{A}_* = \begin{cases} \mathbb{F}_2[\xi_1, \xi_2, \ldots] & \text{if } p = 2, \\ \mathbb{F}_p[\xi_1, \xi_2, \ldots] \otimes E[\tau_0, \tau_1, \ldots] & \text{if } p \text{ is odd.} \end{cases}$$

Over $\mathbb{F}_2$, the degree of each ξ_i is $2^i - 1$. For odd characteristic, the degree of ξ_i is $2(p^i - 1)$ and the degree of τ_i is $2p^i - 1$. Monomials in the ξ_i (and at odd primes, the τ_i) form a basis for $\mathcal{A}_*$. Hence, their duals form a basis called the Milnor basis for $\mathcal{A}$.

A typical element in the Milnor basis is written Sq^R, where R is a multi-index. The multi-index notation $R = (i_1, \ldots, i_n)$ is useful, but has no direct relation to the Serre-Cartan basis. In particular, $\mathrm{Sq}^{(i_1,\ldots,i_n)}$ in the Milnor basis is not the same as $\mathrm{Sq}^{i_1} \cdots \mathrm{Sq}^{i_n}$ in the Serre-Cartan basis. In general, they don't even have the same degree. However, $\mathrm{Sq}^{(r)} = \mathrm{Sq}^r$.

One area in which the Milnor basis simplifies our task is describing the sub-Hopf algebras of $\mathcal{A}$. The key to making the calculations finite is to work over these sub-Hopf algebras. In Sec 3.1, we discuss the Classification theorem of quotient Hopf algebras of $\mathcal{A}_*$, i.e. sub-Hopf algebras of $\mathcal{A}$. The Classification theorem completely characterizes all quotient Hopf algebras of $\mathcal{A}_*$. Thus, we can completely specify any such B in the exact sequence

$$0 \longrightarrow \mathcal{K} \longrightarrow \mathcal{A}_* \longrightarrow B \longrightarrow 0. \tag{7}$$

Applying the contravariant functor $\mathrm{Hom}_{\mathbb{F}_p}(-, \mathbb{F}_p)$ to this sequence, we see the dual sequence

$$0 \longrightarrow B_* \longrightarrow \mathcal{A} \longrightarrow \mathcal{K}_* \longrightarrow 0 \tag{8}$$

where B_* is now a sub-Hopf algebra of $\mathcal{A}$. By studying quotient Hopf algebras of $\mathcal{A}_*$, we in turn study sub-Hopf algebras of $\mathcal{A}$.

As we'll see, extra work must be done to obtain and keep track of the sub-algebra we're working with. Computations involving objects defined over different sub-algebras have to be done by rewriting those objects over a common sub-algebra over which they are all defined. The result is a package that gives complete answers valid in all degrees. While any finite sub-algebra over which our computation can be made would suffice, we try to reduce to the smallest such to reduce computation time.

3.1 Sub-Hopf algebras

Quotient Hopf algebras of $\mathcal{A}_*$ (dual to sub-Hopf algebras of $\mathcal{A}$) have been classified by Anderson and Davis (for $p = 2$) and Adams and Margolis (for p odd). We recall their classification here (see [3] and [5] for the proofs).

Theorem 1 (Classification theorem)**.** *Let J be a quotient Hopf Algebra of $\mathcal{A}_*$. Then*

$$J = \begin{cases} \mathcal{A}_*/(\xi_1^{2^{n_1}}, \xi_2^{2^{n_2}}, \ldots) & \text{if } p = 2, \text{ or} \\ \mathcal{A}_*/(\xi_1^{p^{n_1}}, \xi_2^{p^{n_2}}, \ldots; \tau_0^{\epsilon_0}, \tau_1^{\epsilon_1}, \ldots) & \text{if } p \text{ is odd.} \end{cases} \tag{9}$$

where $n_i \in \{0, 1, 2, \ldots\} \cup \{\infty\}$, and $\epsilon_i \in \{1, 2\}$. Furthermore, these exponents *satisfy the following conditions:*

(i) *For each i and r with $0 < i < r$, either $n_r \geq n_{r-i} - i$ or $n_r \geq n_i$.*

(ii) *If p is odd and $\epsilon_r = 1$, then for each i, $1 \leq i \leq r$, either $n_i \leq r - i$ or $\epsilon_{r-i} = 1$.*

Conversely, any set of exponents $\{n_i\}$ and $\{e_i\}$ satisfying these conditions determines a quotient Hopf Algebra of $\mathcal{A}_*$.

Note that if some $n_i = \infty$, then we impose no relation on ξ_i. Similarly, if $\epsilon_i = 2$, then the sub-algebra of $\mathcal{A}_*$ will contain τ_i.

3.2 Profile functions

By the Classification theorem, a quotient Hopf algebra of $\mathcal{A}_*$ (and thus, a sub-algebra of $\mathcal{A}$) is completely determined by a set of exponents.

Definition 2. *Let J be a quotient Hopf algebra of $\mathcal{A}_*$. The function which assigns to each index i, the exponent n_i, is referred to as the* profile function *for J. If J is defined over an odd primary Steenrod Algebra, then there is also the function which assigns to each index k, the exponent ϵ_k.*

The profile functions of greatest interest to us are those of finite sub-algebras. A finite profile $(n_1, \ldots, n_r)$ is extended by zeros, while the finite $(\epsilon_0, \ldots, \epsilon_r)$ is extended by ones. Condition (i) is independent of the ϵ_i, so results dependent only on Condition (i) are independent of the characteristic.

Lemma 3. *If $(n_1, \ldots, n_r)$ is a valid profile and $n_r \neq 0$, then $n_j \leq r$, for any j.*

Proof. Consider the requirements from $n_{r+k} = 0$, for any $k > 0$. Condition (i) of Theorem 1 implies either $n_{r+k} \geq n_{r+k-i} - i$ or $n_{r+k} \geq n_i$, for any i in the range $0 < i < r+k$. If $i = r$, these two conditions imply either $n_i \leq r$ or $n_r \leq 0$. The latter of these two cannot be true (by assumption), and so $n_i \leq r$. $\square$

Condition (i) has two parameters, i and r. For a finite profile function, the condition is redundant when the r parameter is more than twice the length of the profile.

Lemma 4. *If $(n_1, \ldots, n_r)$ is a valid profile function, then $n_{2r+1} = n_{2r+2} = \cdots = 0$ yield redundant relations on $(n_1, \ldots, n_r)$.*

Proof. For any $k > 0$, $n_{2r+k} = 0$. Theorem 1 implies either $0 \geq n_{2r+k-i} - i$ or $0 \geq n_i$. If $1 \leq i < r$, then the first condition is true, since $n_{2r+k-i} = 0$, for $1 \leq i < r$. If $i = r$, then the first condition is still true, since $n_r \leq r$ (by Lemma 3). Finally, if $r < i \leq 2r + k$, then the second condition is true, since $n_i = 0$, for $r < i \leq 2r$. $\square$

Lemma 1 implies that a profile of length j is valid only if Condition (i) of Theorem 1 holds for $r \leq 2j$. We now prove the analogous statement for odd characteristic algebras.

Lemma 5. *If $(n_1, \ldots, n_r), (\epsilon_0, \ldots, \epsilon_r)$ is a valid profile, then $\epsilon_{2r+1} = \epsilon_{2r+2} = \ldots = 1$ yield no new relations on $(\epsilon_0, \ldots, \epsilon_r)$.*

Proof. Consider the requirements from $\epsilon_{2r+k} = 1$, where $k > 0$. Theorem 1 requires that either $n_i < 2r + k - i$ or $\epsilon_{2r+k-i} = 1$, for any i, $1 \leq i \leq 2r + k$. If $1 \leq i \leq r$, we see $\epsilon_{2r+k-i} = 1$, as this piece of the profile function is extended indefinitely by ones. If $r < i < 2r + k$, then $n_i < 2r + k - i$, since $n_i = 0$ in this range. $\square$

Lemma 6. *Let $(n_1, \ldots, n_r)$ be a profile of length r. Then for any j with $r/2 < j \leq r$, we have $n_j \leq j$.*

Proof. Let j be in the specified range. Consider the relations stemming from $n_{2j} = 0$. Taking $r = 2j$ and $i = j$ in Condition (i), we see two conditions: $0 \geq n_{2j-j} - j$ or $0 \geq n_j$. In either case, $n_j \leq j$. $\square$

These bounds on profile functions reduce the test of validity to a finite calculation.

3.3 Utility Functions

In this section, we show various algorithms for working with profile functions. By the 'profile of an element or set of elements', we mean the profile function of the smallest sub-Hopf algebra of $\mathcal{A}$ containing these elements. These calculations are most easily done by working with elements in the Milnor basis for $\mathcal{A}$.

Near the end of each algorithm, we obtain a list of integers (n_i). We convert this to a 'pseudo-profile' (k_i), satisfying the exponent condition $p^{k_i-1} < n_i \leq p^{k_i}$, for each i. We call this list (k_i) a pseudo-profile since it defines a sub-algebra dual to algebras of the form of (9), which are Hopf sub-algebras only if they also satisfy Conditions (i) and (ii) of the Classification theorem.

Our algorithms convert pseudo-profiles to profiles while attempting to increase the sub-algebra defined by this profile by as little as possible. The phrase 'size of an algebra' is somewhat vague. Pseudo-profile functions have a lexicographic ordering, and we choose to minimize with respect to this. We believe this is a decent notion of minimal increase, and it is very easily implemented.

The complete list of functions used for sub-algebra calculations is listed. The only program intended for external use is find_min_profile.

- enveloping_profile_elements(L), the profile function for a list L of elements.
- enveloping_profile_profiles(P), the profile function for a list of profiles P.
- find_min_profile(J, p), the minimum profile containing J over the mod p Steenrod Algebra.
- next_prof(p, n), the next pseudo-profile following the pseudo-profile p, based at n (see below).

- profile_ele(x), the profile function of x.

The first program we describe is profile_ele. Given any element $x \in \mathcal{A}$,

$$k = \text{profile\_ele}(x)$$

sets the list $k = (k_i)$ to the profile function of x (see the first paragraph of 3.3). This is constructed by considering x in the Milnor basis,

$$x = \sum_{i=0}^{b} c_i \mathrm{Sq}^{R_i},$$

where each $R_i = (r_{i0}, r_{i1}, \ldots, r_{ik})$. Define $n_l = \max\{r_{il} | 0 \leq i \leq b\}$. Then for each l, find k_l so that $p^{k_l - 1} < n_l \leq p^{k_l}$ (the exponent condition). This list (k_l) is the pseudo-profile of a sub-algebra (not necessarily Hopf) and will contain the element x by construction. This list is then passed to find_min_profile to find a profile for a sub-Hopf algebra containing x.

Given a list of elements L,

$$P = \text{enveloping\_profile\_elements}(L)$$

sets P to the profile function of L. This is constructed by applying profile_ele to each element of L, and then computing the component-wise maximum as above. We increase each entry of this list to satisfy the exponent condition. The resulting pseudo-profile is passed to find_min_profile, yielding a profile for a sub-Hopf algebra containing L.

In a similar fashion, we construct the profile function corresponding to a list of profile functions. Namely, if L is a list of profile functions,

$$P = \text{enveloping\_profile\_profiles}(L)$$

computes the profile function corresponding to L. This is done by computing the component-wise maximum of the entries of L, and then increasing each entry to satisfy the exponent condition. This pseudo-profile is then passed to find_min_profile, and the result is returned.

Our algorithm for computing profile functions of modules and morphisms depends on finding the 'next' profile, where next refers to lexicographic order from left to right. This tries to ensure we are increasing the sub-algebra defined by this profile by as small a degree as possible. Explicitly,

$$L = \text{next\_prof}(p,n)$$

sets L to the next pseudo-profile after p, subject to the constraint that each entry in it is greater than the corresponding entry in n.

The next profile after p is obtained by incrementing lexicographically. For example, we increment the first entry of p until it cannot be incremented any further (Lemma 3). At this point, we reset the first entry of p to the first entry of n, and increment the second entry of p. Again, we increment the first entry with this new second entry. This process is repeated until no entry of p can be incremented any further. We then re-initialize p to the base profile n, increase its length by one, redefine n to be this new, longer base, and repeat.

The function find_min_profile ties all of the above programs together and is the only function designed for external use. Given a pseudo-profile J,

$$\text{P} = \text{find\_min\_profile}(J, p)$$

sets P to the profile of the smallest sub-Hopf algebra of the mod p Steenrod Algebra containing J. If P satisfies the Hopf condition, then it is returned. Otherwise, we call next_prof repeatedly (using J as the base profile) until a valid profile function corresponding to a sub-Hopf algebra is returned. This repetition will terminate, since every sub-algebra is contained in some $\mathcal{A}(n)$ [4, p.235].

4 Finitely presented modules

A finitely presented module M over a graded (Hopf) algebra A is one which has a presentation

$$0 \longleftarrow M \longleftarrow \bigoplus_{i=1}^{n} \Sigma^{d_i} A \xleftarrow{r} \bigoplus_{j=1}^{m} \Sigma^{e_j} A. \tag{10}$$

The module is specified by giving the degrees $[d_1, \ldots, d_n]$ of its generators and the values $r_j = [r_{j1}, \ldots, r_{jn}]$ of the relations. If B is a sub-algebra of A containing all the r_{ij}, then r is defined over B and we have a presentation of a B-module M_B

$$0 \longleftarrow M_B \longleftarrow \bigoplus_{i=1}^{n} \Sigma^{d_i} B \xleftarrow{r} \bigoplus_{j=1}^{m} \Sigma^{e_j} B. \tag{11}$$

When A is the Steenrod Algebra, all finitely generated submodules are finite. Since B is finitely generated, M_B is finite [4] and (11) is a sequence of finite modules. Applying $A \otimes_B -$ to (11) yields our original presentation (10). If, in addition, B is a sub-Hopf algebra of A, then $A \otimes_B -$ is exact [Milnor-Moore] so that we can compute kernels, cokernel, and images in the category of B-Modules. Our implementation of FP Modules relies on the fact that $\mathcal{A} \otimes_B -$ is an exact functor. Finitely presented $\mathcal{A}$-modules are infinite so that degree wise calculations will never terminate. In contrast, finitely presented B-modules are finite, so that the calculation of kernels, cokernels, images, etc. is a finite calculation. The fact that $\mathcal{A} \otimes_B -$ is exact means that the results in $\mathcal{A}$-mod will have the same presentations.

For our calculations, we need the degree by degree decomposition of our modules. For any FP Module M and degree n, we may take their underlying graded $\mathbb{F}_p$ vector spaces. Its degree n component is then an exact sequence

$$0 \longleftarrow M_n \longleftarrow \mathcal{F}_n \longleftarrow \mathcal{R}_n$$

of $\mathbb{F}_p$ vector spaces.

4.1 Specification

When defining an FP Module, four parameters can be specified: `degs`, `rels`, `char`, and `algebra`.

- `degs` is a list of integers, specifying the degrees of the generators.
- `rels` is a list of relations. By default, this is empty (i.e. the module is free).
- `char` is the characteristic of the module. By default, it is the characteristic of the algebra specified, or else 2.
- `algebra` is a Steenrod Algebra or some sub-Hopf algebra of it.

Based on the relations, we compute the smallest sub-Hopf algebra of `algebra` over which the module can be defined. This is done by computing the smallest algebra which contains the relations, via the `enveloping_profile_elements` function. We then redefine `algebra` to be this smaller algebra, to record the smallest sub-algebra over which calculations involving this module can be done.

In Section 2, we define several FP_Modules as examples, including the module Hko. Specifically, Hko $= \dfrac{\mathcal{A}}{(\mathrm{Sq}^1, \mathrm{Sq}^2)}$. As an $\mathcal{A}$-module, Hko is $H^*\mathrm{ko}$. To demonstrate our tensored up presentations, consider the following exact sequence in $\mathcal{A}$-mod.

$$\begin{tikzcd}
0 & Hko \arrow[l] & \mathcal{A} \arrow[l] & \Sigma\mathcal{A} \bigoplus \Sigma^2\mathcal{A} \arrow[l, "(\mathrm{Sq}^1{,}\mathrm{Sq}^2)"'] \\
0 & \mathcal{A} \otimes_{\mathcal{A}(1)} \mathbb{F}_2 \arrow[l] \arrow[u, equal] & \mathcal{A} \otimes_{\mathcal{A}(1)} \mathcal{A}(1) \arrow[l] \arrow[u, equal] & \mathcal{A} \otimes_{\mathcal{A}(1)} \left(\Sigma\mathcal{A}(1) \bigoplus \Sigma^2\mathcal{A}(1)\right) \arrow[l] \arrow[u, equal]
\end{tikzcd}$$

This sequence is the result of applying $\mathcal{A} \otimes_{\mathcal{A}(1)} -$ to the following sequence in $\mathcal{A}(1)$-mod.

$$0 \longleftarrow \mathbb{F}_2 \longleftarrow \mathcal{A}(1) \xleftarrow{\left(\mathrm{Sq}^2,\mathrm{Sq}^2\right)} \Sigma\mathcal{A}(1) \oplus \Sigma^2\mathcal{A}(1)$$

Here we list all attributes and methods of the FP_Module class intended for the user. Note the methods are followed by paranthesis and any parameters they require. Although neither an attribute or method, the DirectSum function is also intended for external use. The four which can be explicitly passed by the user when constructing an FP_Module are indicated by (*).

- alg(), the algebra the module is defined over.
- basis(n), the basis of the module in degree n.
- char, the characteristic of the algebra the module is defined over. (*)
- conn(), the connectivity of the module.
- copy(), an isomorphic copy of the module.
- degs, the degrees of the generators of the module. (*)
- gen(i), generator i of the module.
- gens(), the generators of the module.
- identity(), the identity map on the module.

- min_pres(), an isomorphic module with a minimal presentation.
- min_profile(), the profile function of the smallest algebra the module can be defined over.
- _pres_(n), the presentation of the module in degree n.
- profile(), the profile function of the algebra the module is defined over.
- rels, the relations of the module. (*)
- rdegs(), the degrees of the relations of the module.
- resolution(n), a length n free resolution of the module.
- submodule(L), the submodule generated by L.
- suspension(n), the n fold suspension of the module.

In addition to the above list, we have the remaining attributes and methods defined in the code. These are primarily intended for internal use.

- algebra (*)
- prof
- __call__(x).
- __contains__(x)
- __lc__(c, b)
- reldegs
- _repr_

The __call__ method allows the syntax M(coeffs) to define an element of M. Similiarly, __contains__ allows Sage to determine the truth value of 'x in M'.

