SHEAFHOM: Software for Sparse Integer Matrices

MARK MCCONNELL

Sheafhom is a free software package for large-scale computations in the category of finitely-generated modules over the integers and related rings. Its front end is a language for problems in algebraic topology and geometry. These problems come down to sparse systems of linear equations over the integers. Sheafhom's back end solves the sparse systems, with emphasis on avoiding fill-in and integer explosion. We survey and compare algorithms for integer sparse matrices, and we present implementation techniques in Common Lisp. The final section finds the quotient of the free abelian group on 26 letters by sets of words in the dictionary, in the spirit of [8].

With warm regards for Bob MacPherson on his sixtieth birthday.

1. Introduction

Sheafhom is a free software package for large-scale computations in the category of finitely-generated R-modules, where R is any principal ideal domain in which exact computation is possible. At present Sheafhom focuses on $\mathbf{Z}$-modules, and in this paper R equals $\mathbf{Z}$ except when noted. However, the system is adaptable to any R, as described in 4.2.

Sheafhom's front end is a language for problems in algebraic topology and geometry. These problems come down to large sparse systems of linear equations over $\mathbf{Z}$, which are solved by the back end. The major goal for the back end is to avoid fill-in and integer explosion. In Section 2 we introduce these terms. The heart of the paper is Section 3, where we survey and compare integer sparse matrix algorithms. The next section discusses implementation techniques in Common Lisp. Section 5 describes the goals of different versions of Sheafhom. The final section is a tutorial: it finds the quotient of the free abelian group on 26 letters by sets of words in the dictionary, in the spirit of [8].

The sites http://www.lispwire.com and http://www.geocities.com/mmcconnell17704/math.html are Sheafhom's home pages. They offer descriptions of the code, a tutorial, and download links for the source code and manuals.[1]

The development of Sheafhom 1.x was partially supported by grants from the National Science Foundation (1994-97 and 1997-2000).

2. Background on Sparse Linear Algebra

2.1 Linear Algebra Over the Integers

Round-off error is always an aspect of floating-point computations. To solve the system

$$21x + 18z = 0$$

$$28x - 3y + 24z = 0$$

we would row-reduce the matrix

[1] This paper is a snapshot of Sheafhom in October 2005. Like all large programs, Sheafhom continues to evolve, with improvements to the fundamentals and with new layers added to the top. Please consult the websites for up-to-date information.

21	0	18
28	–3	24

We might divide the first row by 21

1	0	0.8571
28	–3	24

and subtract 28 times the top row from the bottom row

1	0	0.8571
0	–3	0.0012

(Here 21 is the *pivot*.) The trouble is, the 0.0012 should be zero.

There is no error using rational numbers:

$$24 - 28 \times (18/21) = 0.$$

However, the rationals can be too slow for large-scale computations, because of the need to carry around denominators and reduce fractions (18/21 = 6/7).

We now turn to linear algebra over **Z**. Let's find the greatest common divisor of 21 and 28 by the Euclidean algorithm:

$$28 \div 21 = 1 \text{ remainder } 7$$

$$21 \div 7 = 3 \text{ remainder } 0$$

The GCD is 7. In our matrix example, carry out the analogous steps by subtracting 1 times the first row from the second

21	0	18
7	–3	6

and then subtracting 3 times the second from the first.

0	9	0
7	–3	6

This clears out the left-hand column, filling it with 0's except for the pivot 7. It also gives a zero in the upper right corner, avoiding the round-off error with 0.0012.

Divide the top row by 9, and use the resulting 1 to clear the –3 out of the bottom row.

0	1	0
7	0	6

The general solution is thus $y = 0$ and $7x + 6z = 0$. If x, y and z must be integers themselves, the general solution is

$$(x, y, z) = (-6k, 0, 7k) \quad \text{for all } k \text{ in } \mathbf{Z}.$$

2.2 Conventions

For simplicity, our exposition in this paper discusses only row reduction. Sheafhom actually performs row and column reduction, converting matrices to *Smith normal form* [3, Def. 2.4.11]. This means the resulting matrix has 0's everywhere besides the diagonal, and has integer diagonal entries d_1, d_2, d_3, … with $d_1 \mid d_2$ and $d_2 \mid d_3$, etc.

In the example above, we reduced matrices by taking the columns in order from left to right. In fact, Sheafhom uses complete pivoting: it permutes rows and columns to bring a better selection of pivot into the row and column being reduced. Pivoting is the topic of Section 3.

2.3 Fill-In

A *sparse matrix* is one where only a small fraction of the entries are non-zero. A matrix coming from topology as in [1] might have tens of thousands of rows and hundreds of thousands of columns, but only about 100 non-zeros in each row and column. For such matrices, one uses data structures that store only the non-zero entries.

