Calculation of Large Ext Modules

Robert R. Bruner

1 Introduction

Let A be a graded algebra of finite type over a finite field k, and let M be an A-module of finite type which is bounded below. Our goal is to compute

$$H^{st}(A) = Ext_A^{st}(k,k) \quad \text{and} \quad H^{st}(M) = Ext_A^{st}(M,k)$$

for small s and t by directly constructing free resolutions

$$0 \leftarrow M \leftarrow D_0 \leftarrow D_1 \leftarrow D_2 \leftarrow \cdots$$

and

$$0 \leftarrow k \leftarrow C_0 \leftarrow C_1 \leftarrow C_2 \leftarrow \cdots$$

We shall let k, A, M, C, and D have these meanings throughout the paper.

From the resolutions, we immediately obtain k bases for Ext. In addition, Lemma 3.2 shows that some product information is already visible in the resolution. In §3 we show how to obtain complete product information by computing chain maps $D \to C$ and $C \to C$, and in §4 we show how to determine Massey products by computing chain null homotopies.

In the programming, our main goal has been economy of space, since memory has been the limiting factor in our experience. A second goal has been to make it as easy as possible to use different algebras and modules. The latter goal has been achieved by using a very small number of routines to define the algebra and the module, so that a new algebra or module can be used by simply redefining these few routines.

We will first describe the algorithms for doing the calculations outlined above, then describe our implementations.

2 Cohomology

2.1 The Connected case

Assume first that A is connected ($A_0 = k$). Then modules have minimal free resolutions, and an A basis for the minimal resolution of M is dual to a k basis for $H(M)$. To find the minimal resolution, apply the following algorithm, stopping when you want to or have to.

$Ext := 0$;
For $t = \mathrm{conn}(M)$ **to** ∞ **begin**
  $Oldker :=$ Basis for M_t;
  For $s = 0$ **to** Maxfilt(t) **begin**
    $Image := \emptyset$;
    $Newker := \emptyset$;
    For $g \in Ext^s$
      For $op \in$ Basis of $A_{t-deg(g)}$ **begin**
        $x := op * g$;
        $dx :=$ Act$(op, \mathrm{diff}(g))$;
        Reduce (x, dx) against $Image$;
        If $dx = 0$ **then** append x to $Newker$
          else insert (x, dx) into $Image$;
      end {op and g};
    For $cyc \in Oldker$ **begin**
      Reduce cyc against $Image$;
      If $cyc \neq 0$ **then begin**
        add a generator g to Ext^s;
        $\mathrm{diff}(g) := cyc$;
        $\deg(g) := t$;
        insert (g, cyc) into $Image$;
    end {cyc};
    $Oldker := Newker$;
  end{s};
end{t}.

Several parts of this require some elaboration or definition.

1. Ext^s is a list of generators in filtration s, for each $s \geq 0$. Tor would probably be more appropriate, but the perfect duality between Tor and Ext here allow us to name it after the actual object of interest to us.

2. $\text{conn}(M) = \min\{t | M_t \neq 0\}$, the connectivity of M.

3. $\text{diff}(g)$ and $\deg(g)$ are the differential and degree of the generator g.

4. *Oldker* and *Newker* are lists of elements which span the kernels.

5. *Image* is an ordered list of pairs (x, dx) such that the dx's form a basis for the image of D^{st} in $D^{s-1,t}$.

6. Maxfilt(t) must be a nondecreasing function of t for the results of the algorithm to have any significance. Within this requirement, Maxfilt may be adapted to the needs or knowledge of the user. Occasionally, we will want only a presentation

$$D_1 \to D_0 \to M \to 0$$

in which case Maxfilt$(t) = 1$ is appropriate. If a complete resolution is required and nothing is known about M or $H(M)$, Maxfilt$(t) = t - \text{conn}(M)$ is appropriate. If we know that $H^s(M) = 0$ for $s > U(t)$ then Maxfilt$(t) = U(t)$ is appropriate. Letting s go any higher than this will merely verify the "vanishing line" $U(t)$. In our original application, $A =$ the mod 2 Steenrod algebra and $M = Z_2$, an interesting variant of this occurs. We have $H^{st}(A) = 0$ for $s > t/3$, approximately, except for $H^{tt}(A)$ which has 1 generator whose differential is Sq^1 of the generator in $H^{t-1,t-1}(A)$. Rather than waste 2/3 of our effort as we would using Maxfilt$(t) = t$, we "prime" the algorithm with the generators in H^{tt} and let Maxfilt$(t) = t/3$ (approximately). A similar procedure can be applied whenever D^{st} can be determined from prior values of D. Note that it is not sufficient to know H^{st} in terms of prior H; we also need to know the differential $d : D^{st} \to D^{s-1,t}$ since these differentials may well play a role in higher internal degrees.

7. Act(op,diff(g)) applies op to diff(g). If $s = 0$, diff(g) $\in M$, so this depends on M. If $s > 0$ then diff(g) $\in D$, so this depends only on the algebra A (since D is free).

8. "Reduce (x, dx) against *Image*" runs through *Image* in order, replacing (x, dx) by $(x - a, dx - da)$ for each pair $(a, da) \in$ *Image* such that the highest term of da occurs in dx, until the highest term of dx is distinct from the highest terms of all the entries in *Image*, or until dx becomes 0.

   "Reduce *cyc* against *Image*" runs through *Image* in order, replacing *cyc* by $cyc - da$ for each pair $(a, da) \in$ *Image* such that the highest term of da occurs in *cyc*, until the highest term of *cyc* is distinct from the highest terms of all the entries in *Image*, or until *cyc* becomes 0.

9. Append simply puts the new element at the end of *Newker*, because we have no need to order the bases for the kernels.

10. Insert inserts (x, dx) into *Image* so that the dx's are in order, where we order elements by their highest terms. Conceptually, this amounts to keeping the matrix representation of the differential in upper triangular (or row echelon) form with respect to the appropriate bases.

11. During the second phase ("**For** $cyc \in Oldker \ldots$") we compute Ext^{st} by adding generators as necessary to make the cokernel

$$Image \to Oldker \to Coker \to 0$$

    equal 0. We do this by using Reduce to make each element of *Oldker* independent of *Image* (or 0).