4.2 Direct Sum

Given a list of FP_Modules $[M_0, M_1, \ldots, M_k]$,

$$K, I, P = \text{DirectSum}([M_0, M_1, \ldots, M_k])$$

sets

- K to the module corresponding to the direct sum $\bigoplus_{i=0}^{k} M_i$
- I to the list of inclusion maps $I_n : M_n \longrightarrow K$, for each n
- P to the list of projection maps $P_n : K \longrightarrow M_n$, for each n.

4.3 Presentations

The decomposition of our graded modules by degree is a very useful tool. The simplest example is determing a basis for the module in degree n. More involved applications include computing kernels and images, as well as the various forms of elements. Given an FP_Module M,

$$M_n, q, s, \text{bas\_gen} = M.\text{\_pres\_}(n),$$

sets

- M_n to the vector space of elements of M in degree n,
- q to the projection map from the degree n term of the free module on the generators of M to M_n (i.e. $q : \mathcal{F}_n \longrightarrow M_n$),
- s to a section of q,
- bas_gen to a standard basis for $\mathcal{F}_n$ (ordered as below).

The _pres_ function takes an optional parameter `algebra`, so M may be thought of as a module over `algebra` for a specific calculation. By default, this is M.alg(). A figure showing this data is shown.

$$
\begin{tikzcd}
\mathcal{F}_n \arrow[d, "q"] & = \langle \text{bas\_gen} \rangle \\
M_n \arrow[u, bend left, "\text{s}"]
\end{tikzcd}
$$

To construct this data, we begin by forming a basis for the degree n part $\mathcal{F}_n$ of the free module $\mathcal{F}$. This basis consists of pairs (i, b), where i is an index on generators g_i of M, and b is a basis element of $\mathcal{A}$ in degree $n - |g_i|$. These pairs form the basis $\langle \text{bas\_gen} \rangle$. This basis has a lexicographic ordering: $(i_1, b_1) < (i_2, b_2)$, if and only if $i_1 < i_2$ or $i_1 = i_2$ and $b_1 < b_2$ in the ordering used by the Milnor basis in the Steenrod Algebra (as in Palmieri's code).

To define M_n, we consider the relations of M in degrees less than or equal to n. For each of these relations r, we form the pairs $(i, b * r)$, where i is the index of a generator of M and b is a basis element of $\mathcal{A}$ in degree $n - |r|$. The union of all such pairs forms a spanning set for the vector space $\mathcal{R}_n$. Finally, M_n is simply the vector space quotient $\mathcal{F}_n / \mathcal{R}_n$.

If M_n is trivial, then q and s are defined trivially. Otherwise, s is defined by lifting a basis of M_n to $\langle \text{bas\_gen} \rangle = \mathcal{F}_n$. Similarly, q is defined by projecting the basis for $\mathcal{F}_n$ down to M_n. Since this is done on the level of vector spaces, Sage computes these lifts using linear algebra.

5 Elements

Elements of modules are defined with the FP_Element class. An element is defined by a list of coefficients on the generators of a module. These coefficients may lie in any sub-algebra of $\mathcal{A}$, not just the algebra specified by the parent module. Elements have various forms tailored to certain computations. We do not allow inhomogeneous elements, so all nonzero elements have a well-defined degree. The obvious arithmetic operators are overloaded, so computations with elements looks mathematical.

5.1 Specification

To define an FP_Element m, we pass a list of elements to the element constructor of an FP_Module M, as in

$$m = M\left([c_0, c_1, \ldots, c_n]\right),$$

where c_i is the coefficient on generator i of M.

Elements are allowed to take coefficients in the entire Steenrod Algebra, not just in the algebra of the parent module. To avoid recomputing the necessary profile, FP_Elements have a profile attribute which keeps track of the algebra the element is defined over. Hence, for any FP_Module M, defining $n = M(a)$ is allowed, even if $a \notin M$.algebra. See Section 2 for examples involving FP_Elements.

Here we give a complete list of attributes and methods of the FP_Element class intended for the user.

- coeffs, the natural form of an element.
- degree, the degree of the element.
- module, the module of the element.

- profile, the profile function of the smallest algebra the element can be defined over.
- free_vec(), the free vector form of the element.
- vec(), the vector form of the element.
- nf(), the normal form of the element.

In addition to the above, we have the following attributes and methods, intended for internal use only. The methods here are used by Sage to allow ordinary algebraic syntax to be used on FP_Elements.

- __add__(x)
- __cmp__(x)
- __iadd__(x)
- __mul__(x)
- __neg__(x)
- __nonzero__()
- _parent
- _repr_
- __sub__(x)

5.2 Arithmetic

Simple arithmetical operators are defined for FP_Elements. This includes testing for equality amongst elements, and testing whether an element is zero.

One peculiarity with our code is how the left module action of the Steenrod Algebra is written. Due to some unknown error, we must write the action on the right, although *we stress that it really is a left action.* So for $a, b \in \mathcal{A}$ and $m \in M$, we write $m \cdot (ab)$ to mean $(ab) \cdot m = a \cdot (b \cdot m)$, which is then written $(m \cdot b) \cdot a$. Care should be taken to avoid confusion over this annoyance.

5.3 Forms for elements

Each element can be represented in several different ways, adapted to the operation to be carried out. The first of these is the *natural form.* The natural form of an element is simply a list of coefficients. If m is an element, its natural form is retrieved by calling m or m.coeffs.

For an element m of an FP_Module M

$$v = m.\text{free\_vec}()$$

sets v to the corresponding *free vector* in $\mathcal{F}_n$, where $\mathcal{F}_n$ is the degree $n = |m|$ vector space of the free $\mathcal{A}$-module on the generators of M. The free vector form of an element allows vector space operations to be carried out, but does not take the relations into account. The construction of the free vector form of an element is similar to the presentation function for FP_Modules. We compute the basis for $\mathcal{F}_n$ consisting of the pairs (i, b_i), where i is an index on the generators of the module, and b_i is a basis element for the Steenrod Algebra in degree $n - |g_i|$. We express m as a sum of $c_i * g_i$, where c_i is the coefficient on the i^{th} generator. We express each c_i in terms of the b_i, and sum to get the free vector form. This function also takes an optional profile parameter. This is to expand $\mathcal{F}_n$ to allow coefficients from a larger sub-algebra than that defined by the module of the element m. Degree 4 of the example in Figure 1 shows the effect of this.

$$M([\text{Sq}(1,1)]).\text{free\_vec}()$$

is the unique nonzero element in a 1 dimensional $\mathbb{F}_2$ vector space, while

$$M([\mathrm{Sq}(1,1)]).\text{free\_vec(profile=(3,2,1))}$$

returns the first basis element in a 2 dimensional $\mathbb{F}_2$ vector space. Note that

$$M([\mathrm{Sq}(4)]).\text{free\_vec()}$$

would return the second basis vector in this 2 dimensional vector space. The profile parameter is not needed here because the coefficient Sq(4) raises the profile automatically.

In a similar manner, we define the *vector form* of an element m to be

$$w, q, s, \mathrm{bas} = m.\mathrm{vec}(),$$

which defines

- w to be the corresponding vector in M_n, where n is the degree of m
- q, s, and bas as in M._pres_(n).

Recall that $M_n = \mathcal{F}_n/\mathcal{R}_n$, so the vector form of an element takes the relations into account. This is constructed by computing the presentation of the module M in degree $n = |m|$, and applying q to the free vector form of m to obtain w. This function also takes the optional profile parameter just as above.

In order to pass from the vector space M_n to the module M, we define the _lc_ function. Given a list of coefficients (co), and a list of pairs (i, b_i) forming a basis (bas) for $\mathcal{F}_n$,

$$k = \text{M.\_lc\_(co,bas)}$$

sets k to the FP_Element $\sum_j co_j * b_j * g_j$, where $\mathrm{bas}_j = (j, b_j)$. This form is not normalized, since the relations on the g_j are not taken into account.

The *normal form* of an element is the most convenient for computations, since it takes the relations of the module into account. In particular, $x == y$ if and only if their normal forms are the same. For an element m,

$$z = m.\mathrm{nf}(),$$

defines z to be the FP_Element corresponding to m under the relations of the module M.

For example, if

$$M = \text{FP\_Module([0,1],[[Sq(2), Sq(1)]])},$$

then $M([0, \mathrm{Sq}(1)])$ and $M([\mathrm{Sq}(2), 0])$ both have normal form $M([\mathrm{Sq}(2), 0])$.

This is constructed by first computing the vector form of m in M_n and then applying _lc_, where n is the degree of m. Each element has many natural forms which differ by elements of the module of relations R. The normal form is the first of these when arranged in lexicographic order. The section s is chosen to implement this. This function also takes the optional profile parameter as usual.

The relationships between the different forms is displayed below.

$$\begin{tikzcd}
\text{free vector form} & \text{vector form} \\[-20pt]
x.\text{free_vec}() & x.\text{vec}() \\
\mathcal{F}_n \arrow[r, "q"] \arrow[d, squiggly, "\text{_lc_}"'] & M_n \arrow[l, bend left, "s"] \\
F \arrow[r] & M \\
\text{natural form} & \text{normal form} \\[-20pt]
x & x.\text{nf}()
\end{tikzcd}$$

Hence, for any FP_Element $x \in M$, we have $x = M$._lc_(x.free_vec(),bas). For any vector $v \in \mathcal{F}_n$, $v = M$._lc_(v,bas).free_vec().

6 Homomorphisms

The FP_Hom class allows the user to define homomorphisms. With the FP_Hom class, we can compute kernels, cokernels, images, minimal presentations of FP_Modules, and submodules generated by a set of FP_Elements. This basic functionality is expanded to compute lifts of FP_Homs and the homology of a pair of maps, to name a few examples. In Section 2, we show examples of the kernel, image, and cokernel examples.

6.1 Specification

Four parameters can be specified when defining an FP_Hom: `domain`, `codomain`, `values`, and `degree`.

- `domain` is the source FP_Module.
- `codomain` is the target FP_Module.
- `values` is a list of values, whose i^{th} entry is a list corresponding to the value on the i^{th} generator.
- `degree` is an integer corresponding to a suspension of the domain before mapping. By default, it is 0.

A morphism must be well-defined and an error is raised if relations don't map to zero. Note that the entries in `values` can either be lists of elements of the Steenrod Algebra which we coerce into the codomain, or lists of elements of the codomain. Furthermore, if `degree` is non-zero, then the domain of a map is not the domain passed to the FP_Hom, but a copy of its suspension, which should be retrieved with the domain attribute.

FP_Homs are necessarily defined over a finite sub-algebra because of the finite generation of the domain. However, they need not be defined over the same sub-algebra as the domain and codomain. Their profile attribute records the enveloping profile function of the domain, codomain, and the map. In other words, the homomorphism is tensored up from the category of modules over this sub-Hopf algebra.

Here we list all attributes and methods of the FP_Hom class intended for the user. Those which can be explicitly passed by the user when constructing an FP_Hom are indicated by (*).

- codomain, the target of the map. (*)
- degree, specifies a suspension of the domain before mapping. (*)
- domain, the source of the map. (*)
- values, the values of the map. (*)
- alg(), the algebra the map is defined over.
- cokernel(), the cokernel of the map.
- _full_pres_(n), the complete degree n presentation of the map.
- image(), the image of the map.
- kernel(), the kernel of the map.
- kernel_gens(), the generators of the kernel of the map.
- min_profile(), the profile function of the smallest algebra the map can be defined over.
- _pres_(n), the degree n presentation of the map.
- profile(), the profile function of the algebra the map is defined over.
- solve(x), an element which maps to x.

- suspension(n), the n^{th} suspension of the map.

In addition to the above list, we have the remaining attributes and methods defined in the code. These are primarily intended for internal use.

- algebra
- prof
- __add__(f)
- __call__(x)
- __cmp__(f)
- is_zero()
- __mul__(f)
- __neg__()
- _repr_()
- __sub__(f)

Note that these allow algebraic operations on FP_Homs, e.g., $f = g - h * k$, where $h * k$ is the composite of h and k.

6.2 Presentations

A useful tool for computing kernels and images of FP_Homs is a degree wise presentation, similar to the presentation of FP_Modules. This includes the domain and codomain vector spaces in the relevant degree, along with a linear transformation between these presentations. Specifically

$$f_i,\ \mathrm{M}_i,\ \mathrm{Mq},\ \mathrm{Ms},\ \mathrm{Mb},\ \mathrm{N}_i,\ \mathrm{Nq},\ \mathrm{Ns},\ \mathrm{Nb} = f.\_\mathrm{full\_pres}\_(i)$$

sets

- f_i to the linear transformation corresponding to f in degree i
- M$_i$, Mq, Ms, and Mb to the output of f.domain._pres_(i)
- N$_i$, Nq, Ns, and Nb to the output of f.codomain._pres_(i).

A figure showing this data is given.

$$\begin{tikzcd}
\langle \mathrm{Mb} \rangle \arrow[d, "\mathrm{Mq}"] & \langle \mathrm{Nb} \rangle \arrow[d, "\mathrm{Nq}"] \\
M_i \arrow[u, bend left, "\mathrm{Ms}"] \arrow[r, "f_i"] & N_i \arrow[u, bend left, "\mathrm{Ns}"] \\
M \arrow[r, "f"] & N
\end{tikzcd}$$

This is done by computing the presentations of the domain and codomain in the relevant degree, and the induced linear transformation between these vector spaces. This linear transformation sends the generators of M_i to the corresponding elements in N_i, specifed by the FP_Hom. There is a _pres_ function, which only returns the linear transformation f_i.

6.3 Kernel

If f is an FP_Hom, then

$$K, i = f.\mathrm{kernel}()$$

sets K to the FP_Module ker(f) and i to the inclusion

$$K = \ker(f) \xrightarrow{i} f.\mathrm{domain}().$$

The calculation of the kernel is done over the algebra f.alg(). This implies all generators and relations of the kernel must lie in a range of degrees, from f.domain.connectivity to f.domain.top_deg, where the top degree of the domain is given by max(f.domain.degs) $+2 \cdot$ top_degree(f.alg()).

We have an associated method

$$G, j = f.\text{kernel\_gens}()$$

which sets G to a free module on the generators of the kernel and $j : G \longrightarrow \ker(f)$. This construction is the same as the kernel method.

Our construction of the kernel works as follows. Suppose

$$K \xrightarrow{i} M \xrightarrow{f} N$$

is a zero sequence ($fi = 0$). Then there is a unique factorization

$$\begin{tikzcd}
K \arrow[dr, "i"] \arrow[dd, dotted, "j"'] & & \\
& M \arrow[r, "f"] & N. \\
\ker(f) \arrow[ur, "k"'] & &
\end{tikzcd}$$

Our algorithm progressively improves K a degree at a time until j is an isomorphism. The construction begins by setting $K = 0$ and noting that in this case, j is an isomorphism in degrees less than the connectivity of M.

Inductively, if j is an isomorphism in degrees less than n, consider the degree n parts of the diagram:

$$
\begin{tikzcd}
K_n \arrow[dr, "i_n"] \arrow[dd, dotted, "j_n"'] & & \\
& M_n \arrow[r, "f_n"] & N_n \\
\ker(f_n) \arrow[ur, "k_n"'] & &
\end{tikzcd}
$$

Here, the $\text{cok}(j_n)$ gives the generators of $\ker(f)$ in degree n which are not present in K. Dually, $\ker(j_n)$ is the set of relations among the generators of K which hold in $\ker(f)$ but not in K.

We may therefore improve $i : K \longrightarrow M$ to make j an isomorphism in degrees less than or equal to n by the following two procedures. First, we add generators to K in degree n in one to one correspondence with a basis for $\text{cok}(j_n)$. The map j_n is defined on these new generators by sending each to a coset representative and i is then defined on these new generators by $k_n \circ j_n$. By construction, this new j_n has zero cokernel. Second, we add relations to K in one to one correspondence with a basis for $\ker(j_n)$. After this, j_n is a monomorphism and hence an isomorphism in degree n.

This algorithm is repeated until n is f.domain.top_deg $+ 2 \cdot$ f.alg().top_deg, at which point there can be no further generators or relations. Above f.domain.top_deg, $\text{cok}(j_n)$ will always be 0, so no further generators will be added after this.

6.4 Cokernel

Given an FP_Hom f, then

$$C, p = f.\text{cokernel}()$$

defines C as the FP_Module $\text{cok}(f)$ and p the projection

$$f.\text{codomain} \xrightarrow{p} C = \text{cok}(f).$$

The following algorithm may produce a cokernel with excess generators and relations. If a more efficient presentation is desired, cokernel takes an optional parameter min, which is False by default. If min is True, then image of the identity map on C is computed and returned, yielding a minimal presentation (see Section 6.5).

The cokernel is computed with the following "lazy" algorithm. Given $f : M \longrightarrow N$, we define C to be a copy of N with extra relations. For each generator g of M, we add the relation $f(g)$ to C. This C is isomorphic to $\mathrm{cok}(f)$. The map p is then the natural map $N \longrightarrow C$ which maps each generator of N to the copy of it in C.

6.5 Image

For any FP_Hom f,

$$I, p, i = f.\mathrm{image}()$$

sets I to the FP_Module $\mathrm{im}(f)$, and i and p to the factorization

$$\begin{tikzcd} f.\mathrm{domain} \arrow[rr, "f"] \arrow[dr, "p"'] & & f.\mathrm{codomain} \\ & I = \mathrm{im}(f) \arrow[ur, "i"'] & \end{tikzcd}$$

where p is the projection and i is the inclusion.

One use of the image method is the computation of submodules. Given a subset X of a module M, let $F(X)$ be the free module on X. The submodule generated by X is then the image of the natural map $F(X) \longrightarrow M$.

The image method also allows one to refine presentations. For a module M, $\mathrm{im}(\mathrm{id}_M)$ gives a minimal presentation. This is useful enough that we give it a name:

$$M', g, h = M.\text{min\_pres}()$$

gives a minimal presentation M' of M together with inverse isomorphisms g and h.

$$\begin{tikzcd}
M \arrow[rr, "\mathrm{id}_M"] \arrow[dr, "g"'] & & M \\
& M' \arrow[ur, "h"'] &
\end{tikzcd}$$

The algorithm for the image method is as follows. Given $f : M \longrightarrow N$, we initially define I' to be the zero module, and $p' : M \longrightarrow I'$ and $i' : I' \longrightarrow N$ to be 0 maps. We proceed by induction on the generators of M. Since we're working with homomorphisms, its sufficient to only consider the generators of M. Furthermore, we work over the algebra f.alg().

$$\begin{tikzcd}
F \arrow[r, "\pi"] \arrow[ddrr, "p'"'] & M \arrow[rr, "f"] \arrow[dr, dotted, "p"] & & N \\
& & I \arrow[ur, dotted, "i"] & \\
& & I' = \bigoplus_{i \in J} \Sigma^{d_i} \mathcal{A} \arrow[u, "q"] \arrow[uur, "i'"'] &
\end{tikzcd}$$

Suppose M is generated by $\{g_1, g_2, \ldots, g_k\}$. For each generator g_j of M, assuming $i' : I' \longrightarrow N$ is onto the image of f in degrees less than $n = |g_j|$:

- if $f(g_j) \in \mathrm{im}(i') = \mathrm{Span}\{f(g_1), \ldots, f(g_{j-1})\}$, let $p'(g_j)$ be an element x satisfying $i'(x) = f(g_j)$.
- If $f(g_j) \notin \mathrm{im}(i') = \mathrm{Span}\{f(g_1), \ldots, f(g_{j-1})\}$, then add a generator h_j to I' in degree n and define $i'(h_j) = f(g_j)$ and $p'(g_j) = h_j$.