Many scientific applications involve sparse matrices whose non-zero elements form a pattern: tridiagonal, banded diagonal, banded diagonal with a border, and so forth. Sheafhom was designed for matrices coming from topology that have no recognizable sparsity pattern. A matrix coming from an n-dimensional topological space would have a pattern in n dimensions, but not when the data is flattened into a two-dimensional table of rows and columns.

Fill-in is a concern in sparse linear algebra using any number system. Imagine row-reducing the following matrix, where the letters are arbitrary non-zero numbers and all x's are distinct.

$$\begin{matrix} a & x & 0 & 0 & x & x \\ x & 0 & x & x & 0 & 0 \\ x & 0 & x & x & 0 & x \\ b & 0 & 0 & x & 0 & x \end{matrix}$$

We want to choose one row and add multiples of it to the other rows until the first column has only a single non-zero entry. If we choose the top row (a is the pivot), the result will in general look like this, where * is an entry that has changed from zero to non-zero.

$$\begin{matrix} a & x & 0 & 0 & x & x \\ 0 & * & x & x & * & * \\ 0 & * & x & x & * & x \\ 0 & * & 0 & x & * & x \end{matrix}$$

If b is the pivot, the result will be

$$\begin{matrix} 0 & x & 0 & * & x & x \\ 0 & 0 & x & x & 0 & * \\ 0 & 0 & x & x & 0 & x \\ b & 0 & 0 & x & 0 & x \end{matrix}$$

The second result has less fill-in, two *'s rather than seven. All other things being equal, *b* is the better choice of pivot. If we do not keep fill-in in check, the sparse matrix will quickly become dense and exceed the resources of the computer.

2.4 Integer Explosion

Floating-point work must balance fill-in with numerical stability. In the example, if *b* is much smaller than *a*, using *b* as pivot may introduce too much round-off error. We may be forced to use *a*, with its larger fill-in.

Over the integers, numerical stability is replaced with the opposite problem: integer explosion. In integer multiplication, the length (number of digits) of the product is roughly the sum of the lengths of the factors. Thus a row operation that puts 0's into one column will tend to increase the length of the numbers in the rest of the row. Even in our small integer example, the –3 grew quickly to a 9. Near the end of a full Smith normal form computation, matrix entries often have hundreds or thousands of digits.

3. Algorithms for Sparse Linear Algebra

3.1 A Survey of Algorithms

A sparse solver must reduce both fill-in and integer explosion by choosing the pivots appropriately. Sheafhom is a platform for experiments with pivoting strategies. We now discuss and compare some of these strategies.

3.1.1 The Markowitz Algorithm

The Markowitz algorithm [4, 7.2] is a well-established approach to reducing fill-in. It is a greedy algorithm, reducing fill-in as much as possible at each step. Let a_{ij} be a non-zero matrix entry. Let r_i be the number of non-zero entries in a_{ij}'s row, and c_j the number of non-zero entries in its column. If one adds a multiple of row i to row k in order to zero out the a_{kj} entry, one will create as many as $r_i - 1$ new non-zeroes in row k, coming from the $r_i - 1$ entries of row i besides a_{ij} itself. Using row i to clear out the jth column will produce as many as $(r_i - 1)(c_j - 1)$ new entries. The Markowitz algorithm, in its simplest form (what we call *pure Markowitz*), chooses at each step the pivot a_{ij} that minimizes the *Markowitz count* $(r_i - 1)(c_j - 1)$. We always exclude the region of the matrix that has already been reduced. The algorithm uses the pivot to clear out the jth column, then goes on to the next j.

3.1.2 Sheafhom 2.1

Sheafhom 2.1, which is the stable version as of October 2005, essentially uses the pure Markowitz algorithm. In addition, though, we need to consider integer explosion, as in the following example.

$$\begin{matrix} 100 & 99 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 55 & 49 & 56 & 69 & 51 & 48 \end{matrix}$$

Among pivots in the left column, 100 has the lowest Markowitz count, but we cannot use it over the integers because 1 and 55 are smaller than 100. Instead, we choose the 1, obtaining

0	–1	–100	–100	–100	–100
1	1	1	1	1	1
0	–6	1	14	–4	–7

Sheafhom 2.1's pivoting algorithm is to search at each stage for pivots of the smallest possible absolute value, and among these to search for the pivot with the smallest Markowitz count.

In the topology problems Sheafhom was designed for, the non-zero entries in the sparse matrices are 1's and –1's, or occasionally a bit larger. Throughout the computation, the vast majority of entries have absolute value 1 until about 5% of the rows remain. Thus the condition "smallest possible absolute value" is not burdensome till the end of the computation, when the matrix has become quite dense anyway.