12. The parts of the algorithm which do not specify an order in which to process elements do not depend on the order for their correctness. However, careful attention to the order can have significant effects on the speed with which Reduce operates. Basically, we want to produce (op, gen) pairs in an order that will cause the highest terms of $op * \text{diff}(gen)$ to appear in reverse order. This means that new terms will tend to go at the beginning of *Image*, so that Reduce will have to do as few comparisons and subtractions as possible.

2.2 The nonconnected case

Here we do not have free resolutions that are minimal in the sense that the differentials become 0 when $- \otimes_A k$ or $Hom_A(-, k)$ are applied. A simple example is provided by $A = Z_2[Z_3]$ concentrated in degree 0. We have $H^s(A) = 0$ for $s > 0$, but there are no free resolutions of Z_2 over $Z_2[Z_3]$ of finite length. This is obvious if one observes that

$$\dim_{Z_2} \mathrm{Ker}(d_s) \equiv (-1)^s \pmod 3$$

(since $\dim_{Z_2} A = 3$) so that $\mathrm{Ker}(d_s)$ can never be 0.

However, with slight modifications, the algorithm above will produce free resolutions even when A is not connected. The modifications are in the second phase, which should be replaced by

```
For cyc ∈ Oldker begin
    Reduce cyc against Image;
    If cyc ≠ 0 then begin
        add a generator g to Ext^s;
        diff(g) = cyc;
        deg(g) = t;
        insert (g, cyc) into Image;
        For op ∈ Basis of A_0 begin
            x := op * g;
            dx := Act(op, cyc);
            Reduce (x, dx) against Image;
            If dx = 0 then append x to Newker
                      else insert (x, dx) into Image;
        end {op };
    end {if};
end {cyc};
```

This algorithm is correct, in the sense that it terminates. However, the size of the resolution it produces is sensitive to the order in which the basis for *Oldker* is processed. Any efficient algorithm for producing a minimal (or at least "small") generating set over A for an A-submodule of a free A-module, given a k-basis for it, would be very useful here. I would be

happy to know one even in the case $A = Z_2[\Sigma_n]$ for n=5 or 6. One obvious first step in this direction would be to order the k generators of $Oldker$ by the dimension of the A_0-submodule of the quotient $Oldker/Image$ that they generate and hit one of maximal dimension first. This will alter the dimensions for the remaining elements however, so could be a very slow process.

In the nonconnected case, the free resolution requires further processing in order to produce Ext. Of course, we will actually compute Tor and use duality to compute Ext. To compute Tor, we replace each $\mathrm{diff}(g) = \sum op_i * g_i$ by $\sum \epsilon(op_i) * g_i$, where $\epsilon : A \to k$ is the augmentation. We then recompute $Image$ and $Newker$ and the cokernel

$$Image \to Oldker \to Tor \to 0$$

just as before. This second homology computation is very fast compared to the first, since $\dim_{Z_2}(D \otimes_A Z_2) = \dim_A(D)$ is generally much smaller than $\dim_{Z_2}(D)$.

We will not discuss the nonconnected case any further in this paper.

3 Products

There are two sorts of products in Ext which we could use to make $H(A)$ an algebra and $H(M)$ an $H(A)$-module. One is the external product

$$Ext_A(M,k) \otimes Ext_A(k,k) \to Ext_{A\otimes A}(M,k)$$

induced by the tensor product of modules, and its internalization by pullback along the coproduct $A \to A \otimes A$, when A is a Hopf algebra. However, this is of little help when A is not a Hopf algebra. Also, it presents computational problems because of the size of the tensor product complex: if C and D strain memory capacity separately, then $C \otimes D$ is completely out of reach. The other possibility, Yoneda's composition product

$$Ext_A(M,k) \otimes Ext_A(k,k) \to Ext_A(M,k)$$

suffers neither of these restrictions. In fact, it is vastly more efficient in memory use, since for long stretches of the computation, only a single

filtration of each of C and D is needed. Thus, it is efficient to keep in memory only those differentials for the filtrations currently of interest. Another advantage is that we may easily restrict attention to the action on indecomposables using the Yoneda definition, a task which is less naturally accomplished using the tensor product.

In fact, there are two versions of Yoneda's composition product. The one we will *not* use views Ext as equivalence classes of extensions. The one we want uses the isomorphism between $Ext^s(M,k)$ and the chain homotopy classes of degree s chain maps from D to C, where D and C are free resolutions of M and k respectively.

In these terms, the product is just composition of chain maps. In terms of representative cocycles, $[y][x] = [yx_{s+s'}]$, where $[x]$ denotes the cohomology class of the cocycle x.

$$\begin{array}{ccccccccccc}
0 & \leftarrow & M & \leftarrow & D_0 & \leftarrow & \cdots & \leftarrow & D_s & \leftarrow & \cdots & \leftarrow & D_{s+s'} \\
 & & & & & & & & \overset{x}{\swarrow} & & & & \downarrow \tilde{x}_{s+s'} \\
 & & & & & & & & \downarrow \tilde{x}_s & & & & \\
 & & & & & & k & \leftarrow & C_0 & \leftarrow & \cdots & \leftarrow & C_{s'} \\
 & & & & & & & & & & & & \downarrow y \\
 & & & & & & & & & & & & k
\end{array}$$

In the following lemma we translate this into terms suitable for mechanical calculation. If $g \in D_s$, let $g^* : D_s \to k$ be the cochain dual to g with respect to a fixed k-basis $\{a_i g_j\}$ of D, where $\{g_j\}$ is an A-basis of D and $\{a_i\}$ is a k-basis of A. If $y : D_s \to k$ is a cocycle (e.g., $y = g_i^*$ for some i), then it can be lifted to a chain map $\tilde{y} : D \to C$.

Lemma 3.1 *If $\{h_i\}$ and $\{g_i\}$ are A-bases of the minimal free resolutions C and D, then*

$$h_i^* g_j^* = \sum_g h_i^*(\widetilde{g_j^*}(g))g^*$$

summing over all g of the correct homological degree. That is, the coefficient of g^* in $h_i^* g_j^*$ is the coefficient of h_i in $\widetilde{g_j^*}(g)$.

Thus, the entire $H(A)$ action on the element g_j^* can be seen by inspecting the chain map $\widetilde{g_j^*}$.