The hypothesis regarding i' is true at the start of the algorithm ($j = 1$), and remains true after each iteration. Precisely, we've defined p' by the following formula, if we let $J = \{\, j \,|\, f(g_j) \notin \mathrm{Span}\{f(g_1), \ldots, f(g_{j-1})\}$, then

$$p'(g_j) = \begin{cases} h_j & \text{if } j \in J \\ \displaystyle\sum_{l=1}^{j-1} \alpha_l h_l & \text{if } j \notin J \text{ and } f(g_j) = \displaystyle\sum_{\substack{l=1 \\ l \in J}}^{j-1} \alpha_l i'(h_l). \end{cases}$$

At the end of this process, I' is a free module mapping onto the image of f, and by construction, is minimal.

Now let $I, q = \mathrm{cok}\big(\ker(i') \longrightarrow I'\big)$. Then by minimality, I' and I will have the same generators, and i' factors uniquely as iq, where the list of values defining i is identical to the list of values defining i'. Further, i is monic by construction and $qp'(\ker(\pi)) = 0$, since f is well-defined. Hence, qp' factors as $p\pi$ and, again, the list of values defining p is identical to the list of values defining p'. For efficiency, it is sensible to replace each of these values by its normal form. The set of relations generated by this algorithm is minimal because i'.kernel() returns a minimal set of generators for the kernel.

At the end of this process, we have produced an epi-mono factorization of f, and so it must be isomorphic to the image.

6.6 Lifts

The notion of a lift will prove necessary for working with chain complexes. The first step in this direction is to lift individual elements with the solve function. Given an FP_Hom $f : M \longrightarrow N$, and $x \in N$,

$$\text{bool}, \, v = f.\text{solve}(x)$$

sets bool to the boolean value of ‘there exists $v \in M$ such that $f(v) = x$’, and v to this value if it exists. This function is computed by considering the corresponding linear transformation f_n on vector spaces in degree $n = |x|$. If the vector form of x lies in the image of f_n, then $f_n(j) = x$, for some $j \in M_n$.

In this case, (true, v) is returned, where v is the FP_Element corresponding to j. If $x.vec()$ is not in the image of f_n, then (false, 0) is returned.

The solve function naturally extends to define lifts of maps over others. Given morphisms f and g,

$$\text{bool}, l = \text{lift}(f,g)$$

sets bool to the boolean value of 'f lifts over g' (i.e. $f = gh$ for some h), and l to this lift, if possible. This lift is defined by lifting each of the generators using the solve function and extending linearly.

7 Chain complexes

In our package, a chain complex is a list of composable maps $[L_0, \ldots, L_k]$

$$\xleftarrow{L_0} \quad \xleftarrow{L_1} \cdots \xleftarrow{L_k}$$

The modules in the complex can be extracted using the domain and codomain attributes. In Section 2 we show examples of chain complexes and their associated functions.

Here we list functions defined for working with chain complexes.

- is_complex(L), a check of whether L forms a chain complex.
- is_exact(L), a check of whether L is an exact sequence.
- homology(f, g), the homology a pair of maps.
- resolution(k), a length n free resolution of a module.
- extend_resolution(L, n), a resolution extending L to length n.
- chain_map(L, R, f), the chain map lifting f between L and R.

7.1 Homology

Given any list of maps, the is_complex function tests whether the sequence of maps form a chain complex. Similarly, is_exact tests whether the sequence of maps form an exact sequence. We can measure the extent to which pairs of maps fail to be exact with the homology function. Specifically,

$$H, p, i, m, l = \text{homology}(f, g)$$

defines

- H as the module $\ker(g)/\operatorname{im}(f)$

- p as the projection $\ker(g) \longrightarrow H$
- i as the inclusion $\ker(g) \longrightarrow g$.domain()
- m as the inclusion $\mathrm{im}(f) \longrightarrow g$.domain()
- l as the lift of m over i.

This information is encoded in the following diagram.

$$\begin{tikzcd}
A \arrow[rr, "f"] \arrow[dr, two heads] & & B \arrow[rr, "g"] & & C \\
& \mathrm{im}(f) \arrow[ur, "m"] \arrow[rr, "l"] & & \ker(g) \arrow[ul, "i"'] \arrow[r, "p"] & H
\end{tikzcd}$$

This construction follows directly from the image, kernel, and lift functions.

7.2 Resolutions

Given an FP_Module M,

$$R = M.\text{resolution}(k)$$

sets R to a free resolution $[R_0, R_1, \ldots, R_{k-1}]$ of M of length k. The resolution R satisfies the following factorization

$$\begin{tikzcd}
M & M_0 \arrow[l, "R_0"'] & & \cdots \arrow[ll, "R_1"'] \arrow[dl, two heads] & & M_{k-1}. \arrow[ll, "R_{k-1}"'] \arrow[dl, two heads] \\
& & \ker(R_0) \arrow[ul] & \cdots & \ker(R_{k-2}) \arrow[ul] &
\end{tikzcd}$$

This method has an optional verbose parameter, which is false by default. If true, a print statement is displayed whenever the next stage of the resolution is computed. There is also an extend_resolution program, which will extend a resolution to any desired length. This also has the optional verbose parameter.

7.3 Chain Maps

Suppose L and R are two chain complexes, and f is a map from the codomain of L_0 to the codomain of R_0. Using the lift function,

$$F = \text{chain\_map}(L, R, f)$$

sets F to a chain map between the complexes L and R. If the lengths of L and R aren't the same, the 0-map will fill the remaining entries of F. This information is displayed below.

$$\begin{tikzcd}
A_0 \arrow[d, "f=F_0"'] & A_1 \arrow[l, "L_0"'] \arrow[d, dotted, "F_1"'] & A_2 \arrow[l, "L_1"'] \arrow[d, dotted, "F_2"'] & \cdots \arrow[l, "L_2"'] \\
B_0 & B_1 \arrow[l, "R_0"'] & B_2 \arrow[l, "R_1"'] & \cdots \arrow[l, "R_2"']
\end{tikzcd}$$

7.4 Extensions

Our code also has the ability to compute extensions defined by cocycles. Given $e_1 \in \mathrm{Ext}^1(M_3, M_1)$, and a resolution R of M_3 (of length at least 3),

$$M, i, j = \text{extension}(e_1, R)$$

defines the associated extension. This is a module M and maps i, e, and j, such that $[j, i]$ is short exact and the following diagram commutes.

$$\begin{tikzcd}
0 \arrow[r] & M_1 \arrow[r, "i"] & M \arrow[r, "j"] & M_3 \arrow[r] & 0 \\
F_2 \arrow[u] \arrow[r, "R_2"] & F_1 \arrow[u, "e_1"] \arrow[r, "R_1"] & F_0 \arrow[u, "e"] \arrow[r, "R_0"] & M_3 \arrow[u, equal] \arrow[r] & 0
\end{tikzcd}$$

This program also has an optional test parameter, which is true by default. If test remains true, we check the exactness and length of R, and the cocycle condition on e_1.

We begin the construction of the extension by first computing the module M. This amounts to computing the pushout of e_1 and R_1, i.e. $M = \mathrm{cok}(i_0R_1 - i_1e_1)$

and maps i and e. Our construction explicitly uses the intermediary direct sum.

$$
\begin{tikzcd}
M_1 \arrow[rr, dotted, "i"] \arrow[dr, "i_1"] & & M \\
& F_0 \bigoplus M_1 \arrow[ur, dotted, "p"] & \\
F_1 \arrow[uu, "e_1"] \arrow[rr, "R_1"] & & F_0 \arrow[ul, "i_0"'] \arrow[uu, dotted, "e"']
\end{tikzcd}
$$

Define $i = pi_1$ and $e = pi_0$. To define a map from M to M_3, we first compute $q : M \longrightarrow \mathrm{cok}(i)$. Then we compute an extension g of qe over R_0, i.e. $gR_0 = qe$. (This requires no further calculation since R_0 is onto.)

$$
\begin{tikzcd}
& & \mathrm{cok}(i) & \\
M_1 \arrow[r, "i"] & M \arrow[ur, "q"] \arrow[r, dotted, "j"] & M_3 \arrow[u, "g"] \arrow[r] & 0 \\
F_1 \arrow[u, "e_1"] \arrow[r, "R_1"] & F_0 \arrow[u, "e"] \arrow[r, "R_0"] & M_3 \arrow[u, equal] \arrow[r] & 0
\end{tikzcd}
$$

To finish this process, we lift q over g, to obtain a map $j : M \longrightarrow M_3$. This completes the algorithm.

8 Future work

While we have laid the groundwork for cohomology calculations in Sage, there is still much to be done. The first point of improvement is more efficient algorithms. Certain programs defined in this package, like kernel and image, can be rewritten to be faster. Along with re-working algorithms, one may use the Cython capability of Python to further increase computation speed.

One operation not implemented in this package is the tensor product. The first step in this construction is to find the smallest sub-algebra B over which $M \otimes N$ is defined. In general, this will be much larger than the algebras of M and N. This involves analyzing the Cartan formula on products of the generators of M and N. From this point, there are several constructions of the tensor product one could implement. Efficiency and computation time are much bigger factors in this computation, as compared to others we have discussed here.

Another possible point of future work is finite modules. Finite modules cannot be defined in our current package, because of the infinite number of relations in their presentations. For example, the cohomology of the Moore space $S^0 \cup_2 e^1$ has a single generator in degree 0, but the relation Sq^{2^i} for every $i > 1$.

A finite module could be thought of as an FP_Module M together with an integer m, the truncation degree, defining the module which is isomorphic to M in degrees less than or equal to m and 0 in higher degrees. A homomorphism in the category of finite modules $f' : M(-\infty, m] \longrightarrow N(-\infty, n]$ would correspond to a diagram

$$\begin{tikzcd}
M \arrow[rr, "f"] \arrow[d] & & N \arrow[d] \\
M(-\infty, m] \arrow[rr, "f'"] & & N(-\infty, n]
\end{tikzcd}$$

where f is the corresponding map in the category of finitely presented modules.

Finite modules could be used to define $\mathcal{A}(n)$ directly as a module. This would allow the user to define finitely presented $\mathcal{A}(n)$-modules tensored up from $\mathcal{A}(i)$-modules, instead of just finitely presented $\mathcal{A}$-modules tensored up from $\mathcal{A}(n)$ modules.

Altogether, our package allows one to perform tedious calculations over the Steenrod Algebra. By working with finitely presented modules and sub-Hopf algebras, we produce complete results valid in all degrees. We've defined numerous methods and functions to give the user a variety of tools for computations.

9 Appendix A: The Sage Code

Here we display the actual source code for `FPMods`.

```
r"""
Finitely presented modules over the Steenrod Algebra.

COMMENT ABOUT RING OF DEFINITION:

This package is designed to work with modules over the whole Steenrod
algebra. To make calculations finite, we take account of the fact
that finitely presented modules are defined over a finite sub Hopf
algebra of the Steenrod algebra, and (eventually) that a finite module
is a finitely presented module modulo the submodule of elements above
its top degree.

A module's profile is taken to be the smallest one over which it can
be defined, unless explicitly raised by passing a profile parameter.
The coefficients of an element of that module (FP_Element), however,
can lie anywhere in the Steenrod algebra: our profiles are simply
recording the subalgebra over which the module is defined, not forcing
the module into the category of modules over that subalgebra. To make
this work nicely, an element also has a profile, and computing with
elements involves finding the enveloping profile. SAY WHY THIS MAKES
IT WORK NICELY IF POSSIBLE. ERASE THIS COMMENT IF NOT.n
```

```
#A paragraph about our 'package' below.
...

AUTHORS:

- Robert R. Bruner (2010--): initial version
- Michael J. Catanzaro
...

EXAMPLES:
#put lots of different examples.
...

#*****************************************************************************
#       Copyright (C) 2010 Robert R. Bruner <rrb@math.wayne.edu>
#                  and          Michael J. Catanzaro <mike@math.wayne.edu>
#
#  Distributed under the terms of the GNU General Public License (GPL)
#                  http://www.gnu.org/licenses/
#*****************************************************************************



"""
#-----------------------------------------------------------------------------
#---------------------------Utility-functions----------------------------------
#-----------------------------------------------------------------------------
```

```
def maxim(L):
    """
    Input a list of tuples and this function returns a tuple whose
        i-th entry is the
    max of the i-th entries in the tuples in L, where shorter
        tuples are extended
    by adding 0's

    INPUT:

    - ``L`` - A list of tuples

    OUTPUT: The component-wise maximum of the entries of L

    EXAMPLES::

    sage: maxim([[1,2,3,4],[5,4]])
    [5,4,3,4]
    sage: maxim([[1,2,3,4],[1,1,1,5],[9]])
    [9, 2, 3, 5]

    """
    if len(L) == 1:
        return [L[0][i] for i in range(len(L[0]))]
    else:
        t1 = maxim([L[i] for i in range(1,len(L))])
        t2 = [L[0][i] for i in range(len(L[0]))]
  mm = max(len(t1),len(t2))
  t1 = t1 + (mm - len(t1))*[0]
  t2 = t2 + (mm - len(t2))*[0]
  return map(max,zip(t1,t2))

def _deg_(degs,co):
    """
```

```
    Computes the degree of an FP_Element. 'degs' is a list of
        integral degrees,
    and 'co' is a tuple of Steenrod operations, as in an FP_Element
        .
    If all coefficients are 0, returns None.

    INPUT:

    - ``degs`` - A list of integers.

    - ``co`` - A list of Steenrod algebra elements.

    OUTPUT: The degree of the FP Element formed by `degs` and `co`.

    EXAMPLES::

    sage: A = SteenrodAlgebra(2)
    sage: _deg_((0,2,4),(Sq(4),Sq(2),Sq(0)))
    4
    sage: _deg_((3,3),(Sq(2)*Sq(1),Sq(1)*Sq(2)))
    6

    """
    if len(degs) != len(co):
        raise ValueError,\
        "Wrong number (%s) of coefficients. Should be %s.\n" % (len
            (co),len(degs))
    nz = filter(lambda i: co[i] != 0, range(len(degs)))  # figure
        out which are
    d = [degs[i]+co[i].degree() for i in nz]            # non-zero
    if len(d) == 0:
        return None
    if min(d) != max(d):
        raise ValueError, "Inhomogeneous element"
    return min(d)
```

```
def max_deg(alg):
    """
    Computes the top nonzero degree of a sub-algebra of the
        Steenrod Algebra.

    INPUT:

    - ``alg`` - A sub-algebra of the Steenrod Algebra.

    OUTPUT:

    - ``topdeg`` - The top nonzero degree of `alg`.

    EXAMPLES::

    sage: A2 = SteenrodAlgebra(p=2, profile = (3,2,1))
    sage: max_deg(A2)
    23
    sage: K = SteenrodAlgebra(p=2,profile=(0,))
    sage: max_deg(K)
    0
    sage: max_deg(SteenrodAlgebra(p=5,profile=((3,2,1),(1,1,1))))
    3136


    """
    if alg._truncation_type == +Infinity:
        raise ValueError, "Maximum degree is +Infinity"
    p = alg._prime
    if p == 2:
        topdeg = 0
        prof = [0] + list(alg._profile)
        for i in range(len(prof)):
            topdeg += (2**(i) -1)*(2**(prof[i])-1)
```

```
            return topdeg
        else: # p odd
            topdeg, epsdeg = (0,0)
    prof = [0] + list(alg._profile[0])
    for i in range(len(prof)):
        topdeg += 2*(p**i-1)*(p**(prof[i])-1)
    prof2 = list(alg._profile[1])
            for i in range(len(prof2)):
        epsdeg += (2*p**(i)-1)*(prof2[i]-1)
    return epsdeg+topdeg


def pmax_deg(prof,p=2):
    """
    Computes the top nonzero degree of the sub-algebra of the
        Steenrod Algebra
    corresponding to the profile passed. Note: Does not have to be
        a valid profile,
    only a tuple or list of nonnegative integers.

    INPUT:

    -  ``p`` - The prime over which the degree computation is made
        . By default, `p` = 2.

    -  ``prof`` - A profile function corresponding to a sub-algebra
         of the Steenrod
                    Algebra. If `p` =2, `prof` is a list or tuple. If
                         `p` is odd, `prof`
      is a 2-tuple, corresponding to each piece of a profile
          function.
    OUTPUT:

    -  ``topdeg`` - The top nonzero degree of the sub-algebra.


    EXAMPLES::
```

```
    sage: pmax_deg((2,1))
    6
    sage: pmax_deg(((3,2,1),(1,1,1)),p=3)
    336


    """
    if p == 2:
        topdeg = 0
        prof = [0] + list(prof)
        for i in range(len(prof)):
            topdeg += (2**(i) -1)*(2**(prof[i])-1)
        return topdeg
    else: # p odd
        topdeg, epsdeg = (0,0)
  prof1 = [0] + list(prof[0])
  prof2 = list(prof[1])
  for i in range(len(prof1)):
      topdeg += 2*(p**i-1)*(p**(prof1[i])-1)
        for i in range(len(prof2)):
      epsdeg += (2*p**(i)-1)*(prof2[i]-1)
  return epsdeg+topdeg



def _del_(i,n):
    """
    A list form of the Kronecker delta function.

    INPUT:

    - ``i`` - The position at which the list will take the value
        1.