3.1.3 Speeding Up a Markowitz Search

It can be slow to compute the Markowitz count for all a_{ij}, at least by the time fill-in has become fairly bad. One approach, suggested by several authors, is to look at only a certain number of rows with small r_i—say the smallest ten rows—and then minimize the Markowitz count for only the a_{ij} in those rows. Early predecessors of the Sheafhom code used this approach. Currently, though, we prefer to avoid fill-in at whatever cost in speed, and we search over all a_{ij}.

3.1.4 Triangular Markowitz Sort

A variation on the Markowitz algorithm, due to the author, aimed to be faster while still controlling fill-in. Sort the columns so those with the smaller c_j are on the left. (As always, we tacitly exclude columns that have already been reduced.) If two columns have the same c_j, find the one whose first non-zero entry has greater index i, and put that column on the left. The result looks like a series of triangular strips, with very sparse strips on the left and denser strips on the right.

0	0	0	0	0	*
0	0	0	*	*	0
0	*	*	0	0	*
*	0	*	0	*	*
*	*	0	*	*	*

For a genuine picture, see Figure 1.

To find a pivot, search the columns from left to right, and return an element of minimal absolute value. The matrix usually has many ±1 entries, and the search terminates as soon as one of these is found. We expect the pivot to have a low Markowitz count, assuming it was found quickly on the far left side of the sorted matrix. The method is faster than pure Markowitz, at least for mild to moderate fill-in, for two reasons. First, the $O(n \log n)$ sort is faster than the $O(n^2)$ search over all a_{ij}. Second, if the pivot is found quickly, it means that essentially all the work went into the $O(n \log n)$ sort.

3.1.5 LLL to Avoid Integer Explosion

The Lenstra-Lenstra-Lovasz (LLL) algorithm [3, 2.6-2.7] reduces an integer matrix using a strategy different from the methods so far. Rather than clear out the columns one after the other, it makes all the columns as short as possible and makes different columns as orthogonal to each other as possible. The notions "short" and "orthogonal" are defined with respect to a fixed Euclidean inner product, which we take to be the standard one. The algorithm works strictly over **Z** [3, Alg. 2.7.2]. Finding absolutely the best columns satisfying these conditions would take exponential time. Part of LLL's value is that its definition of "reduced matrix" is good enough for most purposes and yet allows the algorithm to run in polynomial time.

As a by-product of making the columns small in Euclidean norm, LLL helps control integer explosion. Just as significant, it finds the linear dependencies among the columns. Our matrices usually have far from full rank. LLL throws away columns that are dependent, saving a great deal of memory. However, LLL tends to make fill-in worse among the columns it does not throw away. The definition of "LLL-reduced" allows columns on the right to be longer than those on the left. In our examples, the short columns on the left have many 0's, and their non-zero entries tend to be in the single digits. On the right side, however, columns have large Euclidean norm and tend to be dense.

3.1.6 Triangular Markowitz Sort with LLL

Sheafhom 2.0's reduction algorithm combined the ideas above into several steps.

(0) Let S be a variable representing an algorithm for sorting the columns of a matrix and choosing a pivot. Initialize S to the algorithm S_{Mar} = "perform a triangular Markowitz sort and choose a pivot as in 3.1.4".

(1) Sort the matrix and choose a pivot using S. Clear out the pivot's column, and advance to the next column.

(2) If fill-in has grown so that the matrix is no longer sparse enough—say, when 10% of its entries are non-zero—go to step 3. If $S \neq S_{\text{Mar}}$, go to step 3. Otherwise, go to step 1.

(3) Use LLL to reduce integer explosion. In practice, doing LLL on more than a few hundred columns at a time is very slow, because the intermediate integer results are huge. Hence we do LLL reduction on blocks of 200 columns at a time, moving from left to right. Before doing LLL, we sort all the columns by Euclidean length, so that the first block of 200 will contain relatively short vectors, the next block slightly longer vectors, etc. We ignore columns of zeroes at the left (and LLL fortunately tends to produce many zero columns in our matrices).

(3a) If this is the first time for LLL reduction, perform step 3 a second time: sort by Euclidean length, then do LLL 200 columns at a time. Since the first block of 200 in the previous step contained relatively short vectors, this helps make the second, third, and later blocks relatively short.

(4) Set S to the new algorithm S_{LLL} = "sort the columns by Euclidean length, then scan through all the entries from left to right, choosing as pivot an entry of minimal absolute value." The reason is that once LLL has run, sparsity is largely destroyed. The sort then aims to preserve the LLL structure rather than the sparse structure.