In fact, with a bit of care in setting up $d : C_1 \longrightarrow C_0$, the action of $H^1(A)$ can already be seen in the differential of D. Let I be the augmentation ideal

of A, and choose $\{a_i\} \subset I$ so that $\{a_i + I^2\}$ is a k-basis for I/I^2. Let $\{a_i'\}$ be a k-basis for I^2. Then let us take $\{1\} \cup \{a_i\} \cup \{a_i'\}$ as our k-basis of A. Since C is minimal, $C_0 = A$ generated by 1, and C_1 is free over A on a set $\{h_i\} \cong \{a_i\}$. We may assume that $d(h_i) \equiv a_i \ (mod\ I^2)$.

Lemma 3.2 *$h_i^* x^* = (a_i x)^* d$. That is, the coefficient of g^* in $h_i^* x^*$ is the coefficient of $a_i x$ in $d(g)$.*

Proof: The k-linear (but not A-linear) maps a_i^* and $(a_i x)^*$ satisfy $h_i^* = a_i^* d$ and $a_i^*(\widetilde{x^*})_s = (a_i x)^*$. Therefore,

$$\begin{aligned} h_i^* x^* &= h_i^*(\widetilde{x^*})_{s+1} \\ &= a_i^* d(\widetilde{x^*})_{s+1} \\ &= a_i^*(\widetilde{x^*})_s d \\ &= (a_i x)^* d. \end{aligned}$$

We compute the chain map $\widetilde{g^*}$ induced by a cocycle g^* in exactly the same way we would prove such a lift exists. The algorithm follows.

(Let $g \in Ext^{s't'}$.)

For $g_1 \in Ext^{s'}$
  if $g_1 = g$ **then** $\widetilde{g^*}(g_1) := 1$
    else $\widetilde{g^*}(g_1) := 0$;
For $s = 1$ **to** ∞
  For $t = \mathrm{conn}(D_{s+s'})$ **to** $\mathrm{maxt}(s)$ **begin**
    compute $Image(d : C_{st} \to C_{s-1,t})$;
    For $g_1 \in Ext^{s+s',t+t'}$ **begin**
      $x := 0$;
      $dx := \widetilde{g^*}(\mathrm{diff}(g_1))$;
      Reduce (x, dx) against $Image$;
      $\widetilde{g^*}(g_1) := x$;
    end $\{g_1\}$;
  end $\{t\}$;
end $\{s\}$.

To compute the image, we use the same loop as in the homology program, so we have abbreviated it here. Note that in order to compute $\widetilde{g^*}$ on

D_{st}, we must have already computed it on $D_{s-1,t}$ and hence on D_{s-1,t_1} for $t_1 \leq t$. Any ordering of the computation which ensures this will be correct. However, the order we have used has the advantage that for each s, the only differentials needed are those on $D_{s+s'}$ and C_s, and the only values of $\widetilde{g^*}$ needed are those on D_{s-1}. This cuts memory requirements at a very minor I/O cost: we read in the differentials we need at the beginning of each s loop, and purge the values no longer needed at the end. In the homology program, the fact that we also have to compute the kernel of the differential forces us to vary s faster than t, making this kind of saving impossible.

4 Toda brackets

If we compute products as Yoneda composites of chain maps, it is natural to replace Massey products by Toda brackets. To get the signs right with a minimum of clutter, we find the following conjugation operation useful.

Definition 4.1 *If $c : W \to X$ is a degree s homomorphism of graded A-modules $(c_i : W_i \to X_{i-s})$, let $\overline{c} : W \to X$ be the homomorphism with i^{th} component $(\overline{c})_i = (-1)^i c_i$.*

Clearly conjugation is linear in c and is its own inverse. Also, one can quickly verify that

$$\overline{ab} = a\overline{b} = (-1)^{deg(b)}\overline{a}b.$$

Thus, a is a chain map iff its conjugate is an "anti-chain map":

$$da = ad \iff d\overline{a} = -\overline{a}d.$$

Similarly, for a chain null-homotopy $x : a \simeq 0$,

$$dx + xd = a \iff d\overline{x} - \overline{x}d = \overline{a}.$$

It follows that two null-homotopies of the same map differ by the conjugate of a chain map:

$$dx + xd = dy + yd \iff d\overline{(x-y)} = \overline{(x-y)}d.$$

Definition 4.2 *The* suspension ΣW *of a chain complex* W *has* $(\Sigma W)_i = W_{i-1}$ *and the same differential as* W. *The* mapping cylinder $C(c)$ *of a chain map* $c : W \to X$ *of degree* s *is the complex with* i^{th} *component* $C(c)_i = X_i \oplus W_{i-s-1}$ *and differential*

$$\begin{pmatrix} d & \overline{c} \\ 0 & d \end{pmatrix}.$$

The natural inclusion $i : X \to C(c)$ and projection $\pi : C(c) \to \Sigma W$ are chain maps, and

$$W \xrightarrow{c} X \xrightarrow{i} C(c) \xrightarrow{\pi} \Sigma W$$

is a cofibration sequence.

Definition 4.3 *If* $W \xrightarrow{c} X \xrightarrow{b} Y \xrightarrow{a} Z$ *are chain maps, the* Toda bracket $< a, b, c >$ *is the set of all chain maps* T *such that the following diagram homotopy commutes for some chain map* H.

$$\begin{array}{ccccccc} W & \xrightarrow{c} & X & \xrightarrow{b} & Y & \xrightarrow{a} & Z \\ & & & \searrow i & \uparrow H & & \uparrow T \\ & & & & C(c) & \xrightarrow{\pi} & \Sigma W \end{array}$$

Proposition 4.4

(1) $< a, b, c >= \{T \mid T \simeq a\overline{y} - x\overline{c}$ *where* $b' \simeq b$, $x : ab' \simeq 0$ *and* $y : b'c \simeq 0\}$.

(2) $< a, b, c >= \{T \mid T \simeq a\overline{y} - x\overline{c}$ *where* $x : ab \simeq 0$ *and* $y : bc \simeq 0\}$.

(3) $< a, b, c >$ *depends only on the homotopy classes of* a, b, *and* c.