```

```
    - ``n`` - The length of the list

    OUTPUT:

    - ``ll`` - A list of length `n`, consisting of all zeros
      except
                for a 1 in   `i^{th}` position.

    EXAMPLES::

    sage: _del_(2,4)
    [0, 0, 1, 0]
    sage: _del_(0,3)
    [1, 0, 0]
    sage: _del_(6,4)
    ValueError: List of length 4 has no entry in position 6.

    """
    if i >= n:
        raise ValueError, "List of length %d has no entry in
            position %d." % (n,i)
    ll = n*[0]
    ll[i] = 1
    return ll

def mod_p_log(n,p):
    """
    Input an integer n and a prime p
    Output the k so that p^{k-1} lt n le p^k

    EXAMPLES::

    sage: mod_p_log(1,4)
    1
    sage: mod_p_log(8,3)
```

```
    2
    sage: mod_p_log(9,3)
    3

    """
    k=0
    pow=1
    while n >= pow:
        k += 1
  pow *= p
    return k

def DirectSum(alist,char=None):
    """
    Returns the direct sum of the list of FP_Modules passed. A list
         of inclusion maps and
    a list of projection maps are also returned.

    INPUT:

    - ``alist`` - A list of FP_Modules.

    OUTPUT:

    - ``M`` - The direct sum of Modules in `alist`.

    - ``incl`` - A list of inclusion maps, one from each module
        to M.

    - ``proj`` - A list of projection maps, from M to the modules
        .

    """
    if len(alist) == 0 and char == None:
  raise ValueError, "Empty Direct Sum -- undefined characteristic."
```

```
        #M,incl,proj = FP_Module([],[]).copy() ## default alg p =2?
    #return M,[incl],[proj]
      elif len(alist) == 0:
        M = FP_Module([],[],char=char)
    M, incl, proj = M.copy()
    return M, [incl],[proj]
      elif len(alist) == 1:
        M,incl,proj = alist[0].copy()
    return M,[incl],[proj]  # return lists
      else:
        if char == None:
            char = alist[0].char
    else:
        if alist[0].char != char:
            raise ValueError,\
      "Characteristic passed (%d) differs from characteristic of
        module (%d)."\
        % (char, alist[0].char)
    alg = SteenrodAlgebra(p = alist[0].char,profile=\
                  enveloping_profile_profiles([x.profile() for x in
                      alist],char))
        listdegs = reduce(lambda x,y: x+y, [x.degs for x in alist])
        numdegs = len(listdegs)
        listrels = [[Integer(0) for i in range(numdegs)] for k in
            range(\
                reduce(lambda x,y: x+y, [len(x.rels) for x in
                    alist]))] #initialize rels
        # now scan through each module in sum, each list of rels
            for each module, and
        # each rel in each list of rels. modn keeps track of each
            module, reln keeps
        # track of the rel.
    ## Hard-coded sage Integer(0), else raises ERROR above in e_p_e.
        modn = 0
        reln = 0
```

```
            for i in range(len(alist)):
                for j in range(len(alist[i].rels)):
              for k in range(len(alist[i].rels[j])):
            listrels[modn+j][reln+k] = alist[i].rels[j][k]
          modn += len(alist[i].rels)
          reln += len(alist[i].degs)
    M = FP_Module(listdegs,listrels, algebra = alg)
            incl = len(alist)*[[]]
    proj = len(alist)*[[]]
    posn = 0
    for i in range(len(alist)):
                incl[i] = FP_Hom(alist[i],M,[M.gen(posn+j)\
               for j in range(len(alist[i].degs))])
                pvals = numdegs*[0]
          for j in range(len(alist[i].degs)):
              pvals[posn+j] = alist[i].gen(j)
                proj[i] = FP_Hom(M,alist[i],pvals)
          posn += len(alist[i].degs)    #### FIX. was:  posn += len(
              alist[i].rels)
    return M, incl, proj

#-----------------------------Functions-for-Resolutions
    ------------------------------------------------


  def lift(f,g):
      """
      Computes the lift of f over g, so that g*lift(f,g) = f.
      All maps taken to be FP_Homs. If lift doesn't exist, False is
      returned with the 0 map.

      INPUT:

      -  `f` - The map to be lifted over.

      -  `g` - The map lifted over.
```

```
    OUTPUT: The lift of f over g.

    EXAMPLES::



    """
    if f.codomain != g.codomain:
        raise TypeError, "Cannot lift maps with different codomains
            ."
    vals = []
    cando = true
    for x in f.domain.gens():
  if cando:
            newval = g.solve(f(x))
      cando = cando and newval[0]
            vals.append(newval[1])
    if cando:
        return true,FP_Hom(f.domain,g.domain,vals)
    else:
        return false,FP_Hom(f.domain,g.domain,0)



def Homology(f,g):
    """
    Computes the Homology of a pair of maps.

    INPUT:

    - ``f`` - The FP_Hom corresponding to the first map of the
        composition.

    - ``g`` - The second (or last) FP_Hom in the composition.

    OUTPUT:
```

```
    - ``H`` - The quotient `Ker(g)/Im(f)`

    - ``p`` - The map from `Ker(g)` to `H`

    - ``i`` - The inclusion of `Ker(g)` into domain of `g`.

    - ``m`` - The inclusion of `Im(f)` into the domain of `g`.

    - ``l`` - The lift of `m` over `i`.


    """
    K,i = g.kernel()
    I,e,m = f.image()
    l = lift(m,i)[1]  # we want the map, not the bool
    H,p = l.cokernel()
    return H,p,i,m,l


def extend_resolution_kernels(R,n,verbose=false):
    """
    Extends a resolution `R` to length `n`.

    INPUT:

    - ``R`` - A list of FP_Homs, corresponding to a resolution.

    - ``n`` - The length to which the resolution will be extended
      to.

    OUTPUT: A list of FP_Homs, corresponding to the extended
        resolution.

    EXAMPLES::
```

```

        """
        if n < len(R):
            return R
        if verbose:
            print "Step ",1+n-len(R)
        K,i = R[-1][1]
        kers = [R[i][1] for i in range(len(R))]
        r,k = K.resolution_kernels(n-len(R),kers,verbose=verbose)
        r[0] = i*r[0]
        return R + r, kers

    def extend_resolution(R,n,verbose=false):
        """
        Extends a resolution `R` to length `n`.

        INPUT:

        - ``R`` - A list of FP_Homs, corresponding to a resolution.

        - ``n`` - The length to which the resolution will be extended
          to.

        OUTPUT: A list of FP_Homs, corresponding to the extended
            resolution.

        EXAMPLES::


        """
        if n < len(R):
            return R
        if verbose:
            print "Step ",1+n-len(R)
```

```
        K, i = R[-1].kernel()
        r = K.resolution(n-len(R), verbose=verbose)
        r[0] = i*r[0]
        return R + r

    def is_complex(R):
        """
        Determines whether a resolution `R` is a valid chain complex.

        INPUT:

        - ``R`` - A list of FP_Homs, forming a resolution.

        OUTPUT: True if `R` corresponds to a complex, false otherwise.

        EXAMPLES::



        """
        val = True
        i = 1
        while val and i < len(R):
            val = val and (R[-i-1]*R[-i]).is_zero()
            i += 1
        return val

    def is_exact(R):
        """
        Deteremines whether a resolution `R` is exact.

        INPUT:

        - ``R`` - A list of FP_Homs, forming a resolution.

```

```
    OUTPUT: True if the list of FP_Homs passed is exact, false
        otherwise.

    EXAMPLES::

    """
    if not is_complex(R):
        return False
    val = True
    i = 0
    while val and i < len(R)-1:
        K = R[i].kernel_gens()[0]
  print i
  for x in K.gens():
      val = val and R[i+1].solve(x)[0]
      if not val:
          break
        i += 1
    return val

def chain_map(L,R,f):
    """
    Computes the lift of an FP_Hom over a resolution. `L` and `R`
        are
    resolutions, and `f` is an FP_Hom from L[0].codomain to R[0].
        codomain.

    INPUT:

    - ``L`` - A list of FP_Homs, corresponding to a resolution.

    - ``R`` - A list of FP_Homs, corresponding to a resolution.
    """
    if len(L) == 0 or len(R) == 0:
        return [f]
```

```
    l = lift(f*L[0],R[0])[1]
    i = 0
    Z = [f]  ## can't get Z = ([f]).append(l) to work
    Z.append(l)
    for i in range(1,min(len(L),len(R))):
        Z.append(lift(Z[i]*L[i],R[i])[1])
    return Z

def extension(R,e,test=True):
    """
    Computes the module M corresponding to the presumed cocycle 'e
        '.
    R must be a resolution of length at least 3, and e must be a
        cocycle.
    The checks of these can be bypassed by passing test=False.
    """
    if test == True:
  if len(R) < 3:
      raise ValueError, "Resolution not of length at least 3"
  if not is_exact([R[0],R[1],R[2]]):
      raise TypeError, "Not a valid resolution"
        if not (e*R[2]).is_zero():
      raise TypeError, "Not a cocycle"
    M,I,P = DirectSum([R[0].domain(),e.codomain()])
    C,p = (I[0]*R[1] - I[1]*e).cokernel()
    N,q = (p*I[1]).cokernel()
    v = [R[0].solve(g)[1] for g in R[0].codomain().gens()]
    g = FP_Hom(R[0].codomain(),N,[(q*P*I[0])(x) for x in v])
    j = lift(q,g)
    return M,p*I[1],j
```

```
#------------------------------Profile-Functions----------------------------------------

## Palmieri pads all his profiles by adding a [0] to the beginning.
# Furthermore, when scanning over elements defined over an odd
# primary Steenrod Algebra, the Q part comes first, and then the P part.
# When defining a sub-algebra over an odd Steenrod algebra, the P part
# must come first in the profile.

# These functions return the P part first, and then the Q part.

from sage.algebras.steenrod.steenrod_algebra_misc import *

def profile_ele(alist,char=2):
    """
    Finds the smallest sub-Hopf algebra containing the element passed.
    `alist' is assumed to be an element of the Steenrod Algebra, (the only other
    possible cases are dealt with immediately) so it is treated as a list.

    INPUT:

    - ``alist`` - An element of the Steenrod Algebra (or a sub-Hopf algebra
                of it). Treated as a list.

    OUTPUT: The profile function corresponding to the smallest sub-Hopf algebra
    containing the element passed.

    EXAMPLES::
```

```
        sage: A2 = SteenrodAlgebra(2)
        sage: profile_ele(A2.Sq(2))
        (2, 1)
        sage: profile_ele(A2.Sq(4,8))
    (3, 4, 3, 2, 1)



    """
    if char == 2:
        if type(alist) == type(ZZ(1)) or\
           type(alist) == type(1) or\
           type(alist) == type(GF(2)(1)):
            return (0,)                            ## Better: convert
                to a Palmieri profile
        alist2 = [e[0] for e in alist]
        if not alist2:  ## What is this checking for?
            return (0,)
        maxlength = max([len(e) for e in alist2])
        alist2 = [list(e) + (maxlength-len(e))*[0] for e in alist2]
        minprofile = [max([alist2[i][j] for i in range(len(alist2))
            ]) \
                                                for j in range(
                                                    maxlength)]
        minprofile = tuple(map(lambda xx: mod_p_log(xx,char),
            minprofile))
        return find_min_profile(minprofile,char)
    if char != 2:  # any odd.    FIX THESE CHECKS
        alistQ = [e[0][0] for e in alist]
    alistP = [e[0][1] for e in alist]
    # print "alist = ", alist
    # print "alistQ = ", alistQ
    # print "alistP = ", alistP
        if not alistQ[0] and alistP[0]:    ## No Q, only P
      maxlengthP = max([len(e) for e in alistP])
```

```
            alistP = [list(e) + (maxlengthP-len(e))*[0] for e in
                alistP]
            minprofileP = [max([alistP[i][j] for i in range(len(
                alistP))]) \
                                                    for j in range(
                                                        maxlengthP)]
            minprofileP = tuple(map(lambda xx: mod_p_log(xx,char),
                minprofileP))
        return find_min_profile((minprofileP,()),char = char)
    elif alistQ[0] and not alistP[0]:
            maxlengthQ = max([len(e) for e in alistQ])
            alistQ = [list(e) + (maxlengthQ-len(e))*[0] for e in
                alistQ]
            minprofileQ = [max([alistQ[i][j] for i in range(len(
                alistQ))]) \
                                                    for j in range(
                                                        maxlengthQ)]
        minpQ = [1]*(max(minprofileQ)+1)
          for j in minprofileQ:
            minpQ[j] = 2
            return find_min_profile(((0,),minpQ),char = char)
          elif not alistQ[0] and not alistP[0]:
        return ((0,),())
          else:
            maxlengthQ = max([len(e) for e in alistQ])
            maxlengthP = max([len(e) for e in alistP])
            alistQ = [list(e) + (maxlengthQ-len(e))*[0] for e in
                alistQ]
            alistP = [list(e) + (maxlengthP-len(e))*[0] for e in
                alistP]
            minprofileQ = [max([alistQ[i][j] for i in range(len(
                alistQ))]) \
                                                    for j in range(
                                                        maxlengthQ)]
```

```
            minprofileP = [max([alistP[i][j] for i in range(len(
                alistP))]) \
                                                    for j in range(
                                                        maxlengthP)]
            minprofileP = tuple(map(lambda xx: mod_p_log(xx,char),
                minprofileP))
        minpQ = [1]*(max(minprofileQ)+1)
        for j in minprofileQ:
            minpQ[j] = 2
            return find_min_profile((minprofileP,minpQ),char=char)

#           return find_min_profile(minprofile,p)

def enveloping_profile_elements(alist,char=2):
    """
    Finds the profile function for the smallest sub-Hopf algebra
        containing
    the list of elements passed. Entries of `alist' are elements of
         the Steenrod
    Algebra. Hence, each alist[i] is treated as a list. Accepts
        either a list of
    lists or tuples.

    INPUT:

    - ``alist`` - A list of Steenrod Algebra elements.

    OUTPUT: The profile function for the minimum sub-algebra
        containing all the
    elements of `alist`.

    EXAMPLES::

        sage: enveloping_profile_elements([Sq(2),Sq(4)])
  (3, 2, 1)
```

```
sage: enveloping_profile_elements([Sq(2,1,2),Sq(7)])
(3, 2, 2, 1)

  """
  if char == 2:
      alist2 = list(map(profile_ele,[x for x in alist if  x !=
          0])) ## ERROR?
      if not alist2: # Hard coded
           return (0,)
      maxlength = max([len(e) for e in alist2])
      alist2 = [list(e) + (maxlength-len(e))*[0] for e in alist2]
      minprofile = tuple(max([alist2[i][j] for i in range(len(
          alist2))]) \
                                              for j in range(
                                                  maxlength))
      return find_min_profile(minprofile)
  else:  # odd primes
      masterlist = [profile_ele(x,char) for x in alist if x != 0]
## ^^ Not the correct check, != 0
      alistP = [x[0] for x in masterlist]
alistQ = [x[1] for x in masterlist]
      if not alistP and not alistQ:
    return ((0,),(0,))
      maxlengthQ = max([len(e) for e in alistQ])
maxlengthP = max([len(e) for e in alistP])
      alistQ = [list(e) + (maxlengthQ-len(e))*[0] for e in alistQ
          ]
      alistP = [list(e) + (maxlengthP-len(e))*[0] for e in alistP
          ]
      minprofileQ = tuple(max([alistQ[i][j] for i in range(len(
          alistQ))]) \
                                              for j in range(
                                                  maxlengthQ))
      minprofileP = tuple(max([alistP[i][j] for i in range(len(
          alistP))])\
```

```
                                    for j in range(maxlengthP
                                        ))
    return find_min_profile((minprofileP,minprofileQ),char=char)

def enveloping_profile_profiles(alist,char=2):
    """
    Finds the profile function for the smallest sub-Hopf algebra
        containing
    the sub-algebras corresponding the list of profile functions
        passed. Accepts
    either a list of lists or tuples.

    INPUT:

    - ``alist`` - A list of profile functions.

    OUTPUT: The profile function for the minimum sub-algebra
        containing
    the profile functions in `alist`.

    EXAMPLES::

        sage: enveloping_profile_profiles([[1,2,3],[2,4,1,1]])
  (2, 4, 3, 2, 1)
  sage: enveloping_profile_profiles([[4],[1,2,1],[3,2,3]])
  (4, 3, 3, 2, 1)

    """
    if char == 2:
        alist2 = list(copy(alist))
        maxlength = max([len(e) for e in alist2])
        alist2 = [list(e) + (maxlength-len(e))*[0] for e in alist2]
        minprofile = tuple(max([alist2[i][j] for i in range(len(
            alist2))]) \
```

```
                                                    for j in range(maxlength))
        return find_min_profile(minprofile)
    else:
        alistP = [copy(alist[i][0]) for i in range(len(alist))]
        alistQ = [copy(alist[i][1]) for i in range(len(alist))]
        maxlengthQ = max([len(e) for e in alistQ])
        maxlengthP = max([len(e) for e in alistP])
        alistQ = [list(e) + (maxlengthQ-len(e))*[0] for e in alistQ]
        alistP = [list(e) + (maxlengthP-len(e))*[0] for e in alistP]
        minprofileQ = tuple(max([alistQ[i][j] for i in range(len(alistQ))]) \
                                                    for j in range(maxlengthQ))
        minprofileP = tuple(max([alistP[i][j] for i in range(len(alistP))]) \
                                                    for j in range(maxlengthP))
        return find_min_profile((minprofileP,minprofileQ),char=char)


def valid(LL,char=2):
    """
    Determines if the pseudo-profile passed is a valid profile.
    ## When checking at odd primes, the `P`-part must be the 0th entry in LL,
    and the `Q`-part must be the 1st entry in LL.

    INPUT:

    - ``LL`` - A list of non-negative integers.