(5) Perform step 1 up to 50 times, or until the whole matrix is reduced.

(6) If the whole matrix is reduced, exit. Otherwise, go to step 2.

It is interesting to compare this algorithm with 3.1.2's. On the first few large examples in our experiments, 3.1.2 has proved to offer the better method. The triangular Markowitz sort introduces fill-in a little bit faster than the pure Markowitz algorithm, since pure Markowitz is so miserly. Integer explosion also seems to occur a bit more slowly with pure Markowitz, perhaps because fill-in occurs more slowly. When LLL starts up in step 3, it's "too much, too little, too late": LLL can be very slow, and replacing sparse columns with dense ones can strain memory resources.

For these reasons, the pure Markowitz algorithm is our method of choice in Sheafhom 2.1. Experiments with better algorithms will continue. One open question is whether there is better way to combine pure Markowitz with LLL.

3.2 Graphical Tools

Sheafhom 2.x offers graphical tools for studying fill-in and integer explosion. There are line graphs showing the growth of the sparsity percentage and the numbers of row and column operations. Sheafhom can also generate movies showing the sparsity pattern changing in real time. These movies have been indispensable in carrying out the comparative studies above. Figure 1 shows one frame of a movie for a 545×2146 matrix. The matrix arises in the computation of the equivariant cohomology of one of the locally symmetric spaces in [1]. Each pixel represents a 3×3 submatrix, and is dark or light in proportion to how many of the nine entries are non-zero. The curves are the tops of the triangular strips of 3.1.4.

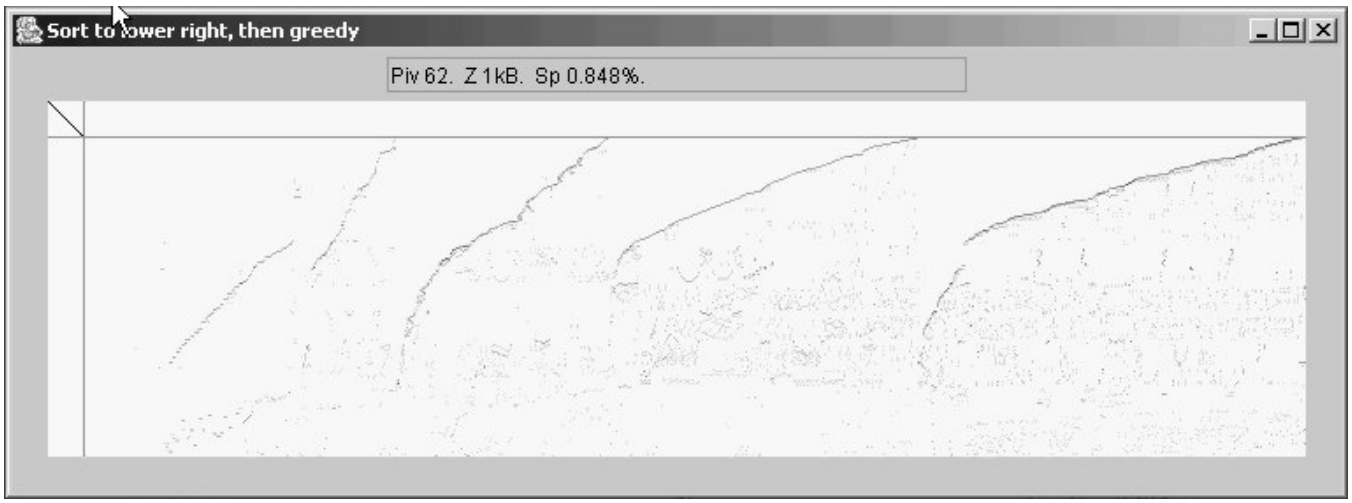

Figure 1. Sparsity pattern for a triangular Markowitz sort.

3.3 Other Approaches

One can consider sparse matrices modulo an integer N. Here there is no problem with integer explosion. Fill-in, of course, is an ever-present concern. One spectacular use of these methods was for the RSA-129 challenge [11], where the final step was to solve a sparse matrix mod 2 with half a million rows and columns. The matrix was reduced by the sparse techniques in [9]. Early code of the author's followed [9] also.

Some authors approach sparse matrices over **Z** by working modulo prime numbers p. One can reduce the matrix mod p for several different p's and reconstruct the integer result at the end. For an example, see [5]. For a similar idea with dense matrices, see [2].