(4) *Up to chain homotopy, the indeterminacy* $\{f - g \mid f, g \in < a, b, c >\}$ *is* $a[W, Y] + [X, Z]c$, *where* $[-, -]$ *denotes the set of chain homomorphisms.*

Proof: (1) First, suppose given chain maps H and T, and homotopies $\lambda : Hi \simeq b$ and $\mu : T\pi \simeq aH$. Write $H = (b', \overline{y})$ and $\mu = (-x, c)$. Then we find that $\lambda : b' \simeq b$, that

H is a chain map $\iff$ b' is a chain map and $y : b'c \simeq 0$,

and that

$$\mu : T\pi \simeq aH \Longleftrightarrow x : ab' \simeq 0 \text{ and } \phi : T \simeq a\overline{y} - x\overline{c}.$$

Conversely, if $\lambda : b' \simeq b$ and $y : b'c \simeq 0$ then $H = (b', \overline{y})$ is a chain map and $\lambda : Hi \simeq b$. If, in addition, $x : ab' \simeq 0$ then $a\overline{y} - x\overline{c}$ is a chain map. Finally, if $\phi : T \simeq a\overline{y} - x\overline{c}$, then $(-x, \phi) : T\pi \simeq aH$.

(2) Suppose give λ, x, and y as in (1). Then $y - \lambda c : bc \simeq 0$ and $x - a\lambda : ab \simeq 0$. Thus, $a\overline{(y - \lambda c)} - (x - a\lambda)\overline{c}$ is in the right hand side of (2). But $a\overline{y} - x\overline{c} = a\overline{(y - \lambda c)} - (x - a\lambda)\overline{c}$ so (1) and (2) agree.

(3) For b, this follows from (1). For a, suppose $\lambda : a' \simeq a$, $x : ab \simeq 0$, and $y : bc \simeq 0$. Then $x + \lambda b : a'b \simeq 0$, so

$$a'\overline{y} - (x + \lambda b)\overline{c} \in <a', b, c> \quad \text{and} \quad a\overline{y} - x\overline{c} \in <a, b, c>,$$

and the map $\lambda\overline{y}$ is a chain homotopy between these. Similarly, if $\lambda : c' \simeq c$, then $x\overline{\lambda}$ is the chain homotopy we need.

(4) By (2) we may assume $f \simeq a\overline{y} - x\overline{c}$ and $g \simeq a\overline{y_1} - x_1\overline{c}$, where x and x_1 are null homotopies of ab, and y and y_1 are null homotopies of bc. Then

$$\begin{aligned} f - g &\simeq a(\overline{y} - \overline{y_1}) - (x - x_1)\overline{c} \\ &= a\overline{(y - y_1)} + (-1)^{deg(c)}\overline{(x - x_1)}c \end{aligned}$$

which, by the observation preceding Definition 4.2, has the form claimed.

This description is rather extravagant computationally. We do not need the entire chain map defining the Toda bracket, only the cocycle it lifts. This is expressed by the following corollary.

Corollary 4.5 *Let a be a cocycle and $\tilde{a} : Y \to Z$ a chain map covering a. Let $W \xrightarrow{c} X \xrightarrow{b} Y$ be chain maps. If $y : bc \simeq 0$ then the map $a\overline{y}$ is a cocycle representing an element of $<\tilde{a}, b, c>$. Thus, the coefficient of g^* in $<\tilde{a}, b, c>$ is the coefficient of a in $\overline{y}(g)$.*

Proof: The component of the null-homotopy $x : \tilde{a}b \simeq 0$ which maps into Z_0 is 0. Therefore, the cocycle corresponding to the chain map $\tilde{a}\overline{y} - x\overline{c}$ is $a\overline{y}$.

Thus, computing a null-homotopy of bc gives us all Toda brackets of the form $<a, b, c>$. This is particularly useful in computing periodicity

operators in the cohomology of the Steenrod algebra and its subalgebras since they can be expressed as

$$P^n(x) = <x, h_0^{2^n}, h_n> .$$

At first glance, this Corollary appears to say that the null homotopy of $\tilde{a}b$ is irrelevant and only that of bc matters. However, the asymmetry is more apparent than real. The homomorphism $\tilde{a}\overline{y}$ is not a chain map, and in order to lift the cocycle $a\overline{y}$ to a chain map we have to subtract $x\overline{c}$ from $\tilde{a}\overline{y}$. Thus, the complementary term $x\overline{c}$ is implicit in the cocycle which we have shown represents the Toda bracket.

As in the case of products, elements of $H^1(A)$ have special behaviour which reduces the amount of computation needed to find Toda brackets involving them. Retain the assumptions made preceding Lemma 3.2 about $d : C_1 \to C_0$ and recall that $d(h_i) \equiv a_i \ (mod\ I^2)$.

Corollary 4.6 $< \widetilde{h_i^*}, \tilde{b}^*, c > = (a_i b)^* \overline{c}$. *That is, the coefficient of* g^* *in* $< \widetilde{h_i^*}, \tilde{b}^*, c >$ *is the coefficient of* $a_i b$ *in* $\overline{c}(g)$.

Proof: We use the preceding Corollary and the same k-linear maps as in the corresponding result for products:

$$h_i^* \overline{y} = a_i^* d\overline{y} = a_i^* \tilde{b}^* \overline{c} = (a_i b)^* \overline{c},$$

where the middle equality follows from the general formula $d\overline{y} = \overline{y}d + b\overline{c}$ and the fact that the component of $\overline{y}$ mapping into C_0 is 0.

The algorithm for calculating a null-homotopy y of $\tilde{b}^*\tilde{c}^*$ is not significantly different than the algorithm for computing an induced chain map. We initialize the calculation by setting $y = 0$ on $D_{s'+s''-1}$, where $s' = \deg(b)$ and $s'' = \deg(c)$, and then proceed to lift $\tilde{b}^*\tilde{c}^* - yd$ over the differentials $d : C_{st} \to C_{s-1,t}$.

5 Representation of the algebra

We assume given an ordered k basis $\{a_i\}$ for A such that each degrree is linearly ordered (and hence, well ordered, since A has finite type). In terms of this basis, there are two natural representations for the elements of A:

sparse case an ordered list $((k_{i_1}\ a_{i_1})\ (k_{i_2}\ a_{i_2})\ \ldots)$ of the nonzero terms, and

dense case a vector $(k_1\ k_2\ \ldots)$ of the coefficients for *all* the basis elements in the degree under consideration.