```

```
    OUTPUT: True or False, depending on whether the list passed is
        a valid profile.

    EXAMPLES::

        sage: valid([3,2,1])
  True
  sage: valid([1,2,3])
  False

    """
    if char == 2:
        L = [0] + list(LL) + [0]*(len(LL)) # Add 0 in beginning to
            keep rels correct
        value = true                       # so r - i works when i
            = 0
        for r in range(2,len(L)):
            for i in range(1,r):
                value = value and ((L[r] >= L[r-i] - i) or (L[r] >=
                    L[i]))
        return value
    else:
        (alistP,alistQ) = (LL[0],LL[1])
  M = [0] + list(alistP) + [0]*len(alistP)
  L = list(alistQ) + [1]*(len(alistQ)+1)
  M = M + [0]*abs(len(M)-len(L))  # Pad so they're the same length,
      then the lemmas apply.
  L = L + [1]*abs(len(M)-len(L))
  value = valid(alistP,char=2)  # P part must satisfy same
      conditions, regardless of prime.
  for r in range(len(L)): # \tau's indexed at 0, unlike \xi 's
      if (L[r] == 1) and value:
              for i in range(r+1):
        value = value and ((M[i] <= r -i) or (L[r-i] == 1))
        return value
```

```
def nextprof(p,n,char=2):
    """
    Takes a possible profile 'p' and a base profile 'n'. Returns
        the next
    profile in lexicographic order. After all valid profiles 'p' of
    length = len(n) have been checked, n is increased. Intended for
         internal
    use only. The odd primary piece only alters the Q part of the
        profile. To
    increment the P part of a profile for odd primes, call nextprof
         with char =2.
    This works since the P part of the profile is identical to the
        profile
    function when char == 2.

    INPUT:

    -  ``p``  - A pseudo-profile function which is incremented
        lexicographically
    and checked for validity.

    -  ``n``  - The base pseudo-profile function.

    OUTPUT: The next lexicographic profile.

    EXAMPLES::

        sage: nextprof([1,2],[1,2])
  [2, 2]
  sage: nextprof([2,2],[1,2])
  [1, 2, 1]

  sage: nextprof([2,2,3],[1,2,3])
  [3, 2, 3]
```

```
  sage: nextprof([3,2,3],[1,2,3])
  [1, 3, 3]

    """
    if char == 2:
        for i in range(len(p)):
            if p[i] < len(p):     # we increment here because we can without altering length
                p[i] += 1
                return p
            else:
                if i > len(n)-1:                # Past end of n, so reset to 0
                    p[i] = 0
                else:                           # inside n still, so reset to n
                    p[i] = n[i]
        return n + [0]*(len(p)-len(n)) + [1]    # fell off the end
    else:  # odd primes
        pP,pQ,nP,nQ = list(p[0]),list(p[1]),list(n[0]),list(n[1])
        for i in range(len(pQ)):
      if pQ[i] < 2:
          pQ[i] += 1
    return pQ
      else:
          if i > len(nQ) -1:
        pQ[i] = 1
    else:
        pQ[i] = nQ[i]
  return nQ + [1]*(len(pQ)-len(nQ)) +[1]


def find_min_profile(prof,char=2):
    """
```

```
    Given a tuple of integers (a pseudo-profile), this function
        will
    output the smallest legal profile function containing it. This
    function combines the above functions, and is the only one
        intended
    for external use.

    INPUT:

    - ``prof`` - A list or tuple of nonnegative integers.

    OUTPUT:

    - ``p`` - A valid profile containing ``p``.

    EXAMPLES::

        sage: find_min_profile([1,2])
  (1, 2, 1)
  sage: find_min_profile([2,1])
  (2, 1)
  sage: find_min_profile([1,2,3])
  (1, 2, 3, 1, 1)

    """
    if char == 2:
        prof2 = list(prof)
        if not prof2:
            return (0,)
        r = 0
        for i in range(len(prof2)):
            if prof2[i] != 0:
                r = i
        n = [prof2[i] for i in range(r+1)]
        p = copy(list(n))
```

```
            while not valid(p,char):
                p = nextprof(p,n,char)
            return tuple(p)
        else:
            pP,pQ = list(prof[0]), list(prof[1])
    P = find_min_profile(pP,char=2)
    Q = copy(pQ)
    while not valid([P,Q],char):
        Q = nextprof([P,Q],[P,pQ],char)
    return (P,Q)

#           pQ,pP = prof[0],prof[1]
# r = 0
# for i in range(len(pP)):
#       if pP[i] != 0:
#             r = i
# norm = [pP[i] for i in range(r+1)]
# norm = copy(list(norm))
# while not valid(pP,char=2):
#       a = nextprof(a,ll,char)
# return tuple(a[0]),tuple(a[1])



#------------------------------------------------------------------------------------------

#------------------------Finitely-Presented-Modules
    -----------------------------------
#------------------------------------------------------------------------------------------


class FP_Module(SageObject):
    r"""
    A finitely presented module over the Steenrod Algebra. Also
        defined
```

```
    over a sub-Hopf Algebra over the Steenrod Algebra.
    """

    def __init__(self,degs,rels=[],char=None,algebra=None):
        r"""


        """

        if (char is None) and (algebra is None):
            self.char = 2
            self.algebra = SteenrodAlgebra(self.char, profile=(0,))
        elif (char is None) and (algebra is not None):
            self.algebra = algebra
            self.char = self.algebra._prime
        elif (char is not None) and (algebra is None):
            self.char = char
      if char == 2:
                self.algebra = SteenrodAlgebra(p=self.char, profile
                    =(0,))
      else:
          self.algebra = SteenrodAlgebra(p=self.char, profile = (()
              ,(0,)))
        else:
            self.char = char
            self.algebra = algebra
        if (self.char != self.algebra.prime()):
            raise TypeError, "Characteristic and algebra are
                incompatible."
  if degs != sorted(degs):
      raise TypeError, "Degrees of generators must be in non-
          decreasing order."
        if not rels:
            prof = self.algebra._profile
        else:
```

```
            prof = enveloping_profile_profiles(\
                    [enveloping_profile_elements(r,self.char) for
                        r in rels]\
                    +[list(self.algebra._profile)],self.char)
        self.algebra = SteenrodAlgebra(p=self.char,profile=prof)
for r in rels:  ## Added!!!!!
    if r == [0]*len(degs):
        rels.remove([0]*len(degs))
        self.rels = [[self.algebra(coeff) for coeff in r] for r in
            rels]
        self.degs = copy(degs)
        try:                        # Figure out if a rel isnt
            right
            self.reldegs = [_deg_(self.degs,r) for r in self.rels]
        except ValueError:
            for r in rels:          # Figure out which rel isnt
                right
                try:
                    _deg_(degs,r)
                except ValueError:
                    raise ValueError, "Inhomogeneous relation %s" %
                        r



    def profile(self):
        return self.algebra._profile

    def alg(self):
        return self.algebra

    def conn(self):
        return min(self.degs+[+infinity])

    def __contains__(self,x):
```

```
        r"""
        Returns true if `x` is contained in the module.

        INPUT:

        -   ``x`` - some element

        OUTPUT: True if is in the module.

        EXAMPLES::

            sage: M = FP_Module([0,2],[[Sq(3),Sq(1)]])
            sage: m = FP_Element([Sq(2),Sq(1)],M)
            sage: M.__contains__(m)
            True
        """
        return x.module == self


    def _repr_(self):
        """
        String representation of the module.

        OUTPUT: string

        EXAMPLES::

            sage: M = FP_Module([0,2,4],[[Sq(4),Sq(2),0]]); M
      Finitely presented module on 3 generators and 1 relations
          over sub-Hopf
      algebra of mod 2 Steenrod algebra, milnor basis, profile
          function [3, 2, 1]

      sage: N = FP_Module([0,1],[[Sq(2),Sq(1)],[Sq(2)*Sq(1),Sq(2)
          ]]); N
```

```
    Finitely presented module on 2 generators and 2 relations
        over sub-Hopf
    algebra of mod 2 Steenrod algebra, milnor basis, profile
        function [2, 1]

      """
      return "Finitely presented module on %s generators and %s
          relations over %s"\
                      %(len(self.degs),len(self.rels),self.
                          algebra)

  def __call__(self,x):
      """
      Forms the element with ith coefficient x[i].
The identity operation if x is already in the module.

      INPUT:

      -   ``x``  - A list of coefficient.

      OUTPUT: An FP_Element with coefficients from x.

EXAMPLES::

    sage: M = FP_Module([0,2,4],[[Sq(4),Sq(2),0]]); M([Sq(2)
        ,0,0])
    [Sq(2), 0, 0]

      """
      if x == 0:
          return FP_Element([ 0 for i in self.degs],self)
elif type(x) == type([0]):
          return FP_Element(x,self)
```

```
    elif x.module == self: ## Is this handled in isinstance in
        __init__?
        return x
    else:
        raise ValueError,"Element not in module"


  def _pres_(self,n,profile=None):          # prof FIX
      """
      Returns a vector space, a quotient map, and elements.
          Internal use only.

      INPUT:

      -    ``n`` - The degree in which all computations are made
          .


      OUTPUT:


      -  ``quo``  - A vector space for the degree `n` part of
          Module.


      -  ``q``  - The quotient map from the vector space for the
          free module on
         the generators to quo.


      -  ``sec``  - Elements of the domain of `q` which project
          to the std basis for
         quo.


      -  `` bas_gen``  - A list of pairs (gen_number, algebra
          element)
         corresponding to the std basis for the free module.

      EXAMPLES::
```

```
            sage:
        """
    if profile == None:
       profile = self.profile()
    alg = SteenrodAlgebra(p=self.char, profile=profile)
        bas_gen = reduce(lambda x,y : x+y,\
                [[(i,bb) for bb in alg.basis(n-self.degs[i])]\
                      for i in range(len(self.degs))],[])
        bas_vec = VectorSpace(GF(self.char),len(bas_gen))
        bas_dict = dict(zip(bas_gen,bas_vec.basis()))
        rel_vec = bas_vec.subspace([0])
        for i in range(len(self.rels)):
            if self.reldegs[i] <= n:
                for co in alg.basis(n-self.reldegs[i]):
                    r = zip(range(len(self.degs)),[co*c for c in
                        self.rels[i]])
                    r = filter(lambda x : not x[1].is_zero(),r) #
                        remove trivial
                    if len(r) != 0:
                        r = reduce(lambda x,y : x+y,
                            [map(lambda xx: (pr[0],alg.
                                _milnor_on_basis(xx[0]),xx[1]),\
                                 [z for z in pr[1]]) for pr in r])
                        rel_vec += bas_vec.subspace(\
                            [reduce(lambda x,y: x+y,\
                            map(lambda x: x[2]*bas_dict[(x[0],x[1])
                                ],r))])
        quo = bas_vec/rel_vec
        if quo.dimension() == 0:  # trivial case
            sec = Hom(quo,bas_vec)(0)
            q = Hom(bas_vec,quo)([quo(0) for xx in bas_vec.basis()
                ])
        else:
```

```
            sec = Hom(quo,bas_vec)([quo.lift(xx) for xx in quo.basis()])
            q = Hom(bas_vec,quo)([quo(xx) for xx in bas_vec.basis()])
        return quo, q, sec, bas_gen

    def _lc_(self,co,bas):
        """
        INPUT:

        -    ``co``   -  A list of (either GF(p) elements or algebra elements)
              coefficients.

        -    ``bas``   -  A list of tuples (gen_number, algebra elt)
              corresponding to the std basis for the free module on self.degs

       OUTPUT: The linear combination given by the sum of co[i]*bas[i][1]*gen(bas[i][0])

       NOTE: The list of coefficients can lie in GF(p) or the algebra.
              This does not normalize, the sum is taken in the free module.

        EXAMPLES::

            sage:
        """
        if len(co) != len(bas):
            raise ValueError,\
            "Number of coefficients (%s) must be the same as number of basis elements (%s) " \
```

```
            % (len(co),len(bas))
        return reduce(lambda x,y : x+y, \
            [self.gen(bas[i][0])*(co[i]*bas[i][1]) for i in range
                (len(co))],
        self(0))

    def basis(self,n,profile=None):         ## prof FIX
    """
    Returns elements of the free module mapping to self. These
        elements
        form a basis for the degree n piece of the module.

    INPUT:

        - ``n`` - The degree in which all computations are
            made

        OUTPUT: A list of elements forming a basis for the degree n
            part of the
                module.

        EXAMPLES::

            sage:
    """
##  if profile == None:
##      if self.char == 2:
##          profile = ()    ## what to change this to?
##      else:
##          profile = ((),())
        if profile == None:
            profile = self.profile()
    quo,q,s,bas = self._pres_(n,profile=profile)
    return [self._lc_(s(v),bas) for v in quo.basis()]
```

```
    __getitem__ = basis

    def gens(self):
    """
    Returns the list of generators of the module.

    OUTPUT: A list corresponding to the generators of the module.
    """
    return [FP_Element(_del_(i,len(self.degs)),self) \
             for i in range(len(self.degs))]

    def gen(self,i=0):
    """
    Returns the `i^{th}` generator of the module as an FP_Element

        ## return gens()[i]?
    """
    if i < 0 or i >= len(self.degs):
              raise ValueError,\
              "Module has generators numbered 0 to %s; generator %s
                  does not exist" % (len(self.degs)-1,i)
    return FP_Element(_del_(i,len(self.degs)),self)


    def identity(self):
        """
        Returns the identity homomorphism of the module.

        OUTPUT: The identity homomorphism of the module as a
            finitely
                presented homomorphism.

        EXAMPLES::

            sage :
```

```
        """
        return FP_Hom(self,self,[_del_(i,len(self.degs))\
                             for i in range(len(self.degs))])

    def min_pres(self):
        """
    Returns the minimal presentation of the module, along with maps
    between min_pres and self.
    """
    M,e,i = self.identity().image()
    return M,e,i

    def min_profile(self):
        """
    Returns the profile of the smallest sub-Hopf algebra containing
        self.

    OUTPUT: The profile function of the sub-Hopf algebra with the
        smallest
    degree containing self.

    EXAMPLES::


    """
        if not self.rels:
            return self.algebra._profile
        else:
            profile = enveloping_profile_profiles(\
                       [enveloping_profile_elements(r,self.char) for
                           r in self.rels],\
          self.char)
            return profile
```

```
    def copy(self):
        """
        Returns a copy of the module, with 2 ``identity'' morphisms
            from
        1. the copy to the module
        2. the module to the copy.

        OUTPUT:

        -    ``C`` - A duplicate of the module.

        -    Two Finitely Presented Homomorphisms, the first is a
             map from `C` to self,
             and the second is the map from self to `C`.

        EXAMPLES::

             sage: M = FP_Module([0,4],[[Sq(1),0],[Sq(5),Sq(1)]])
        sage: N,i,p = M.copy(); N,i,p
        (Finitely presented module on 2 generators and 2 relations
            over sub-Hopf
        algebra of mod 2 Steenrod algebra, milnor basis, profile
            function [3, 2, 1],
        Homomorphism from
        Finitely presented module on 2 generators and 2 relations
            over sub-Hopf
        algebra of mod 2 Steenrod algebra, milnor basis, profile
            function [3, 2, 1] to
        Finitely presented module on 2 generators and 2 relations
            over sub-Hopf
        algebra of mod 2 Steenrod algebra, milnor basis, profile
            function [3, 2, 1]
        , Homomorphism from
```

```
    Finitely presented module on 2 generators and 2 relations
        over sub-Hopf
    algebra of mod 2 Steenrod algebra, milnor basis, profile
        function [3, 2, 1]
    to
    Finitely presented module on 2 generators and 2 relations
        over sub-Hopf
    algebra of mod 2 Steenrod algebra, milnor basis, profile
        function [3, 2, 1]
    )

      """
      C = FP_Module(self.degs, self.rels, algebra=self.algebra)
      return C,\
        FP_Hom(C,self,[_del_(i,len(self.degs))\
                for i in range(len(self.degs))]),\
        FP_Hom(self,C,[_del_(i,len(self.degs))\
                for i in range(len(self.degs))])

  def suspension(self,t):
      """
      Suspends a module by degree t.

INPUT:

-    ``t``  - An integer by which the module is suspended.

OUTPUT:

-    ``C``  ` A copy of the module `self` which is suspended by `t
     `.

      EXAMPLES::

    sage:
```

```
    """
    if t == 0:
        return self
    else:
        C = self.copy()[0]
        C.degs = map(lambda x: x+t, C.degs)
        C.reldegs = map(lambda x: x+t, C.reldegs)
        return C


  def submodule(self,L):
      """
    Returns the submodule of self spanned by elements of the list L,
    together with the map from the free module on the elements of L
        to
    the submodule, and the inclusion of the submodule.

    INPUT:

    -  ``L`` - A list of elements of `self`.

    OUTPUT: The submodule of `self` spanned by elements of `L`.

    EXAMPLES::
    """
        F = FP_Module([x.degree for x in L],algebra=self.algebra)
    pr = FP_Hom(F,self,L)
    N,p,i = pr.image()
    return N,p,i


  def resolution(self,k,verbose=false):
      """
    Returns a list of length `k`, consisting of chain maps. These
    maps form a resolution of length `k` of `self`.
```

```
"""
        C0 = FP_Module(self.degs, algebra=self.algebra)
eps = FP_Hom(C0,self,self.gens())
if verbose:
        print "Step ",k
if k <= 0:
    return [eps]
else:
    K0,i0 = eps.kernel()
    r = K0.resolution(k-1,verbose=verbose)
    r[0] = i0*r[0]
    return [eps] + r


  def resolution_kernels(self,k,kers=[],verbose=false):
        """
Returns a list of length `k`, consisting of chain maps and
a list of pairs [K_n,i_n] corresponding to the kernels
and inclusions of the resolution. These
maps form a resolution of length `k` of `self`.

A list should never be passed for kers. This is only used
for recursion.
"""
        C0 = FP_Module(self.degs, algebra=self.algebra)
eps = FP_Hom(C0,self,self.gens())
if verbose:
        print "Step ",k
if k <= 0:
    return [eps],kers
else:
    K0,i0 = eps.kernel()
    kers.append([K0,i0])
    r,k = K0.resolution_kernels(k-1,kers,verbose=verbose)
    r[0] = i0*r[0]
```

```
        return [eps] + r, kers

    # L = [eps]
    # ker,incl = eps.kernel()
            # kerlist = [ker]
    # incllist = [incl]
    # for i in range(k):
        # Ci = FP_Module(kerlist[i-1].degs)
        # print i
            # print [x for x in kerlist[i].gens()]
        # di = FP_Hom(Ci,L[i-1].domain,[incllist[i](x) for x in
            kerlist[i].gens()])
        # Ki,incli = di.kernel()
        # kerlist.append(Ki)
        # incllist.append(incli)
        # L.append(di)
    # return L

#     def _coerce_map_from(self,S):
#

#------------------------------------------------------------------------------

#----------------------Homomorphisms-between-FP_Modules
    ----------------------------
#------------------------------------------------------------------------------


class FP_Hom(Morphism):
    r"""
    A finitely presented Homomorphism between two Finitely
        Presented Modules.
    If degree is passed, dom is suspended by degree before mapping.
    The 0 hom can be created by passing '0' for values.
```

```
    """

    def __init__(self,domain,codomain,values, degree=0):
        if domain.algebra.prime() != codomain.algebra.prime():
            raise ValueError,\
                "Domain algebra defined at the prime %s but codomain
                    algebra defined at prime %s"\
                    %(domain.algebra._prime, codomain.algebra._prime
                        )
        domain = domain.suspension(degree)
  if values == 0:
            values = [FP_Element([codomain.algebra(0) for j in
                codomain.degs],\
                        codomain) for i in domain.degs]
        if len(values) != len (domain.degs):
            raise ValueError,\
                "Domain has %s generators, but %s values were given
                    \,"\
                     %(len(domain.degs), len(values))
        for i in range(len(values)):
            if values[i] == 0:
                values[i] = FP_Element([codomain.algebra(0) for j
                    in\
                           codomain.degs],codomain)
            else:                                      # if its a
                list of coeffs, make it
                values[i]= FP_Element(values[i],codomain) # an
                    FP_Element.Otherwise ought to
                                                       # already be
                                                           one.
        self.values = [x.nf() for x in values]
  initialval = FP_Element([0]*len(domain.degs),domain)
        self.domain = domain
        self.codomain = codomain
        self.degree = degree
```

```
        if self.domain.rels:  ## Check like this, or other way? Rem
            'd != []
            for x in self.domain.rels:
                ximage = reduce(lambda xx,y: xx+y, [values[i]*x[i]
                    for i in\
                      range(len(x))])
                if not ximage.is_zero():
                    raise ValueError, "Relation %s is not sent to
                        0" % x
        prof = enveloping_profile_profiles([domain.profile(),
            codomain.profile(),\
                      enveloping_profile_elements(reduce(lambda x,y:
                          x+y,\
                            [x.coeffs for x in values],initialval.
                                coeffs),\
          domain.char)],domain.char)
        self.algebra = SteenrodAlgebra(p = domain.algebra.prime(),\
                        profile = prof)


    def profile(self):
        return self.algebra._profile


    def alg(self):
        return self.algebra

    def _repr_(self):
        return "Homomorphism from\n %s to\n %s\n" % (self.domain,
            self.codomain)


    def __add__(self,g):
        """
        Sum the homomorphisms, so (f+g)(x) == f(x)+g(x)
        """
        if self.domain != g.domain:
            raise ValueError,\
```

```
            "Morphisms do not have the same domain."
        if self.codomain != g.codomain:
            raise ValueError,\
            "Morphisms do not have the same codomain."
        if self.degree != g.degree:
            raise ValueError,\
            "Morphisms do not have the same degree."
        return FP_Hom(self.domain,self.codomain,\
                   [self(x)+g(x) for x in self.domain.gens()], self
                       .degree)

    def __neg__(self):
        return FP_Hom(self.domain,self.codomain,\
               [-self.values[i] for i in range(len(self.values))],
                   self.degree)

    def __sub__(self,g):
        return self.__add__(g.__neg__())

    def __mul__(self,g):
        """
        Composition of morphisms.
        """
        if self.domain != g.codomain:
            raise ValueError,\
                "Morphisms not composable."
        return FP_Hom(g.domain,self.codomain,\
                   [self(g(x)) for x in g.domain.gens()],self.
                       degree+g.degree)

    def is_zero(self):
        return reduce(lambda x,y: x and y, [x.is_zero() for x in
            self.values],True)

    def __cmp__(self,g): ## __hash__ ?
```

```
        if self.domain != g.domain:
            raise ValueError, "Morphisms not comparable, different
                domains."
        if (self-g).is_zero():
            return 0
        else:
            return 1



    def __call__(self,x):
        """
        Evaluate the morphism at an FP_Element of domain.

INPUT:

- ``x`` - An element of the domain of self.

OUTPUT: The FP_Hom evaluated at `x`.

EXAMPLES::



        """
        if x not in self.domain:
            raise ValueError,\
                  "Cannot evaluate morphism on element not in
                     domain"
        value = reduce(lambda x,y: x+y,\
                [self.values[i]*x.coeffs[i] for i in range(len(self
                   .domain.degs))],
    self.codomain(0))
return value.nf()

    def _full_pres_(self,n,profile=None):        # prof FIX
        """
```

```
        Computes the linear transformation from domain in degree n
    to codomain in degree n+degree(self). 9 items returned: the
        linear transformation, self.dom._pres_(n), & self.codomain.
            _pres_(n).
    See the documentation for _pres_ in class FP_Module for further
        explanation.

    INPUT:

    -  ``n``  - The degree in which all computations are made.

    -  ``profile``  - A profile function corresponding to the sub-
        Hopf algebra
         of the Steenrod Algebra for which this computation will be
             computed over.
         The default, `profile=None`, uses the profile of self.

    OUTPUT:

    -  The linear transformation corresponding to the degree `n`
        piece of this
       mapping (see the documentation for _pres_ below).

    -  ``dquo``  - The vector space corresponding to self.domain in
        degree `n`.

          -  ``dq``  - The quotient map from the vector space for the
               free module on
             the generators to `dquo`.

          -  ``dsec``  - Elements of the domain of `dq` which project
               to the standard
       basis for `dquo`.
```

```
        - ``dbas_gen`` - A list of pairs (gen_number, algebra
            element)
          corresponding to the standard basis for the free module.

    - ``cquo`` - The vector space corresponding to self.codomain in
        degree `n` +
      self.degree.

    - ``cq`` - The quotient map from the vector space for the free
        module on
      the generators to `cquo`.

    - ``csec`` - Elements of the domain of `cq` which project to
        the standard basis
      for `cquo`.

        - ``cbas_gen`` - A list of pairs (gen_number, algebra
            element) corresponding
      to the standard basis for the free module.

    EXAMPLES::

        sage:
          """
    if profile == None:
        profile = self.profile()
        dquo,dq,dsec,dbas_gen = self.domain._pres_(n,profile=
            profile)
        cquo,cq,csec,cbas_gen = self.codomain._pres_(n+self.degree,
            profile=profile)
        return Hom(dquo,cquo)(\
                    [cq(self(self.domain._lc_(dsec(x),dbas_gen)).
                        free_vec(profile=profile))\
                    for x in dquo.basis()]),\
                    dquo,dq,dsec,dbas_gen,\
```

```
                    cquo,cq,csec,cbas_gen

    def _pres_(self,n,profile=None):        # prof FIX
        """
        Computes the linear transformation from domain in degree n
            to
        codomain in degree n + degree(self). Intended for internal
            use only.

    INPUT:

    -    ``n`` - The degree in which all computations are made.

    -    ``profile`` - A profile function corresponding to the sub-
        Hopf algebra
         of the Steenrod Algebra for which this computation will be
            computed over.

    OUTPUT: The linear transformation from the degree `n` part of
        self.domain
            to the degree `n` + self.degree part of self.codomain.
                The basis for
      the vector space corresponding to the deg `n` piece of self.
          domain
      is mapped to the basis for the deg `n` + self.degree piece of
          self.codomain.

        EXAMPLES::

        sage:
        """
    return self._full_pres_(n,profile)[0]

    def min_profile(self):
        """
```

```
Returns the profile function for the smallest sub-Hopf algebra
    over which self
is defined.

This function is useful when reducing to the smallest profile
    function (and sub-Hopf algebra)
an FP_Module can be defined over.

OUTPUT:

- ``profile`` - The profile function corresponding to the
    smallest sub-Hopf algebra
                containing self.
"""
initialval = FP_Element([0]*len(self.domain.degs),self.domain)
      profile = enveloping_profile_profiles([self.domain.profile
          (),self.codomain.profile(),\
                    enveloping_profile_elements(reduce(lambda x,y:
                         x+y,\
                          [x.coeffs for x in self.values],
                              initialval.coeffs),\
       self.domain.char)], self.domain.char)
      return profile

  def suspension(self,t):
      """
Suspends an FP_Hom, which requires suspending the domain and
    codomain as well.

INPUT:

- ``t`` - The degree by which the homomorphism is suspended.

OUTPUT: The FP_Hom suspended by degree `t`.