One can solve mod p matrices with Lanczos mod p. Lanczos methods [6, Chs. 9—10] are best known over the real or complex numbers as techniques for finding eigenvalues or solving matrix equations. They are iterative methods that do not change the matrix itself, but run through a convergence process in some small auxiliary vectors of storage. There is no fill-in at all. Lanczos methods rely on a fixed positive-definite inner product, say $\sum x_i y_i$, and use the notion of orthogonal vectors. When one works mod p, it makes sense to say $\sum x_i y_i$ defines a notion of orthogonality, because the quadratic form is non-degenerate. However, it no longer makes sense to say it is positive definite, and this obstacle can make Lanczos methods mod p fail if applied directly. The paper [1] worked around this obstacle in one way. A systematic way of overcoming the obstacle appears in [10]. In the future, we may introduce Lanczos mod p techniques into Sheafhom, at least for the largest matrices.

3.4 Size Records

Working over **Z** and using the algorithm of 3.1.2, Sheafhom 2.1 has reduced a matrix of size 8407 × 32826. This took about half an hour and peaked at about 151M of RAM usage. The intermediate integers never grew larger than four digits.[2] We used Allegro Common Lisp 7.0 on a Compaq laptop running Windows XP. The laptop had 1G of RAM and a Celeron M processor with a speed of 1.30 GHz. Using the algorithm of 3.1.6, Sheafhom 2.0 was not able to reduce matrices of even half this number of rows and columns, and even when the upper bound on the Java virtual machine's heap size was set to 160M.

In [1], we used Lanczos methods modulo the prime 31991 to reduce matrices as large as 110464 × 30836. We did not attempt to lift the solutions to **Z**.

4. Sheafhom and Lisp

A mathematician should program in Lisp. –Bob MacPherson, ca. 1984

4.1 Implementations

Common Lisp is an ideal language for Sheafhom for several reasons. Arbitrary-precision integers are built into the language and can be very fast. Lisp is object-oriented; indeed, its object package CLOS is the most flexible of any language's. The crowing glory of Lisp is its macros, which allows one to redefine the syntax of the language itself. For introductions to Lisp, see the comic-book site http://www.lisperati.com and the practical textbook http://www.gigamonkeys.com/book.

Sheafhom 2.1 is written entirely in ANSI Common Lisp. The only exception is the graphics file `gui.lisp`, which calls out to Java to draw pictures like Figure 1. Strictly speaking, we do not call out to Java, but to Linj, a Lisp-like language created by António Menezes Leitão that compiles to high-quality Java source code.

Sheafhom 2.1 was developed in Allegro Common Lisp 6.2 and 7.0. Version 1.x was developed using Carnegie-Mellon University Common Lisp.

---

[2] Other examples have produced integers of up to 1000 digits, but in our experience such examples do not run to completion.

4.2 No One Ring to Rule Them All: Low-Level Arithmetic

Many computer algebra programs treat the underlying ring as a plug-and-play part of the system. The same code for (say) Gröbner bases works with coefficients in a wide range of rings, and one can change the ring as one goes along. The philosophy is object-oriented: a polynomial leaves it up to its coefficients to add and multiply themselves together.

Other forces pull in the opposite direction. While Lisp is a high-level language, its users are proud of how efficient its compiled code can be down at the level of machine language. If the programmer commits in advance to which ring they will use, the efficiency will be greater. This is because the programmer can declare to the compiler which variables are 32-bit integers, which are arbitrary-precision integers, and so on. Macros help to hide the low-level details of these declarations, and Lisp's `disassemble` command shows what effect the declarations have in each compiled function.

In the latter spirit, the Lisp versions of Sheafhom have stepped away from the object-oriented approach to rings. The challenge is to keep open the possibility of using different rings while still preserving efficiency. Sheafhom 2.1 handles the issue with the data types `sparse-elt` and `sparse-v`. A `sparse-v`, or sparse vector, is a simply-linked list (the classic Lisp `list`) whose elements are `sparse-elt`s. A `sparse-elt` is a data structure holding an $i \geq 0$ and a v; here i means this is the ith element of the vector, and v is a value in the underlying ring. The elements always appear in the list in order of increasing index, and an element of value zero never appears.

All ring operations are expressed as operations on `sparse-elt`s. For instance, the addition operator `sp-add` takes as input two `sparse-elt`s. It returns a new `sparse-elt` whose value is the sum of the given values, and whose index is derived from the given indices (typically just the index of the first argument).

Operators like `sp-add` are optimized for the specific number type of the chosen ring, like arbitrary-precision integers for work over **Z**, or 32-bit integers for work modulo small primes. The ring operations are separated into one marked section of the source files. The rest of the code, including the main vector and matrix operations, uses the ring operations as black boxes.