Whichever of these representations we use, the operations that we need to perform are:

1. produce the basis for a given degree,
2. compare elements to determine which has the higher highest term,
3. add and subtract elememts,
4. multiply elements.

We find it convenient to deal with the two cases (sparse and dense) separately. We have used both in our calculations involving the mod 2 Steenrod algebra, and have found the density of nonzero coefficients in the minimal resolutions we have constructed such that the dense representation uses roughly 1/4 the storage of the sparse representation. In general, the dense case representation is more compact iff $jd < CN$, where

j is the number of bits per coefficient in the dense representation,

d is $\dim_k A_n$, where n is the degree in question,

C is the average number of nonzero coefficients in the algebra elements under consideration, and

N is the number of bits required to hold a coefficient and an identifier for a basis element of A_n in the sparse case.

Clearly C will be difficult to compute exactly and will have to be estimated. Also, the sparse representation can be made more efficient in its use of space by reducing N, but this may make everything else more difficult.

5.1 The dense case

It is trivial to produce the vector representations of the basis elements, if we know the dimension of the algebra in the degree of interest. They are $(1,0,0,\ldots)$, $(0,1,0,\ldots)$, etc.

Comnparison for order is similarly easy, since the highest term is the last one with a nonzero coefficient. In our current implementation, we have an order preserving function which assigns to an element an integer identifying the last nonzero coefficient. With each entry (a, da) in *Image* we also store the integer identifying the highest coefficient of da. Then Reduce only has to search dx to determine whether to replace (x, dx) by $(x - a, dx - da)$, to look at the next entry in *Image*, or to terminate the search. Note that after replacing (x, dx) by $(x - a, dx - da)$, the highest operation decreases, so the search for it can start at the old highest operation, rather than at the end.

Addition (and subtraction) are somewhat different for $k = Z_2$ and for $k = Z_p$, $p > 2$. When $p = 2$, the vector of coefficients is a sequence of bits, and addition is exclusive-or of these bitstrings. This can generally be accomplished in comparatively few instructions even for long bitstrings. When $p > 2$, if we allow each coefficient one bit more than it must have, i.e. n bits, where $2^{n-2} < p < 2^{n-1}$, then we may pack these into words (16, 32 or 64 bits typically) and add without danger of "interference" between adjacent coefficients. Adding $(2^{n-1} - p,\ 2^{n-1} - p,\ \ldots)$ then causes the leftmost bit to reflect the need (or lack of it) for reduction mod p in that coefficient, which may then be carried out a word at a time by a short sequence of logical and arithmetic operations.

Multiplication is the most difficult operation in this representation. Note that, aside from needing to know $\dim_k A_n$ for each n, the first three operations are perfectly generic. The multiplication routine is the opposite: it is completely specific to the algebra. In order to implement it, we find that we alter the way in which the first three operations are carried out somewhat. By linearity, it is suficient to be able to multiply basis elements. Typically, to multiply a_i and a_j, given only i and j, we must first produce some more intrinsic representation of a_i and a_j than their *sequence numbers* i and j. We then apply the multiplication routine specific to the algebra to those intrinsic representations. We generally receive the answer

in intrinsic terms, which must then be converted back into sequence numbers. In principle then, we must be able to compute $Opno(i)$, the intrinsic representation of a_i, and $Seqno(op)$, the sequence number of the intrinsic representation op. In practice, a_i and a_j are not randomly distributed, so we take a slightly different approach. In the cohomology program, a_i is being stepped through the entire basis for its degree, one element at a time, while a_j runs through the nonzero terms of some element. In the products and Toda brackets programs, both a_i and a_j run through the nonzero terms of an element. Thus the functions we use are

Firstop(n), which produces the intrinsic representation of the first operation of degree n,

Nextop(op), which produces the intrinsic form of the successor to op, and

Advance(op,k), which produces the intrinsic form of the operation k steps beyond op.

We use $Firstop$ and $Nextop$ to produce each basis element in a given degree. We use $Advance$ to skip through an element's nonzero terms, since it is usually faster to produce the intrinsic form of a term from that of the preceding nonzero term, than to produce it from the sequence number. Depending on the algebra, we may define one of $Nextop$ and $Advance$ in terms of the other.

For the transformation from intrinsic form to sequence number, we use

Seqno(op) = sequence number of the basis element with intrinsic representation op, and

Rseqno(op_1,op_2) $= Seqno(op_1) - Seqno(op_2)$.

(of course, for efficiency, we probably will not compute $Rseqno$ by this formula.) The point of $Rseqno$ is that, for the Steenrod algebra at least, and certainly in many other cases, it is possible to devise the multiplication routine so that it produces the terms in its answer in roughly ascending order. If we already know the sequence number of the preceding operation, it may be quite a bit faster to compute only the relative sequence number of the next term and add, rather than compute its sequence number from scratch.

5.2 The sparse case

Here we must assign an identifier, which we take to be an integer, to each a_i. It is convenient to use a "packed" version of the intrinsic form for speed of conversion in the multiplication routine.

We order $\{a_i\}$ by numerical order of their identifiers. Of course, we will try to arrange the packed version so that this coincides with an ordering that makes sense for the algebra. For example, with the mod 2 Steenrod algebra's monomial basis $\{\xi_1^{r_1}\xi_2^{r_2}\dots\xi_n^{r_n}\}$, we use the identifier

$$Pack(r_1,\ r_2,\ \dots, r_n) = r_1 + 2^8(r_2 + 2^7(r_3 + 2^6(r_4 + 2^4(r_5 + 2^3(r_6 + 2^2 r_7))))),$$

which is easily computed by a sequence of shift operations. Numerical order of the packed forms then coincides with reverse lexicographic ordering of the monomials based on the ordering $\xi_1 < \xi_2 < \dots$ of generators. Note that this function is only one-to-one through dimension 240, so will have to be replaced if we ever exceed that dimension.

Of course, we also need the inverse function *Unpack*, which takes an integer identifier and produces the corresponding intrinsic form of the basis element.

To produce the basis for a given degree, the operations $Firstop(n)$ and $Nextop(op)$ composed with *Pack* will work fine. Note that *Nextop* must return some indication when its argument is the last basis element in its degree.