```

```
    EXAMPLES::

        sage:
    """
    if t == 0:
        return self
    else:
        return FP_Hom(self.domain.suspension(t),\
                      self.codomain.suspension(t),\
                      self.values)

  def cokernel(self,minimal=''):
      """
    Computes the cokernel of an FP Hom.


      Cheap way of computing cokernel. Cokernel is on same degs
          as codomain,
      with rels = codomain.rels + self.values. Returns cokernel
          and the
      projection map to it.

    OUTPUT:

    - ``coker`` - The FP_Module corresponding to the cokernel of
        self.

    - The FP_Hom corresponding to the natural projection from self.
        codomain
      to `coker`.

      EXAMPLES::


      """
```

```
        coker = FP_Module(self.codomain.degs,\
                self.codomain.rels + [x.coeffs for x in self.values
                    ],\
                algebra = self.alg()) ## self.codomaina.alg()
        vals = [_del_(i,len(self.codomain.degs)) for i in \
                range(len(self.codomain.degs))]
                                            ### RRB: passing FP_Hom a
                                                  list
                 ### of coeffs, not FP_Elements.
  if minimal == '':
            return coker,FP_Hom(self.codomain,coker,vals)
  else:
      MM,e,m = coker.min_pres()
      p = FP_Hom(self.codomain,coker,vals)
      return MM, e*p



    def kernel(self):
        """
        Computes the kernel of an FP_Hom, as an FP_Module.
        The kernel is non-zero in degrees starting from
            connectivity of domain
        through the top degree of the algebra the function is
            defined over plus
        the top degree of the domain.

 OUTPUT:

  -  ``ker``  - An FP_Module corresponding to the kernel of `self`.

  -  ``incl``  - An FP_Hom corresponding to the natural inclusion
      of `ker`
                 into the domain.
```

```
        EXAMPLES::
        """
        n = self.domain.conn()
    if n == +Infinity:
        ker = FP_Module([])
        return ker, FP_Hom(ker,self.domain,values=0)
        notdone = True
        limit = max_deg(self.algebra) + max(self.domain.degs)
        while notdone and n <= limit:
            fn = self._pres_(n)
            notdone = (fn.kernel().dimension() == 0)
            if notdone:  # so the kernel is 0 in this degree n.
                Move on to the next.
                n += 1
        if notdone: # If the kernel is 0 in all degrees.
            ker = FP_Module([],[],algebra=self.alg())
            return ker, FP_Hom(ker,self.domain,values=0)
        else:
            ker = FP_Module(fn.kernel().dimension()*[n],[],algebra=
                self.alg())
            quo,q,sec,bas_gen = self.domain._pres_(n,profile=self.
                profile())
            incl = FP_Hom(ker,self.domain,\
                    [self.domain._lc_(sec(v),bas_gen) for v in fn.
                        kernel().basis()])
        n += 1
            while n <= limit:
                incln,Kn,p,sec,bas,Mn,q,s,Mbas_gen = incl.
                    _full_pres_(n)
                fn = self._pres_(n)
                if fn.kernel().dimension() != 0:  # so we found
                    something new
                    Kfn = VectorSpace(GF(self.domain.algebra._prime
                        ),\
                      fn.kernel().dimension())
```

```
                kin = Hom(Kfn,Mn)(fn.kernel().basis())
                jn = Hom(Kn,Kfn)(kin.matrix().solve_left(incln.matrix()))
                imjn = jn.image()
                num_new_gens = 0
                for v in Kfn.basis():  # we don't compute coker for new gens but
                    if not v in imjn:  # find enough basis vectors to complement image
                        num_new_gens += 1
                        imjn += Kfn.subspace([v])
                        incl.values.append(self.domain._lc_(s(kin(v)),Mbas_gen))
                ker.degs += num_new_gens*[n]
                pad = num_new_gens*[0]
                ker.rels = [x + copy(pad) for x in ker.rels]
            ker.rels += [ker._lc_(sec(v),bas).coeffs for v in incln.kernel().basis()]
            ker.reldegs += incln.kernel().dimension()*[n]
            n += 1
    # All generators have been found.  Now see if we need any more relations.
    while n <= max_deg(self.algebra) + max(ker.degs):
            incln,Kn,p,sec,bas,Mn,q,s,Mbas_gen = incl._full_pres_(n,profile=self.profile())
            ker.rels += [ker._lc_(sec(v),bas).coeffs for v in incln.kernel().basis()]
            ker.reldegs += incln.kernel().dimension()*[n]
            n += 1
        ker.algebra = SteenrodAlgebra(p=ker.char, profile = ker.min_profile())
    incl.algebra = SteenrodAlgebra(p=ker.char, profile = incl.min_profile())
        return ker, incl
```

```

    def kernel_gens(self):
        """
        Computes the generators of the kernel of an FP_Hom, and
            returns a free module
and an epi from it to the kernel of self as a map from the free
    module to self.domain.

        The kernel is non-zero in degrees starting from
            connectivity of domain
        through the top degree of the algebra the function is
            defined over plus
        the top degree of the domain.


OUTPUT:

- ``ker`` - A free FP_Module corresponding to the generators of
     the kernel of `self`.

- ``incl`` - An FP_Hom corresponding to the natural inclusion
    of `ker`
                 into the domain.

        EXAMPLES::
        """
        n = self.domain.conn()
        notdone = True
        limit = max_deg(self.algebra) + max(self.domain.degs)
        while notdone and n <= limit:
            fn = self._pres_(n)
            notdone = (fn.kernel().dimension() == 0)
            if notdone:  # so the kernel is 0 in this degree n.
                Move on to the next.
                n += 1
        if notdone: # If the kernel is 0 in all degrees.
```

```
            ker = FP_Module([],[],algebra=self.alg())
            return ker, FP_Hom(ker,self.domain,values=0) ## 0 = vals?
        else:
            ker = FP_Module(fn.kernel().dimension()*[n],[],algebra=self.alg())
            quo,q,sec,bas_gen = self.domain._pres_(n,profile=self.profile())
            incl = FP_Hom(ker,self.domain,\
                   [self.domain._lc_(sec(v),bas_gen) for v in fn.kernel().basis()])
      n += 1
            while n <= limit:
                incln,Kn,p,sec,bas,Mn,q,s,Mbas_gen = incl._full_pres_(n,profile=self.profile())
                fn = self._pres_(n)
                if fn.kernel().dimension() != 0:  # so we found something new
                    Kfn = VectorSpace(GF(self.domain.algebra._prime),\
                       fn.kernel().dimension())
                    kin = Hom(Kfn,Mn)(fn.kernel().basis())
                    jn = Hom(Kn,Kfn)(kin.matrix().solve_left(incln.matrix()))
                    imjn = jn.image()
                    num_new_gens = 0
                    for v in Kfn.basis():  # we don't compute coker for new gens but
                        if not v in imjn:  # find enough basis vectors to complement image
                            num_new_gens += 1
                            imjn += Kfn.subspace([v])
                            incl.values.append(self.domain._lc_(s(kin(v)),Mbas_gen))
                    ker.degs += num_new_gens*[n]
```

```
                n += 1
            ker.algebra = SteenrodAlgebra(p=ker.char, profile = ker
                .min_profile())
        incl.algebra = SteenrodAlgebra(p=ker.char, profile = incl.
            min_profile())
            return ker, incl


    def image(self):
        """
        Computes the Image of an FP_Hom, as an FP_Module. Returns
            the factorization of
        self into epi, Image, mono.

    Assumes generators of FP_Modules are in order of increasing
        degree.

    OUTPUT:

    - ``F`` - The FP_Module corresponding to the image of self.

    - ``mono`` - The FP_Hom corresponding to the natural inclusion
        of `F` into
                the codomain of self.

        - ``epi`` - The FP_Hom corresponding to the natural
            projection of the
                domain of self onto `F`.

        EXAMPLES::



        """
    ##F = FP_Module([self.domain.degs[0]],algebra=self.alg())
```

```
    ##mono = FP_Hom(F, self.codomain,[self.values[0]])
    ##epivals = [F.gen(0)]
    F = FP_Module([],algebra=self.alg())
    mono = FP_Hom(F, self.codomain,[])
    epivals = []
    #
          # Loop to find a minimal set of generators for the image
    #
    for i in range(len(self.domain.degs)):
        ##
        ## HERE IS WHERE WE NEED TO FIX THINGS
        ##
        n = self.domain.degs[i]
              pn,Fquo,Fq,Fsec,Fbas,Cquo,Cq,Csec,Cbas = mono.
                  _full_pres_(n,profile=self.profile())
        v = self.values[i].vec(profile=self.profile())[0]
        if Cquo(v) in pn.image():
            y = pn.matrix().solve_left(Cquo(v))
      # Now convert the vector y into an FP_Element using lc
      epivals.append(F._lc_(Fsec(y),Fbas))
        else:
            F.degs.append(n)
      epivals.append(F.gen(len(F.degs)-1))
      mono.values.append(self.values[i])
    # Now compute the relations
    #
    K,i = mono.kernel()
    F.reldegs = K.degs
    F.rels = [x.coeffs for x in i.values]
    l = len(F.degs)
    epivals = [ F(x.coeffs + [0]*(l-len(x.coeffs))) for x in epivals]
    epi = FP_Hom(self.domain,F,epivals)
    # Now reduce profile functions
    F.algebra = SteenrodAlgebra(p=F.char, profile = F.min_profile())
```

```
    mono.algebra = SteenrodAlgebra(p=F.char, profile = mono.
        min_profile())
    epi.algebra = SteenrodAlgebra(p=F.char, profile = epi.min_profile
        ())
    return F, epi, mono

    def solve(self, x):
        """
     Computes the element in self.domain, such that self(y) = x

     INPUT:

     - ``x`` - The element to be solved for.

     OUTPUT:

     - A boolean corresponding to whether or not the equation can be
         solved.

     - The element which maps to x under self.

     EXAMPLES::
     """
     pn, dquo, dq, dsec, dbas, cquo, cq, csec, cbas =\
                   self._full_pres_(x.degree, profile=self.profile())
     v = x.vec()[0]
     if v not in pn.image():
         return False, self.domain(0)
     else:
         w = pn.matrix().solve_left(v)
         return True, self.domain._lc_(dsec(w), dbas)
```

```
#------------------------------------------------------------------------------
#-------------------------Elements-of-FP_Modules--------------------------------------
#------------------------------------------------------------------------------

class FP_Element(ModuleElement):
    r"""
    Yields an element of an FP_Module, given by defining the coefficients on each
    generator of the module.
    ## Since we're overloading an already defined class (ModuleElement), we should
    ## use single underscores. This can't be implemented until parenting and
    ## coercion is figured out, however.

    *** Do we really need FP_Elements to have profiles?
    ***
    """

    def __init__(self, coeffs, module):
        """
        Defines an element of a Finitely Presented module.

  INPUT:

  -  ``coeffs``  - A list of Steenrod Algebra elements of GF(p)
                  coefficients.

  -  ``module``  - An FP_Module corresponding to the parent module.

  OUTPUT: The FP_Element defined by the sum over `i` of coeffs[i]*module.gen(i).
```

```
        Users can also define elements using the call() method of FP_Modules. See
    that function for documentation.

    EXAMPLES::

    sage: m = FP_Element([0,Sq(3),Sq(1)],FP_Module([2,3,5]));m
    [0, Sq(3), Sq(1)]

    """
        self.module = module
        if isinstance(coeffs,FP_Element): ## is this implementation correct?
            self.coeffs = coeffs.coeffs
        else:
            self.coeffs = [SteenrodAlgebra(module.algebra._prime)(x) for x in coeffs]
        self._parent = module
        self.degree = _deg_(self.module.degs,self.coeffs) # degree will find errors passed
        profile_coeffs = [profile_ele(j,self.module.char) for j in self.coeffs]
        self.profile = enveloping_profile_profiles(\
                [list(module.algebra._profile)]+profile_coeffs,
                    self.module.char)

    def __iadd__(self,y):
        if self.module != y.module:
            raise TypeError, "Can't add element in different modules"
        if self.degree == None: # if self = 0, degree is undef
            return y
        if y.degree == None:    # if y = 0, degree is undef
            return
```

```
        if self.degree != y.degree:
            raise ValueError, "Can't add element of degree %s and %s"\
                %(self.degree,y.degree)
        return FP_Element([self.coeffs[i]+y.coeffs[i] for i in range(len(self.coeffs))],self.module)

    def __add__(self,y):
        if self.module != y.module:
            raise TypeError, "Can't add element in different modules"
        if self.degree == None: # if self = 0, degree is undef
            return FP_Element(y.coeffs,y.module)
        if y.degree == None:    # if y = 0, degree is undef
            return FP_Element(self.coeffs, self.module)
        if self.degree != y.degree:
            raise ValueError, "Can't add element of degree %s and %s"\
                %(self.degree,y.degree)
        return FP_Element([self.coeffs[i]+y.coeffs[i] for i in range(len(self.coeffs))],self.module)

    def __neg__(self):
        """
        Returns the negative of the element.
        """
        return FP_Element([-self.coeffs[i] for i in range(len(self.coeffs))],self.module)

    def __sub__(self,y):
        """
        Returns the difference of the two elements.
        """
        return self.__add__(y.__neg__())
```

```
    def __cmp__(self,y):
        """
        Compares two FP_Elements for equality. Cannot compare elements in
        different degrees or different modules.
        """
        if self.module != y.module:
            raise TypeError, "Cannot compare elements in different modules."
        if self.degree != y.degree and self.degree != None and y.degree != None:
                                      ## Do we need to check degrees not None?
                                      ## ie self and y are not zero
            raise ValueError, \
            "Cannot compare elements of different degrees %s and %s"\
            %(self.degree, y.degree)
        if (self.__add__(y.__neg__())).__nonzero__():
            return 1
        else:
            return 0

    def __nonzero__(self):
  if self.degree == None:
            return False
        v,q,sec,bas = self.vec()
        return v != 0

    def _repr_(self):
        return '%s' % self.coeffs ## TO DO: Add parents when coeffs are sums:
                                  ## Sq(3)*M.0 + Sq(1)*M.2 is fine, but we'll
```

```
                                    ## need (Sq(3) + Sq(0,1))*M.0.
                                        Still a problem?

    def __mul__(self,x):
        """
        This is the action which is called when x*Sq(2) is
            evaluated. Really a left
        action but must be written on the right.
        """
        return FP_Element(\
          [x*self.coeffs[i] for i in range(len(self.coeffs))],self.
              module)

    def _l_action_(self,x):
        """
        ## FIX
        Multiplication of an FP_Element by a Steenrod Algebra
            element.
        This is written as a right multiplication, but its really a
             left
        multiplication.
        """
        return FP_Element(
               [x*self.coeffs[i] for i in range(len(self.coeffs))],
                   self.module)

    def free_vec(self,profile=None):            # prof FIX
        """
      Returns the vector in the free vector space corresponding to
          self.coeffs.
        If the coeffs are all 0, then we return the scalar 0, since
             it will be
        coerced up to the 0 vector in any vector space.

        INPUT:
```

```
-  ``profile`` - The profile function of a larger algebra than
   the one currently defined.

OUTPUT:  The vector in the vector space for self.parent
    corresponding
        to self.
      """
if profile == None:
   profile = self.profile
      n = self.degree
      if n == None:
          return 0
alg = SteenrodAlgebra(p=self.module.char, profile=profile)
      bas_gen = reduce(lambda x,y : x+y,\
        [[(i,bb) for bb in alg.basis(n-self.module.degs[i])] \
                 for i in range(len(self.module.degs))])
      bas_vec = VectorSpace(GF(self.module.char),len(bas_gen))
      bas_dict = dict(zip(bas_gen,bas_vec.basis()))
      r = zip(range(len(self.coeffs)),self.coeffs)  #[...(gen,op)
          ...]
      r = filter(lambda x: not x[1].is_zero(),r)   #remove
          trivial ops
      r = reduce(lambda x,y: x+y,\
       [map(lambda xx: (pr[0],\
             alg._milnor_on_basis(xx[0]), xx[1]),\
             [z for z in pr[1]]) for pr in r])
                # now, r = [....(gen,basis_op,coeff)...]
      return reduce(lambda x,y: x+y, map(lambda x : x[2]*bas_dict
          [(x[0],x[1])],r))
      # lll =  map(lambda x : x[2]*bas_dict[(x[0],x[1])],r)
# print "lll is ",lll,"======="
      # mmm =  reduce(lambda x,y: x+y, lll)
# print "mmm is ",mmm," ________"
      # return mmm
```

```
# this is the sum coeff*gen*basis_op = coeff*vector


    def vec(self,profile=None):    # prof FIX
        """
        Returns the vector form of self, as well as the linear
            transformation
        `q : F_n \rightarrow M_n` and `s:M_n \rightarrow F_n`,
            where `M_n`
        and `F_n` are the degree `n` parts of the module and free
            vector
        space, respectively.

        OUTPUT:

-     ``x``    - The unique vector form of self in `M_n`.

        -     ``q``    - The linear transformation from the free
            vector
                        space to the module.

        -     ``s``    - The linear transformation from the module
            to the
                        free vector space.

        -     ``bas``  - A list of pairs (gen_number,algebra element
            )
                        corresponding to self in the std basis of
                            the free module.

        """
    if profile == None:
        profile = self.profile
        n = self.degree
        if n == None:
            return 0,0,0,0
```

```
            quo, q, s, bas = self.module._pres_(n,profile=profile)
            return q(self.free_vec(profile=profile)),q,s,bas


        def nf(self,profile=None):          # prof FIX
            """
            Computes the normal form of self.
            """
        if profile == None:
            profile = self.profile
            if self.degree == None:
                return self
            v,q,sec,bas = self.vec(profile=profile)
            return self.module._lc_(sec(v),bas)

# The End
```