Writing Sheafhom 2.0 in Java was instructive in this regard. Almost everything in Java is object-oriented, so—if the ring was to be changeable at all—we were forced to implement numbers as instances of a class. It was a small step to package the vector index together with the ring element. We defined an abstract class `SparseElt` holding index and value, and concrete subclasses to handle values and arithmetic in specific rings. To use memory efficiently, we had to introduce two classes representing **Z**, one for 32-bit `int`s and one for the arbitrary-precision `BigInteger`. We wrote our own code to convert the sum or product of two `int`s to a `BigInteger` when necessary. It was a relief to come back to Lisp, which switches from small to large integers automatically.

In Sheafhom 2.1, a `sparse-elt` over **Z** is a `cons` (an ordered pair) holding the index and value as its two elements. By contrast, a `sparse-elt` mod 2 could be implemented as a single non-negative 32-bit integer i, standing for index i and value 1. We would need a convention for occasionally passing around elements of index i and value 0; since that convention could hardly fail to be ugly, we would hide it under macros.

4.3 Vectors and Matrices

A sparse matrix is a data structure `csparse` that is basically an array holding the columns of the matrix. Each column is a `sparse-v`, essentially a list. The list structure is flexible: columns grow in

size when fill-in occurs, and shrink when matrix entries cancel each other out. When they shrink, the memory is reclaimed by the garbage collector.

A design decision in Sheafhom is that, above the lower levels, speed will be less important than avoiding fill-in and integer explosion. We use heavy computational algorithms like Markowitz sorting and LLL, even while we carefully optimize the low-level routines.

4.4 The Macro `with-splicer`

Implementing sparse vectors as singly-linked lists is flexible, as we have said, but it carries the usual risk of accesses taking $O(n)$ time. To find the sum **a** + **b** of sparse vectors **a** and **b**, we should run exactly once through each vector. We also want to alter **a** destructively rather than allocate the top-level `cons`es from scratch; for instance, changing the first non-zero entry of **a** to a different non-zero value should amount to changing only the first `cons`. Sheafhom 2.1 includes a macro `with-splicer` as a mini-language for this kind of surgery on lists. `with-splicer` iterates down a list and offers read, insert, modify and delete operations for individual elements. The rest of the iteration is not disturbed. The macro also offers splicing commands to add and remove larger chunks.

Here is an example using `with-splicer`. The function `sqrt-if-real` deletes elements of `arglist` unless they are non-negative real numbers, in which case they are overwritten with their square root.

```
(defun sqrt-if-real (arglist)
  (with-splicer arglist
    (loop until (splicer-endp) do
      (let ((x (splicer-read)))
        (cond ((and (realp x) (>= x 0))
               (splicer-modify (sqrt x))
               (splicer-fwd))
              (t
               (splicer-delete))))))
  arglist)

(sqrt-if-real (list -1 4 9 'a -81 144))
 ⇒ (2.0 3.0 12.0)
```

5. History

Sheafhom 1.x was developed in 1993-99 in Common Lisp. It was also ported to Macaulay 2, an elegant computer algebra system for commutative ring theory and algebraic geometry written by Dan Grayson and Mike Stillman [7]. Sheafhom 2.0 was written in 2001-2004 in Java and released at the time of Bob MacPherson's birthday conference. The system moved back to Common Lisp for Sheafhom 2.1, which was released in March 2005.

Versions of Sheafhom through 2.1 worked with **Q**-vector spaces rather than **Z**-modules, though the computations were over **Z**. They could report on the torsion components of certain objects, but could not handle the torsion as an object in its own right. Sheafhom 2.2, which works fully with finitely-generated **Z**-modules, is currently under development.

As the name suggests, Sheafhom was originally about sheaves. Sheafhom 1.x allowed one to construct sheaves of **Q**-vector spaces, and complexes of sheaves, on topological spaces that could be expressed in finite terms. The spaces included simplicial complexes, regular cell complexes, and the skeleta of various combinatorial spaces. In effect, Sheafhom 1.x worked in the derived category of complexes of sheaves on these topological spaces. Toric varieties were a prominent example: the system computed their intersection cohomology in arbitrary perversity, as well as perverse sheaves and morphisms between these objects.

Sheafhom 2.x has produced new results in a research project of Avner Ash, Paul Gunnells and the author, which will appear in early 2007. The previous published results from this project are at [1]. Publication of the new results is anticipated.

6. Quotients modulo the Dictionary

We now move to a tutorial session illustrating Sheafhom's capabilities. The entertaining article [8] looks at the free group on the letters A through Z modulo equivalence of words that have the same pronunciation in French. For instance, *soie* = *soi* implies *e* = 1. The authors prove the quotient is trivial.