Comparison of elements is just lexicographic order of lists based on the standard order for the integers:

$$(i_1\ i_2\ \dots) < (j_1\ j_2\ \dots) \Longleftrightarrow i_1 < j_1 \text{ or } (i_1 = j_1 \text{ and } (i_2\ \dots) < (j_2\ \dots)).$$

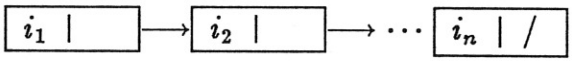
Addition (or subtraction) is a kind of merge with cancellations, since the lists are presumed ordered: we perform a merge of the identifiers i_j and i'_j of the lists

$$((k_1\ i_1)\ (k_2\ i_2)\ \dots) \text{ and } ((k'_1\ i'_1)\ (k'_2\ i'_2)\ \dots)\ ,$$

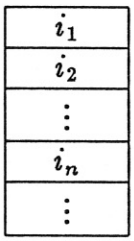
where $k_i,\ k'_i \in Z_p$ and $i_1 < i_2 < \cdots$, and $i'_1 < i'_2 < \cdots$, adding coefficients when the same identifier occurs in both, and eliminating the term if the sum of the coefficients is 0. When $k = Z_2$ we omit the coefficients k_i,

naturally, and eliminate terms which occur in both lists (we're computing the symmetric difference of sets in this case).

Multiplication is slightly simpler than in the dense case. We simply pass through the list, unpacking each item in order to multiply. The basis elements making up the product are packed and inserted in order into the list making up the sum so far. Note that the efficiency of this last insertion is enhanced if we can arrange the multiplication routine to produce its results in order, or in reverse order, depending upon the precise method we use to store lists. If we use a LISP style arrangement



then reverse order is best, since each insertion requires only 1 comparison to verify that it goes at the front. If we use contigous allocation,



increasing order is probably best, because insertion at the end is most efficient. (Insertion anywhere else requires copying whatever follows.)

5.3 Polynomial and Copolynomial algebras

Note that if A or the dual of A is a polynomial algebra then the manipulation of the basis elements consists of standard operations on weighted partitions. In particular, this is the case for the Steenrod algebra and its subalgebras.

6 Representation of modules

6.1 Free modules

The modules which make up the resolution are free, so are especially easy to represent. If $\{h_i\}$ is a well-ordered A-basis for the free A-module C, then we represent elements of C by ordered lists $((op_1\ h_{i_1})\ (op_2\ h_{i_2})\ \ldots)$, where $h_{i_1} < h_{i_2} < \ldots$ and the op_j are algebra elements (*not* just basis elements) represented as in the previous section. Such a list represents the obvious sum $\sum op_j h_{i_j}$.

Addition of such elements is essentially the same as addition of algebra elements in the sparse representation. To act on such an element by an element of the algebra is just the obvious bilinear extension of the multiplication of algebra elements.

6.2 General modules

The module being resolved is defined to the programs by two routines: one writes a k-basis for a specified degree of the module on the file which holds *Newker*, and the other returns the result of letting a basis element of the algebra act on an element of the module.

We represent *all* module elements in the same format as free module elements since this means that the same addition routines and I/O routines can be used. Conceptually, this amounts to representing a module as a quotient of a free module. The routine which gives a k-basis for the module is equivalent to giving a k splitting of the quotient homomorphism.

A general technique is to represent the module M as the quotient of the free A module on a k basis $\{m_i\}$ of M. Each module element then has a canonical form $((1\ m_{i_1})\ (1\ m_{i_2})\ \ldots)$, where 1 is the identity element of A. To define the action of A on M then requires that we specify the elements $a_i m_j$ in this form, for each a_i in the k basis of A.

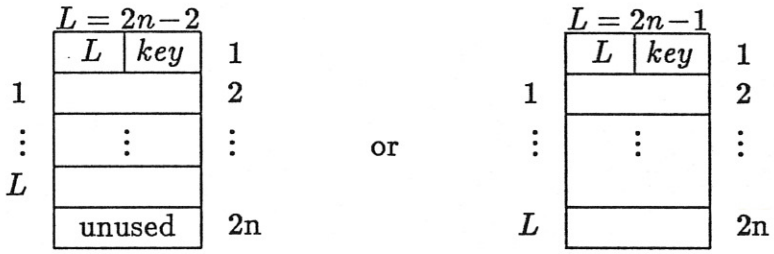
6.3 Special cyclic modules

Cyclic modules which are at most one dimensional (over k) in each degree, can be represented in a much simpler fashion, as quotients of A. If m is the

generator of the module, then to define the k basis for degree $t + \deg(m)$ of the module, we need only specify a basis element a_{i_t} of A such that $a_{i_t} m \neq 0$. To define the action of A on M, we give, for each degree t, the projection $A_t \to k$ defined by letting A_t act on m and recording the coefficient of the result. When $k = Z_2$ this is a bitstring of length $\dim_k A_t$. Then, to compute a_i acting on $((op\ m))$ we compute the product $a_i * op$ and apply the projection. When $k = Z_2$, this consists of a logical-and of the bitstring representing the product and the bitstring representing the projection, followed by counting (mod 2) the number of nonzero bits. The result is then the result of the projection times $((a_{i_{t'}}\ m))$, where $t' = deg(op) + deg(a_i)$.

7 Data structures

We have found that a single datatype, which we call a *block*, suffices for all our dynamically allocated memory. It consists of an even number, say $2n$, of contiguous 4 byte words, which we think of as split into two half-words (2 bytes each), followed by $2n - 1$ full words. The first two bytes contain either $2n - 1$ or $2n - 2$, reflecting the number of words of data following the first word of the block. The second two bytes of the block (i.e., the last half of the first word) also hold data, generally of a different nature than the data held in the full words. We generally call these two bytes the *key*, and the first two bytes, the *length*.

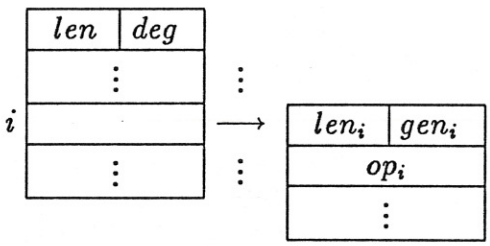


The use of these blocks to represent lists has the advantage of keeping elements of a list contiguous, speeding access (via binary search or hashing) and allowing the use of machine operations which act on large blocks of memory. Of course, it also introduces the need for memory management which we discuss below, and means that inserting elements into the middle

of lists requires some copying.