10 Appendix B: Resolution of $\mathcal{A}/\!\!/\mathcal{A}(2)$

In this section we show a resolution of $\mathcal{A}/\!\!/\mathcal{A}(2)$. Each stage of the resolution is determined by the kernel of the previous stage, and the i^{th} stage is denoted Ki.

```
sage: N = FP_Module([0],[[Sq(1)],[Sq(2)],[Sq(4)]])
sage: load computeresofA2.py
sage: K0.degs
[1, 2, 4]
sage: K0.rels
[
[Sq(1), 0, 0],
[Sq(0,1), Sq(2), 0],
[Sq(4), Sq(0,1), Sq(1)],
[Sq(0,0,1), Sq(0,2), Sq(4)]]

sage: K1.degs
[2, 4, 5, 8]
sage: K1.rels
[
[Sq(1), 0, 0, 0],
[Sq(4), Sq(2), Sq(1), 0],
[Sq(2,0,1) + Sq(6,1), Sq(0,0,1) + Sq(1,2) + Sq(4,1), Sq(6), Sq(0,1)
    + Sq(3)],
[Sq(6,0,1), 0, Sq(4,2), Sq(0,0,1) + Sq(1,2) + Sq(4,1)],
[Sq(0,3,1) + Sq(6,1,1), Sq(4,1,1), Sq(0,2,1) + Sq(6,0,1), Sq(0,1,1)
    + Sq(4,2)]]

sage: K2.degs
[3, 6, 11, 15, 18]
sage: K2.rels
[
[Sq(1), 0, 0, 0, 0],
[Sq(6,1), Sq(3,1), 0, 0, 0],
[Sq(0,1,1) + Sq(4,2), Sq(0,0,1) + Sq(1,2) + Sq(4,1), Sq(2), 0, 0],
```

```
[Sq(0,2,1), 0, 0, Sq(1), 0],
[Sq(2,2,1) + Sq(6,3), Sq(2,1,1) + Sq(5,0,1) + Sq(6,2), Sq(0,0,1) +
    Sq(1,2) + Sq(4,1), Sq(0,1), 0],
[0, Sq(0,2,1) + Sq(4,3) + Sq(6,0,1), Sq(1,0,1), Sq(4), Sq(1)],
[Sq(2,3,1), Sq(2,2,1) + Sq(6,3), Sq(4,2), Sq(0,2), Sq(0,1)],
[Sq(6,2,1), Sq(0,3,1) + Sq(3,2,1), 0, 0, Sq(4)],
[0, Sq(2,3,1), Sq(4,3), 0, Sq(0,2)]]

sage: K3.degs
[4, 12, 13, 16, 18, 19, 21, 22, 24]
sage: K3.rels
[
[Sq(1), 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(6,1), Sq(1), 0, 0, 0, 0, 0, 0, 0],
[0, Sq(2), 0, 0, 0, 0, 0, 0, 0],
[Sq(2,1,1) + Sq(6,2), Sq(4), Sq(3), 0, 0, 0, 0, 0, 0],
[Sq(0,2,1), 0, 0, Sq(1), 0, 0, 0, 0, 0],
[Sq(2,2,1) + Sq(6,3), Sq(0,0,1), Sq(0,2) + Sq(6), 0, Sq(1), 0, 0,
    0, 0],
[0, 0, Sq(0,0,1), Sq(4), Sq(2), Sq(1), 0, 0, 0],
[Sq(2,3,1), Sq(4,2), Sq(0,3) + Sq(2,0,1), 0, Sq(4), 0, Sq(1), 0,
    0],
[Sq(6,2,1), 0, Sq(0,1,1) + Sq(4,2), 0, 0, Sq(4), Sq(2), Sq(1), 0],
[0, Sq(0,2,1), Sq(5,0,1), Sq(0,3) + Sq(6,1), Sq(0,0,1), 0, Sq(4),
    0, Sq(1)],
[Sq(6,3,1), 0, Sq(0,2,1) + Sq(4,3), Sq(0,1,1), Sq(2,2), Sq(0,0,1),
    0, Sq(4), Sq(2)],
[0, 0, Sq(1,3,1) + Sq(4,2,1), 0, 0, Sq(4,0,1), Sq(2,0,1), Sq(2,2),
    Sq(0,2)]]

sage: K4.degs
[5, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 30]
sage: K4.rels
[
[Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
[Sq(6,1), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(0,1), Sq(2), 0, 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(2,1,1) + Sq(6,2), Sq(4), Sq(0,1), Sq(1), 0, 0, 0, 0, 0, 0, 0,
    0],
[Sq(0,2,1), 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0],
[Sq(6,3), Sq(0,0,1), Sq(0,2), Sq(4), Sq(0,1), Sq(1), 0, 0, 0, 0, 0,
     0],
[0, 0, 0, 0, Sq(0,2), Sq(4), Sq(0,1) + Sq(3), Sq(1), 0, 0, 0, 0],
[0, Sq(0,2,1) + Sq(6,0,1), 0, Sq(0,1,1) + Sq(4,2), 0, 0, Sq(6), Sq
    (4), Sq(0,1) + Sq(3), Sq(1),
    0, 0],
[0, 0, Sq(0,3,1), Sq(4,1,1), Sq(6,0,1), Sq(2,3) + Sq(4,0,1), Sq
    (3,0,1) + Sq(4,2),
    Sq(2,2), Sq(0,0,1) + Sq(1,2) + Sq(4,1), Sq(2,1), Sq(1,1), 0],
[0, 0, 0, 0, Sq(2,2,1) + Sq(6,3), Sq(0,2,1), Sq(2,1,1) + Sq(5,0,1)
    + Sq(6,2), 0,
    Sq(0,3) + Sq(3,2) + Sq(6,1), Sq(4,1), Sq(6), Sq(2)],
[0, 0, 0, 0, Sq(6,1,1), 0, Sq(4,3) + Sq(7,2), Sq(4,0,1), Sq(0,1,1)
    + Sq(4,2), Sq(2,2),
    Sq(0,0,1) + Sq(1,2), Sq(0,1)],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(0,2)]]

sage: K5.degs
[6, 14, 16, 17, 18, 20, 23, 26, 30, 32, 33, 36]
sage: K5.rels
[
[Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(6,1), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(2,1,1) + Sq(6,2), Sq(4), Sq(2), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(0,2,1), 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0],
[0, Sq(2,0,1), Sq(0,0,1) + Sq(1,2) + Sq(4,1), Sq(6), Sq(2,1), Sq
    (0,1) + Sq(3), 0, 0, 0, 0, 0, 0],
[0, Sq(4,3) + Sq(6,0,1), 0, Sq(0,1,1) + Sq(4,2), 0, Sq(0,0,1) + Sq
    (1,2) + Sq(4,1),
    Sq(1,1), 0, 0, 0, 0, 0],
```

```
[0, Sq(0,3,1), Sq(4,1,1), Sq(6,0,1), 0, Sq(4,2), Sq(0,0,1) + Sq
    (1,2) + Sq(4,1), Sq(1,1),
    0, 0, 0, 0],
[0, 0, 0, 0, Sq(0,2,1), 0, 0, 0, Sq(1), 0, 0, 0],
[0, 0, 0, Sq(6,1,1), Sq(2,2,1), Sq(0,2,1), Sq(3,0,1) + Sq(4,2), Sq
    (0,0,1) + Sq(1,2) + Sq(4,1),
    Sq(0,1), 0, 0, 0],
[0, Sq(4,3,1), 0, 0, Sq(0,3,1), Sq(1,2,1), Sq(4,0,1) + Sq(5,2), Sq
    (1,0,1) + Sq(2,2), Sq(4),
    Sq(2), Sq(1), 0],
[0, 0, Sq(4,3,1), 0, Sq(2,3,1), Sq(0,3,1) + Sq(6,1,1), Sq(7,2), Sq
    (4,2), Sq(0,2),
    0, Sq(0,1), 0],
[0, 0, 0, Sq(6,3,1), 0, 0, Sq(0,3,1) + Sq(6,1,1), Sq(6,0,1), 0, Sq
    (0,0,1) + Sq(4,1),
    Sq(0,2) + Sq(6), Sq(3)],
[0, 0, 0, 0, 0, 0, 0, Sq(0,2,1) + Sq(4,3), Sq(2,0,1), Sq(0,0,1), Sq
    (6), Sq(0,1) + Sq(3)],
[0, 0, 0, 0, 0, Sq(6,3,1), Sq(6,2,1), 0, Sq(6,2), Sq(4,2), 0, Sq
    (0,2)]]

sage: K6.degs
[7, 15, 18, 19, 23, 27, 30, 31, 33, 34, 36, 39, 39, 42]
sage: K6.rels
[
[Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(6,1), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(0,2,1), 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(6,1), Sq(3,1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(0,1,1) + Sq(4,2), Sq(0,0,1) + Sq(1,2) + Sq(4,1), 0, Sq(2),
    0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(0,2,1), 0, Sq(6,1), 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(2,2,1) + Sq(6,3), Sq(2,1,1) + Sq(5,0,1) + Sq(6,2), 0, Sq
    (0,0,1) + Sq(1,2) + Sq(4,1),
    Sq(0,1), 0, 0, 0, 0, 0, 0, 0, 0],
```

```
[0, 0, Sq(0,2,1) + Sq(4,3) + Sq(6,0,1), Sq(6,2), Sq(1,0,1), Sq(4),
    Sq(1), 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(0,2,1), 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0],
[0, Sq(2,3,1), Sq(2,2,1) + Sq(6,3), 0, Sq(4,2), Sq(0,2), Sq(0,1),
    0, 0, 0, 0, 0, 0, 0],
[0, Sq(6,2,1), Sq(0,3,1) + Sq(3,2,1), 0, 0, 0, Sq(4), 0, Sq(1), 0,
    0, 0, 0, 0],
[0, 0, Sq(2,3,1), 0, Sq(4,3), 0, Sq(0,2), Sq(2,1), Sq(0,1), 0, 0,
    0, 0, 0],
[0, Sq(6,3,1), 0, 0, Sq(1,2,1) + Sq(4,1,1), Sq(4,2), Sq(0,0,1), 0,
    Sq(4), Sq(3), Sq(1), 0, 0, 0],
[0, 0, 0, 0, Sq(0,3,1), 0, 0, Sq(2,2), Sq(0,2), 0, Sq(0,1), 0, 0,
    0],
[0, 0, Sq(6,3,1), 0, 0, Sq(0,2,1), 0, Sq(0,3) + Sq(6,1), Sq(0,0,1),
     Sq(0,2) + Sq(6),
    Sq(4), Sq(1), 0, 0],
[0, 0, 0, Sq(6,3,1), Sq(5,2,1), 0, Sq(4,0,1), Sq(4,2), Sq(2,2), Sq
    (0,0,1) + Sq(1,2) + Sq(7),
    0, 0, Sq(2), 0],
[0, 0, 0, 0, 0, Sq(2,2,1), Sq(6,2), 0, Sq(2,0,1), Sq(5,1), Sq(0,2),
     Sq(0,1), Sq(0,1), 0],
[0, 0, 0, 0, 0, 0, Sq(2,2,1), Sq(4,1,1), 0, Sq(1,1,1) + Sq(4,0,1) +
     Sq(5,2), Sq(2,0,1), Sq(0,2),
    Sq(0,2), Sq(0,1)],
[0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(7,0,1), Sq(6,2), Sq(0,3), 0, Sq(0,2)
    ],
[0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(6,3,1), 0, Sq(4,2,1), 0, Sq(7,0,1)]]

sage: K7.degs
[8, 16, 20, 24, 25, 28, 30, 31, 32, 33, 34, 36, 37, 39, 40, 41, 42,
     45, 48, 56]
sage: K7.rels
[
[Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
[Sq(6,1), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0],
[Sq(0,2,1), 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0],
[0, Sq(6,1), 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0],
[0, 0, 0, Sq(2), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(4,3,1), Sq(2,1,1) + Sq(6,2), 0, Sq(4), Sq(3), 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(0,2,1), Sq(6,1), 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0],
[0, Sq(2,2,1) + Sq(6,3), 0, Sq(0,0,1), Sq(0,2) + Sq(6), 0, Sq(1),
   0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, Sq(6,2), 0, Sq(0,0,1), Sq(4), Sq(2), Sq(1), 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0],
[0, 0, Sq(0,2,1), 0, 0, 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0],
[0, 0, Sq(4,1,1), 0, Sq(0,3) + Sq(2,0,1), Sq(6), Sq(4), Sq(0,1), 0,
    Sq(1), 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
[0, Sq(6,2,1), Sq(6,3), 0, Sq(4,2), Sq(0,0,1) + Sq(4,1), 0, Sq(4),
   0, Sq(2), Sq(1), 0, 0,
   0, 0, 0, 0, 0, 0, 0],
[0, 0, Sq(4,2,1), Sq(0,2,1), 0, Sq(2,0,1) + Sq(6,1), Sq(0,0,1), Sq
   (0,2) + Sq(6), Sq(2,1),
   Sq(4), 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, Sq(0,2,1) + Sq(4,3) + Sq(6,0,1), Sq(0,1,1), 0, 0, 0,
   0, Sq(4), Sq(2), Sq(1),
   0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(4,2,1), Sq(0,3,1), Sq(0,2,1) + Sq(6,0,1), 0, Sq(4,2),
   Sq(0,3) + Sq(2,0,1), 0, 0, 0,
   Sq(4), Sq(2), Sq(1), 0, 0, 0, 0, 0],
[0, 0, 0, 0, Sq(4,2,1), Sq(4,1,1), 0, Sq(2,3) + Sq(4,0,1), Sq
   (0,1,1), Sq(2,0,1),
```

```
    Sq(2,2), Sq(0,2), Sq(2,1), Sq(0,1), 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, Sq(5,2,1), 0, 0, Sq(2,1,1) + Sq(6,2), Sq(2,3), Sq(4,2)
    ,
    Sq(0,3) + Sq(2,0,1) + Sq(6,1), Sq(0,0,1), Sq(6), Sq(4), 0, Sq
        (2), Sq(1), 0, 0, 0],
[0, 0, 0, 0, 0, Sq(4,2,1), Sq(2,2,1), 0, Sq(0,2,1), 0, Sq(4,0,1),
    Sq(2,0,1), Sq(2,2),
    Sq(0,2), Sq(2,1), 0, Sq(0,1), 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, Sq(4,2,1), Sq(6,1,1), Sq(2,2,1), 0, 0, Sq
    (4,0,1), Sq(2,0,1), Sq(2,2),
    Sq(4,1), Sq(0,2), Sq(0,1), 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, Sq(6,2,1), 0, 0, Sq(2,2,1), Sq(4,1,1), Sq
    (6,2), 0, Sq(4,2),
    Sq(2,0,1), Sq(0,2), Sq(0,1), 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(4,3,1), 0, 0, Sq(2,2,1), 0, Sq
    (0,2,1) + Sq(4,3), 0,
    Sq(2,0,1) + Sq(3,2), Sq(0,2), 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(5,2,1), Sq(4,2,1), Sq
    (0,3,1), Sq(2,2,1), 0,
    0, Sq(1)],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(6,2,1), 0, Sq(1,3,1) +
    Sq(4,2,1), 0,
    Sq(6,0,1) + Sq(7,2), Sq(3,0,1), Sq(2)],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(4)]]

sage: K8.degs
[9, 17, 21, 25, 26, 28, 29, 31, 32, 33, 34, 35, 37, 38, 41, 42, 43,
     45, 48, 51, 54, 57, 58, 60]
sage: K8.rels
[
[Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0],
[Sq(6,1), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0],
```

```
[Sq(0,2,1), 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0],
[0, Sq(6,1), 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0],
[0, 0, 0, Sq(0,1), Sq(2), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0],
[Sq(4,3,1), Sq(2,1,1) + Sq(6,2), 0, Sq(4), Sq(0,1), Sq(1),
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(0,2,1), Sq(6,1), 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(6,3), Sq(2,3) + Sq(4,0,1), Sq(0,0,1), Sq(0,2), Sq(4), Sq
    (0,1), Sq(1),
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, Sq(0,2,1), 0, 0, 0, 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0],
[0, 0, Sq(4,1,1), 0, 0, 0, Sq(0,2), Sq(4), Sq(0,1) + Sq(3), 0, Sq
    (1),
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(0,2,1) + Sq(6,0,1), Sq(5,0,1), Sq(0,1,1), Sq(6,1), Sq
    (0,0,1), Sq(6), 0,
    Sq(4), Sq(0,1), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, Sq(0,3,1), Sq(4,1,1), Sq(0,2,1) + Sq(4,3), Sq(2,3) +
    Sq(4,0,1),
    Sq(0,1,1) + Sq(4,2), Sq(6,1), Sq(2,2), Sq(0,0,1) + Sq(1,2) + Sq
        (4,1), Sq(2,1), Sq(1,1),
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(6,2,1), Sq(5,2,1), Sq(0,3,1), 0, Sq(0,2,1) + Sq(4,3),
    Sq(5,0,1) + Sq(6,2),