Let's solve a related problem: find the quotient of the free abelian group on A through Z by the relations $W = 0$, where W runs through a complete English dictionary. If the quotient is too boring, we can also try subsets of the dictionary.

We'll use the dictionary at http://www.itasoftware.com/careers/WORD.LST, a Lisp-related site with several enjoyable programming puzzles. In this dictionary, no hyphens or other punctuation marks appear, just the letters A through Z. Words of length one are excluded because they make combinatorial problems like this one too easy.

We load the words into a list for flexibility. We find there are 173528 words (is it a coincidence that $\sqrt{3} = 1.7320508$?) We examine the first few words and the 100000^{th}.

```
CL-USER(17): (defconstant +words+
               (let ((result '()))
                 (with-open-file (stream "WORD.LST" :direction :input)
                   (loop
                     (let ((symbol (read stream nil 'end-of-file)))
                       (if (eq symbol 'end-of-file)
                           (return (nreverse result))
                         (push (symbol-name symbol) result)))))))
+WORDS+
CL-USER(18): +words+
("AA" "AAH" "AAHED" "AAHING" "AAHS" "AAL" "AALII" "AALIIS" "AALS"
 "AARDVARK" ...)
CL-USER(19): (length +words+)
173528
CL-USER(20): (nth 100000 +words+)
"NONJURY"
```

What's the range of word lengths?

```
CL-USER(21): (remove-duplicates (sort (mapcar #'length +words+) #'<))
(2 3 4 5 6 7 8 9 10 11 ...)
```

While we're at it, what is the longest word?

```
CL-USER(22): (last *)
(28)
CL-USER(23): (remove-if-not #'(lambda (w) (= (length w) 28)) +words+)
("ETHYLENEDIAMINETETRAACETATES")
```

ANTIDISESTABLISHMENTARIANISM is also 28 letters long, but isn't in this dictionary for some reason.

Let's return to the problem on free abelian groups. We'll create a 26×173528 matrix with one column for each word. The column for DAD, written horizontally, is 1 0 0 2 0 … 0, standing for one A and two D's.

Here is a utility for converting A through Z to the row indices 0 through 25.

```
CL-USER(24): (defconstant +code-A+ (char-code #\A))
+CODE-A+
CL-USER(25): (defun az-to-int (ch)
               "Converts ASCII A to 0, B to 1, ..., Z to 25."
               (- (char-code ch) +code-A+))
AZ-TO-INT
```

The next function takes a list of words and returns the matrix. `(make-csparse-zero m n)` creates an $m \times n$ matrix of zeroes. `(csparse-incf A i j)` adds 1 to the (i,j) entry of matrix A.

```
CL-USER(26): (defun az-list-to-csparse (words)
               (let ((result (make-csparse-zero 26 (length words)))
                     (j 0))
                 (dolist (word words result)
                   (map nil #'(lambda (ch)
                                (csparse-incf result (az-to-int ch) j))
                        word)
                   (incf j))))
AZ-LIST-TO-CSPARSE
```

Let's apply this to the whole dictionary.

```
CL-USER(27): (compile 'az-to-int) ;; to speed things up
AZ-TO-INT
NIL
NIL
CL-USER(28): (compile 'az-list-to-csparse)
AZ-LIST-TO-CSPARSE
NIL
NIL
CL-USER(29): (defvar words-all (az-list-to-csparse +words+))
WORDS-ALL ;; after a little while
```

```
CL-USER(30): words-all
[CSPARSE 26 by 173528]
```

Finally, we find the Smith normal form (SNF) of the matrix.

```
CL-USER(31): (let ((*show-stats* t))
               (make-snf words-all nil nil t))
[SNF 26 by 173528, [twenty-six units] [zero 0s]]
```

The last line says the SNF has nothing but twenty-six units down the diagonal. This proves

Theorem 6.1. *The free abelian group on A through Z modulo the dictionary is trivial.*

When `*show-stats*` is true (`t`), `make-snf` pops up a line graph showing how the pattern of sparsity changed during the computation. The graph shows that the matrix started with 27.8% of its entries non-zero. The sparsity decreased slightly while the first 20 of the 26 rows were being reduced, then increased till the end.