We use a single block to represent an element of the algebra, in either the sparse form or the dense form. In the sparse form, each word of the block holds an identifier for a basis element. In the dense form, the words in the block are thought of as one long string of bits. In either case the *key* field is not used by the algebra element. Instead, it is used to hold an identifier for a module generator, so that a block can represent a term of the form $op * gen$.

To represent an element of a module, we use a block whose entries are themselves (pointers to) blocks representing terms $op * gen$ as just described. The *key* of the main block is set equal to the internal degree of the module element. Thus, the structure



represents the element

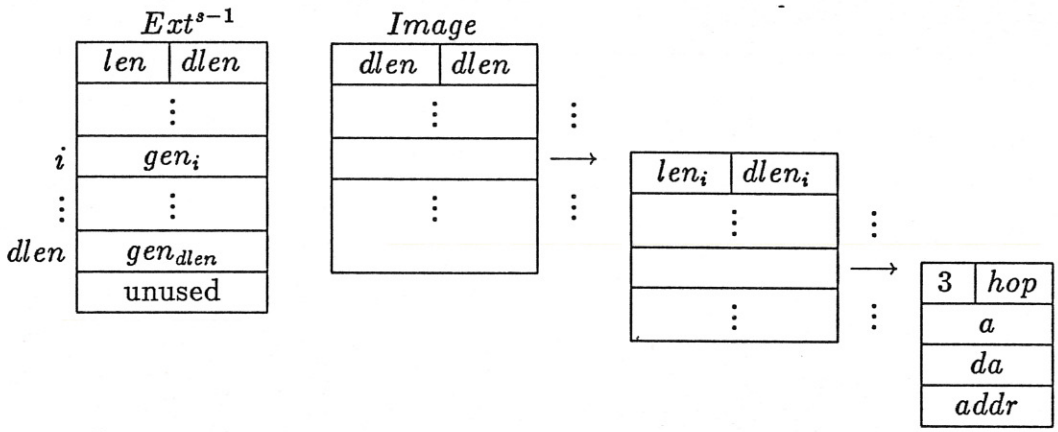
$$\sum_i op_i * gen_i$$

of degree *deg*.

The lists Ext^s of generators for the resolution in cohomological degree s are examples of lists which need to be able to grow. To keep such lists in a single block without having to recopy them each time an element is added, we use a block whose *length* field is the full length $2n-1$ of the block, while the *key* field is set to the actual data length of the block. When the block is full, we copy it into a new block with some room to spare for expansion.

We store *Image* as a block whose i^{th} entry is itself a block holding the list of (a, da) entries with highest generator of da equal to the i^{th} generator in Ext^{s-1}. Thus, *Image* is a block whose length is the same as the data length of Ext^{s-1}. When we are reducing (x, dx), we find the highest generator *gen* of dx in Ext^{s-1} by binary search, and look at the block located at the corresponding position in *Image*. This block is an expandable block, each entry of which corresponds to an (a, da) pair, with the entries arranged in increasing order of the highest operation *hop* on *gen*. Thus, another binary search determines whether or not *Image* contains an (a, da) pair whose highest term $hop * gen$ agrees with the highest term of dx. Each (a, da)

pair is stored in a block of length 3, along with the disk address *addr* at which a and da were written, and an identifier *hop* for the highest operation in da, so that we may compare it to the highest operation of dx without having to look at da. This is primarily important if the pair (a, da) has been paged out, since it means we may determine whether or not we need (a, da) to reduce (x, dx) without having to read (a, da) back in. Thus, we will not read it back in unless we actually need it.



Organizing *Image* in this way has two significant beneficial effects. First, searching *Image* for an (a, da) pair with highest term the same as the highest term of (x, dx) is faster because, when one is found, and (x, dx) is replaced by $(x - a, dx - da)$, we can continue our search inside the subblock if the highest generator is not changed, and we can immediately skip the rest of the subblock if it is. Second, when a new (a, da) entry is to be inserted, only the part of the *sub*block following it has to be moved down. This can significantly reduce the amount of copying necessary.

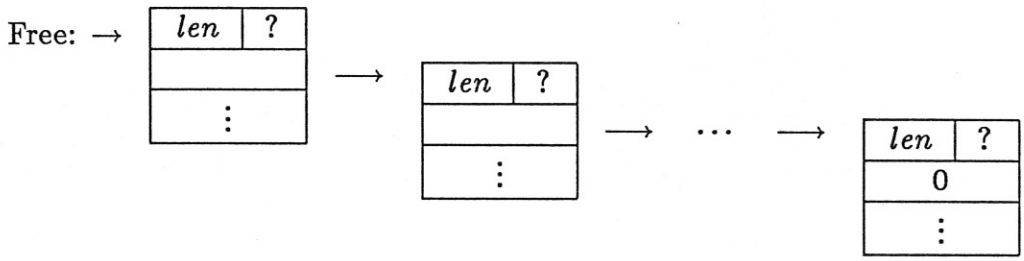
The only other dynamically allocated data structures needed are *Newker* and *Oldker*. Since these are generated and used in one sequential pass, they are simply written on and read from sequential files.

8 Memory management

We maintain a free list of unused blocks in order of address. This permits amalgamation of adjacent free blocks. We allocate space from this list by first fit; that is, the block requested is carved from the first block large

enough to hold it and the remainder (if any) is left in place. An exact length or a range of acceptable lengths can be requested. The latter form is used for lists that will expand (such as Ext^s) and for blocks that are likely to soon have a piece removed from the end. For example, in computing the sum of elements of lengths i and j, we request a block of length $i+j$ to hold the sum. However, cancellations may mean that the sum will have fewer than $i+j$ entries and the remainder of the block will have to be freed. Thus, if a block of length $i+j+2$ is available, it makes more sense to take all of it than to chop off two words now, then perhaps chop off a few more momentarily, especially since they will likely have to be reattached to the two originally removed.

A block in the free list must contain at least two words in order to hold its length information and the address of its successor. Blocks are allocated in multiples of 2 words to ensure this.



If less than an entire block is being returned, it is taken from the end of the first large enough block. In this way, no pointers in the free list have to be changed.