    Sq(2,3), 0, Sq(2,0,1) + Sq(3,2), Sq(4,1), Sq(6), 0, Sq(2), 0,
        0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, Sq(0,3,1) + Sq(6,1,1), 0, Sq(6,0,1), Sq(2,1,1) +
     Sq(6,2), Sq(4,0,1),
    Sq(0,1,1) + Sq(3,0,1) + Sq(4,2), Sq(2,2), Sq(0,0,1) + Sq(1,2),
        Sq(1,1), Sq(0,1),
    0, 0, 0, 0, 0, 0, 0, 0],
```

```
[0, 0, 0, 0, 0, 0, 0, 0, Sq(1,2,1), Sq(0,2,1) + Sq(4,3) + Sq(6,0,1)
    , 0, 0, 0, Sq(1,0,1),
    Sq(5), 0, Sq(3), Sq(1), 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, Sq(4,3,1), Sq(6,2,1), 0, Sq(0,3,1) + Sq(3,2,1), 0,
    0, 0, Sq(2,3),
    0, Sq(0,0,1) + Sq(1,2), Sq(0,2), 0, Sq(0,1), 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, Sq(6,3,1), 0, Sq(6,2,1), 0, Sq(4,2,1), Sq(0,3,1)
    , 0, Sq(4,3) + Sq(7,2),
    Sq(3,0,1), 0, Sq(5,1), Sq(0,2), Sq(0,1), 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(4,3,1), 0, 0, Sq(0,3,1) + Sq
    (6,1,1),
    Sq(0,2,1) + Sq(6,0,1) + Sq(7,2), Sq(5,0,1), Sq(1,1,1) + Sq
        (4,0,1) + Sq(5,2), 0,
    Sq(0,2), Sq(0,1), 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(0,3,1) + Sq(3,2,1) +
    Sq(6,1,1), 0, Sq(7,0,1),
    0, 0, Sq(0,2), Sq(0,1), 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(4,2,1), 0, Sq(2,2,1),
     Sq(0,2,1), Sq(3,0,1),
    0, 0, Sq(1), 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(6,3,1), 0, 0, Sq(1,3,1)
    + Sq(4,2,1), Sq(2,2,1),
    Sq(5,0,1), Sq(2,0,1) + Sq(3,2), Sq(6), Sq(0,1), Sq(2), 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(5,2,1), Sq(1,3,1)
    + Sq(4,2,1), 0, Sq(6,2),
    0, Sq(0,2) + Sq(6), Sq(0,1), Sq(2), 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(7,2,1), 0, 0, 0, Sq
    (0,2,1), Sq(4,2), Sq(0,0,1),
    Sq(4), Sq(0,1), Sq(1)],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    Sq(6,2,1), 0, Sq(0,2,1), 0, Sq(0,0,1), Sq(0,2), Sq(4)]]

sage: K9.degs
[10, 18, 22, 26, 28, 29, 30, 32, 34, 35, 38, 42, 44, 45, 46, 48,
    51, 54, 57, 58, 60, 60, 61, 64]
```

```
sage: K9.rels
[
[Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0],
[Sq(6,1), Sq(1), 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(0,2,1), 0, Sq(1), 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(6,1), 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[Sq(4,3,1), Sq(2,1,1) + Sq(6,2), 0, Sq(4), Sq(2), Sq(1), 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, Sq(0,2,1), Sq(6,1), 0, 0, 0, Sq(1), 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, Sq(0,2,1), 0, 0, 0, 0, 0, Sq(1), 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(0,3) + Sq(6,1), Sq(0,0,1), Sq(6), Sq(2,1), Sq(0,1) +
Sq(3), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(4,3) + Sq(6,0,1), Sq(4,0,1) +
Sq(5,2), Sq(0,1,1) + Sq(4,2), 0, Sq(0,0,1) + Sq(1,2) + Sq(4,1), 0,
   Sq(1,1), 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(6,1,1), Sq(1,2,1) + Sq(5,3), Sq(6,0,1), 0, Sq(4,2), 0,
    Sq(0,0,1) +
Sq(1,2) + Sq(4,1), Sq(1,1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, Sq(0,2,1),
0, Sq(6,1), 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, Sq(1,3,1) + Sq(4,2,1),
Sq(6,1,1), Sq(2,2,1), Sq(0,2,1), 0, Sq(3,0,1) + Sq(4,2), Sq(0,0,1)
   + Sq(1,2) + Sq(4,1), Sq(0,1), 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, Sq(2,1,1) + Sq(6,2), Sq(2,3) + Sq(4,0,1)
+ Sq(5,2), Sq(2,2), Sq(4), Sq(2), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0],
[0, 0, 0, 0, Sq(6,2,1), 0,
```

```
Sq(4,2,1), Sq(2,2,1) + Sq(6,3), Sq(0,2,1), Sq(6,2), Sq(0,3) + Sq
    (2,0,1) + Sq(3,2) + Sq(6,1), 0, 0,
0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(6,3,1), 0, Sq(6,2,1), 0, Sq(6,1,1), 0, Sq(4,3),
Sq(3,0,1) + Sq(4,2), Sq(0,2), Sq(1,1), Sq(0,1), 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0],
[0, 0, 0, 0, 0,
Sq(6,3,1), 0, 0, 0, Sq(3,2,1), Sq(0,2,1) + Sq(4,3) + Sq(6,0,1), 0,
    Sq(0,0,1) + Sq(1,2) + Sq(4,1),
Sq(6), 0, Sq(0,1) + Sq(3), 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, Sq(6,2,1), 0, Sq(0,3,1) +
Sq(6,1,1), Sq(0,2,1) + Sq(6,0,1) + Sq(7,2), Sq(2,0,1), 0, Sq(0,2) +
     Sq(6), Sq(2,1), Sq(3), 0, 0, 0,
0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, Sq(6,3,1), 0, 0, Sq(0,3,1) + Sq(3,2,1), Sq
    (6,2), Sq(1,3) +
Sq(4,2), 0, Sq(2,2), Sq(0,2), Sq(0,1), 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(6,3,1),
Sq(6,2,1), 0, Sq(4,3) + Sq(6,0,1), Sq(6,2), 0, Sq(2,0,1), Sq(0,2),
    Sq(0,1), 0, 0, 0, 0, 0, 0],
[0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(2,2,1) + Sq(5,1,1), 0, Sq
    (0,2,1) + Sq(4,3) + Sq(6,0,1), 0,
Sq(1,0,1), Sq(5), 0, Sq(1), 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(0,3,1), 0, 0,
Sq(5,0,1), Sq(2,0,1), Sq(0,2), Sq(0,1), 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Sq(4,2,1), Sq(0,3,1) + Sq(6,1,1), 0, Sq(5,2), Sq(1,0,1) + Sq(2,2),
    0, Sq(4), Sq(2), Sq(2), Sq(1),
0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(4,2,1), 0, Sq(5,0,1),
     Sq(2,0,1), Sq(0,2), 0, 0,
Sq(0,1), 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(5,2,1), Sq(2,2,1),
     Sq(5,0,1) +
```

```
Sq(6,2), 0, 0, 0, Sq(0,2), 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(3,2,1),
Sq(7,2), Sq(4,2), Sq(2,0,1), Sq(0,0,1) + Sq(1,2) + Sq(7), 0, Sq(6),
     Sq(0,1) + Sq(3)],
[0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(1,2,1) + Sq(7,0,1), Sq
    (4,3) + Sq(6,0,1), Sq(4,0,1) +
Sq(5,2), 0, Sq(0,1,1) + Sq(4,2), Sq(0,0,1) + Sq(1,2) + Sq(4,1)],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, Sq(7,2,1), 0, 0, Sq(1,2,1) + Sq(4,1,1) + Sq(5,3),
     0, Sq(4,3) + Sq(6,0,1),
Sq(0,1,1) + Sq(4,2)]]

sage: K10.degs
[11, 19, 23, 27, 30, 31, 35, 35, 39, 42, 43, 45, 46, 47, 48, 51,
    51, 54, 57, 59, 60, 62, 63, 66,
67, 71, 74]
sage: K10.rels
[
[Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0],
[Sq(6,1),
Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0],
[Sq(0,2,1), 0,
Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0],
[0, Sq(6,1), 0,
Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0],
[0, Sq(0,2,1),
Sq(6,1), 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0],
[0, 0,
```

```
Sq(0,2,1), 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0],
[0, 0, 0,
Sq(6,1), Sq(3,1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0],
[0, 0,
Sq(4,1,1), Sq(0,1,1) + Sq(4,2), Sq(0,0,1) + Sq(1,2) + Sq(4,1), 0,
    0, Sq(2), 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(0,2,1), 0, Sq(6,1), 0, 0, Sq(1), 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, Sq(6,2,1), Sq(2,2,1) + Sq(6,3), Sq(2,1,1) + Sq(5,0,1) +
Sq(6,2), 0, 0, Sq(0,0,1) + Sq(1,2) + Sq(4,1), Sq(0,1), 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0],
[0, 0, 0, Sq(6,1,1), Sq(0,2,1) + Sq(4,3) + Sq(6,0,1) + Sq(7,2), Sq
    (2,1,1) + Sq(6,2), 0,
0, Sq(4), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0],
[0, 0, 0, 0, 0, Sq(0,2,1),
Sq(6,1), 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0],
[0, 0, 0, Sq(2,3,1),
Sq(2,2,1) + Sq(6,3), 0, 0, Sq(4,2), Sq(0,2), Sq(0,1), 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0],
[0, 0, 0, Sq(6,2,1), Sq(3,2,1) + Sq(6,1,1), 0, 0, Sq(4,0,1) + Sq
    (5,2), Sq(0,0,1), Sq(4),
Sq(0,1), Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, Sq(2,3,1), 0, 0, 0, 0,
Sq(0,2), 0, Sq(0,1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, Sq(0,2,1), 0,
0, 0, 0, 0, 0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, Sq(6,3,1), Sq(6,2,1), 0, 0,
```

```
Sq(1,2,1) + Sq(4,1,1), Sq(4,2), 0, Sq(0,2) + Sq(6), Sq(4), Sq(3),
     0, Sq(1), 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0],
[0, 0, 0, 0, 0, Sq(4,3,1), 0, 0, Sq(6,2), Sq(2,0,1), 0, Sq(0,2), 0,
      0, Sq(0,1), 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, Sq(4,2,1), Sq(4,2,1), Sq(0,2,1) + Sq(6,0,1), 0,
Sq(2,0,1), 0, Sq(0,2) + Sq(6), Sq(2,1), Sq(4), Sq(1), 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0],
[0, 0, 0,
0, 0, 0, Sq(2,3,1), Sq(5,2,1), 0, Sq(4,0,1), Sq(4,2), Sq(2,2), Sq
     (0,0,1) + Sq(1,2) + Sq(7), 0, 0,
Sq(2), Sq(2), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(6,2), 0, 0, 0, 0,
Sq(0,2), 0, Sq(0,1), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, Sq(6,3,1), 0, 0, 0, 0,
Sq(6,2), Sq(2,3), Sq(0,1,1), Sq(2,0,1), 0, Sq(0,2), Sq(0,1), 0, 0,
     0, 0, 0, 0, 0, 0, 0],
[0, 0, 0,
0, 0, 0, 0, 0, 0, 0, Sq(4,2,1), Sq(2,2,1), Sq(1,2,1) + Sq(5,3), 0,
     0, Sq(2,0,1), 0, Sq(0,2),
Sq(0,1) + Sq(3), Sq(1), 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(0,2,1), 0,
0, 0, 0, Sq(3), Sq(1), 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(4,2,1),
Sq(0,3,1) + Sq(6,1,1), Sq(2,2,1), 0, Sq(5,0,1), Sq(2,0,1) + Sq(3,2)
     , Sq(0,2), 0, Sq(0,1), 0, 0, 0,
0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(4,3,1) + Sq(7,2,1), Sq
     (6,2,1), 0, 0, 0,
Sq(5,0,1), Sq(3,2), Sq(4,1), Sq(0,2), 0, Sq(0,1), 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, Sq(4,2,1), 0, Sq(7,0,1), 0, Sq(6,1), 0, Sq(3,1), 0, 0, 0,
     0, 0],
[0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, Sq(6,3,1), 0, 0, Sq(5,2,1), 0, Sq(5,0,1) + Sq(6,2),
    Sq(0,1,1), Sq(3,2), Sq(0,0,1) +
Sq(1,2) + Sq(4,1) + Sq(7), Sq(0,2), Sq(0,1), Sq(2), 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, Sq(2,2,1), Sq(5,0,1), Sq(4,2), Sq(2,0,1), Sq(4,1), Sq
    (0,2), Sq(0,1), 0, 0, 0],
[0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(5,2,1), 0, Sq
    (0,2,1) + Sq(4,3), Sq(5,0,1) +
Sq(6,2), Sq(3,0,1) + Sq(4,2), 0, Sq(0,2), 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, Sq(2,2,1), Sq(0,2,1) + Sq(4,3) + Sq(6,0,1), 0, Sq(3,0,1),
    0, 0, Sq(5), Sq(1), 0],
[0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(2,2,1), 0, Sq
    (6,2), Sq(5,2), Sq(1,0,1),
Sq(0,0,1) + Sq(1,2), Sq(0,1), 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Sq(0,3,1) + Sq(6,1,1), 0, Sq(4,3) + Sq(7,2), Sq(5,0,1) + Sq(6,2),
    Sq(2,0,1), Sq(1,0,1), Sq(4),
Sq(1)],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(7,2,1),
    0, 0, Sq(2,2,1),
Sq(1,2,1) + Sq(7,0,1), Sq(4,0,1), 0, Sq(0,2), Sq(0,1)],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, Sq(5,2,1), Sq(3,2,1), Sq(2,2,1), 0, Sq(4,0,1), 0,
     Sq(4)],
[0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Sq(7,0,1), 0, 0, Sq
    (0,2)]]
```

References

[1] J. F. Adams and H. R. Margolis, *Modules over the Steenrod algebra*, Topology **10** (1971), 271-282.

[2] J. Adem, *The iteration of the Steenrod squares in algebraic topology*, Proc. Nat. Acad. U.S.A. **38** (1952), 720-726.

[3] D.W. Anderson and D.M. Davis, *A vanishing theorem in homological algebra*, Comment. Math. Helv. **48** (1973), 318 - 327.

[4] H. R. Margolis, *Spectra and the Steenrod algebra*, North-Holland, 1983.

[5] J. F. Adams and H. R. Margolis, *Sub-hopf algebras of the Steenrod algebra*, Math. Proc. Cambridge Philos. Soc. **76** (1974), 45-52.

[6] J. W. Milnor, *The Steenrod algebra and its Dual*, Ann. of Math. (2) **67** (1958), 150-171.

[7] R. E. Mosher and M. C. Tangora, *Cohomology Operations and Applications in Homotopy Theory*, Harper and Row Publ., 1968.

[8] N. E. Steenrod, *Cohomology Operations*, Princeton University Press, 1962.