As we've suggested, the trivial group is boring. What is the quotient of A through Z modulo the two-letter words? The quotient will be at least $\mathbf{Z}/2\mathbf{Z}$ for parity reasons: no word with an odd number of letters can be equivalent to 0. But are there other invariants?

```
CL-USER(32): (remove-if-not #'(lambda (w) (= (length w) 2)) +words+)
("AA" "AB" "AD" "AE" "AG" "AH" "AI" "AL" "AM" "AN" ...)
CL-USER(33): (length *)
96

CL-USER(34): (az-list-to-csparse **)
[CSPARSE 26 by 96]
```

Setting `*show-csw*` true generates the movie of how the sparsity pattern changes as the SNF is computed. `csw` stands for *csparse window*.

```
CL-USER(35): (let ((*show-csw* t))
               (make-snf * nil nil t))
[SNF 26 by 96, [twenty-one units] -2 [four 0s]]
```

The torsion coefficient 2 (up to sign) is the one we predicted. But the matrix has rank 22. Hence there are four linearly independent words that are not combinations of two-letter words. The quotient is $\mathbf{Z}^4 \oplus (\mathbf{Z}/2\mathbf{Z})$.

What about four-letter words?

```
CL-USER(36): (remove-if-not #'(lambda (w) (= (length w) 4)) +words+)
("AAHS" "AALS" "ABAS" "ABBA" "ABBE" "ABED" "ABET" "ABLE" "ABLY" "ABOS"
 ...)
CL-USER(37): (az-list-to-csparse *)
[CSPARSE 26 by 3919]
CL-USER(38): (make-snf * nil nil t)
[SNF 26 by 3919, [twenty-five units] 4 [zero 0s]]
```

We have torsion of order 4, again for parity reasons. But that is all we have: the quotient by four-letter words is $\mathbf{Z}/4\mathbf{Z}$, as small as possible.

What about lengths greater than or equal to some cut-off value? The same methods show that the quotient by words of length 20 or more is trivial. But for length 21 or more, the quotient is **Z**.

```
CL-USER(39): (remove-if-not #'(lambda (w) (>= (length w) 21)) +words+)
("ACETYLCHOLINESTERASES" "ADRENOCORTICOSTEROIDS"
 "ADRENOCORTICOTROPHINS" "ANTHROPOMORPHIZATIONS"
 "ANTIAUTHORITARIANISMS" "ANTIFERROMAGNETICALLY"
 "BUCKMINSTERFULLERENES" "CARBOXYMETHYLCELLULOSE"
 "CARBOXYMETHYLCELLULOSES" "CLINICOPATHOLOGICALLY" ...)
CL-USER(40): (make-snf (az-list-to-csparse *) t nil t)
[SNF 26 by 118, [twenty-five units] [one 0]]
```

The function `shh::coker-section` picks out a generator for the quotient by choosing a section for the quotient map.

```
CL-USER(41): (shh2::coker-section *)
```

The result is a column vector with all entries 0 except for a 1 standing for the letter Q. We've proved that CARBOXYMETHYLQCELLULOSE is not a word.

7. References

[1] Ash, A., Gunnells, P., and McConnell, M. Cohomology of congruence subgroups of SL(4,**Z**). *J. Number Theory 94*, 1 (2002), 181-212.

[2] Ash, A. and McConnell, M. Doubly cuspidal cohomology for principal congruence subgroups of GL(3, **Z**). *Math. Comput. 59*, 200 (Oct. 1992), 673-688.

[3] Cohen, H. *A Course in Computational Algebraic Number Theory* (Springer Grad. Texts in Math. 138). Springer-Verlag, Berlin/Heidelberg, 1993.

[4] Duff, I. S., Erisman, M., and Reid, J. K. *Direct Methods for Sparse Matrices* (in *Monographs on Numerical Analysis*). Clarendon Press, Oxford, 1989.

[5] Dumas, J.-G., Saunders, B. D., and Villard, G. On efficient sparse integer matrix Smith normal form computations. *J. Symb. Comput. 32* (2001), 71-100.

[6] Golub, G. H. and Van Loan, C. F. *Matrix Computations*, 3rd ed. Johns Hopkins Univ. Press, Baltimore, 1996.

[7] Grayson, D. and Stillman, M. *Macaulay 2*. Home page at http://www.math.uiuc.edu/Macaulay2/

[8] Mestre, J.-F., Schoof, R., Washington, L., and Zagier, D. Quotients Homophones des Groupes Libres: Homophonic Quotients of Free Groups, *J. Exper. Math. 2*, 3 (1993), 153-155.

[9] Pomerance, C. and Smith, J. W. Reduction of huge, sparse matrices over a finite field via created catastrophes. *J. Exper. Math. 1* (1992), 90-94.

[10] Teitelbaum, J. Euclid's algorithm and the Lanczos method over finite fields. *Math. Comput. 67*, 224 (1998), 1665-1678.

[11] Wright, D. Web page at http://www.math.okstate.edu/~wrightd/numthry/rsa129.html#answer

27 WOLF HILL DR., WARREN, NJ 07059; mmcconnell17704@yahoo.com