Our original memory allocation system also maintained separate lists of common size blocks. This eliminated the need to search the free list when a block of such a size was requested, and eliminated the need to chop larger blocks into pieces. However, it increased fragmentation to such an extent that it was counterproductive. (Segregating the free blocks into separate lists of different sizes eliminated much of the amalgamation that takes place when all the blocks are in the same free list.)

To improve locality, a region called the *clear* area is maintained. Initially, the free list is empty and the clear area consists of all space allocated for blocks. If a request for a new block cannot be granted from the free list, the allocation routine checks to see if it can be granted from the clear

area. If so, the requested amount of space is taken from there. Similarly, if a block located on the edge of the clear area is freed, the clear area is expanded to include it, instead of adding it to the free list. A similar effect could be achieved by taking small blocks from the beginning rather than the end of oversize blocks, or by ordering the free list in reverse order. However, with only 16 bits for the length field of the block, we are limited to blocks whose length is 64K words, quite a bit less than the initial size of memory requested for some computations, which would force us to begin by carving it into pieces. If memory were requested from the operating system in 64K word (or smaller) segments, this latter method of encouraging locality would be the method of choice.

When space for a new block is requested and no space is available, we begin paging out parts of *Image*, since this is what uses most of memory. The scheme we have adopted is rather crude but effective. We remove from memory the last 1/4 of the entries in each of *Image*'s subblocks. More precisely, we remove *a* and *da* but keep the block containing *hop*, *addr* and space to put pointers to *a* and *da* if they are returned to memory. If that is sufficient, we stop there. If not, we remove the 1/4 before that, etc. An entry is returned to memory only when it is again needed. This works as well as it does largely because we have also arranged to produce elements in reverse order, so that their position in *Image* can be rapidly established. We should also point out that when we remove an entry from memory, no I/O is required because every entry is written out as it is generated, and will never change. Thus, only half of the I/O cost commonly associated with paging is incurred.

Finally, if the algebra is actually finite dimensional, not just of finite type, we remove differentials that can no longer contribute to the calculation. (If we have reached internal degree t, and the algebra is 0 above degree t' then differentials of elements in degrees less than $t - t'$ can have no further effect.)

9 I/O

Module elements have a linear form which consists of the first word of their main block, containing their length and degree, followed by the contents of

their subblocks, in order. When writing a block of odd total length, i.e., even data length, the unused last word is omitted. Writing this to a file is elementary. Similarly, reading it is easy. The first word tells us the size block to allocate and the number of further blocks to read. For each of those, the first word we read tells us how many more words we need to read.

The files holding *Oldker* and *Newker* consist of a header identifying the bidegree for which it is the kernel, following by a stream of elements spanning the kernels linearized in the above fashion. The end is indicated by an element of length 0. At the end of the computation for each bidegree the files holding *Oldker* and *Newker* swap roles.

The other files we use are all random access files. These are

Diff which holds the differentials defining a resolutiom,

Map which holds the chain maps lifting cocycles corresponding to a set of generators of a particular homological degree,

Image which holds the image of the differential $d : C_{st} \to C_{s-1,t}$ while that image is in use.

Each of these contains a header that tells

- the number of words in the file,
- the number of words in the header (allowing for a variable length description of the file located between this header and the main body),
- the last bidegree completed,
- the module(s) involved, and
- (optional) text describing the contents of the file.

In addition, the header for a *Map* file gives the number n, homological degree s, and identifiers $g_1, \ldots, g_n$ of the generators of D whose induced chain maps $\widetilde{g_i^*} : D_{s+s'} \to C_{s'}$ are contained in thr file.

Following the header, the file is simply a sequence of individual entries. In *Diff* the entries have the form $(g, s, t, diff)$, where g is the identifier for a

generator of the resolution, s and t are its homological and internal degrees, and *diff* is the linearization of the differential on g. In *Map* the entries have the form (i, g, elt), where *elt* is the linearization of $\widetilde{g_i^*}(g)$. In *Image* the entries have the form (dx, x) where dx and x are the linearizations of the two elements. We put dx first because in the second phase of the homology calculation the x entries are no longer needed. Thus, if we need to page dx back in, its address is the same as the address of the pair.

The functions needed to carry out the I/O are

- *Read* or *Write* the linearization of an element at the current position in a sequential (kernel) file,
- *Rewind* a sequential file,
- *Read* or *Write* the linearization of an element at a specified address in a random access file, and report the address of the next word following it, and
- *Read* or *Write* a vector of specified length at a specified address in a random access file.

10 Results

The previous version (using the sparse representation of algebra elements) was used to calculate a minimal resolution of the Steenrod algebra through internal degree 69, together with the chain maps induced by indecomposables in homological degrees 3 to 6 through this range.

With the current version of the program (using the dense representation of algebra elements), we have computed

- a minimal resolution of the Steenrod algebra through internal degree 62,
- a minimal resolution of $A(2)$, the subalgebra of the Steenrod algebra generated by $Sq^1, Sq^2,$ and Sq^4, through internal degree 90,
- minimal resolutions for $H^*(\mathbf{R}P^\infty/\mathbf{R}P^{n-1})$ over $A(2)$, for $n = 1, 3, 5, 7$, through internal degree $n + 45$,

- a minimal resolution for a cyclic $A(2)$ submodule of $H^*(MO<8>)$, through internal degree 80, and
- chain maps for indecomposables in low homological degrees for the resolutions of $A(2)$ and the submodule of $H^*(MO<8>)$.

Our next projects are to extend the calculation of the minimal resolution of the Steenrod algebra and compute all products and many Toda brackets in it, to calculate a minimal resolution of $A(3)$, to calculate minimal resolutions for some modules over the Steenrod algebra of interest in our other work, and to calculate minimal resolutions of the Steenrod algebra for the primes 3,5 and 7. By the time this appears, much of this will undoubtedly have already been accomplished.

11 Acknowledgements

Professor Neil Rickert was helpful in innumerable ways, both in dealing with the MVS and CMS systems, and in suggesting and discussing ways to make the algorithms and data organization more efficient. This work was also aided by the excellent computer facilities and staff at the University of Illinois at Chicago. In particular, Dr. Nora Sabelli was always quick to solve problems that arose.

I am also grateful to the Research Office and the Computer Center at Wayne State University for funding.

Robert R. Bruner
Department of Mathematics
Wayne State University
Detroit, Michigan 48202
USA