

```

/*
 * homology.c
 *
 * This file contains the following functions which the kernel
 * provides for the UI:
 *
 *      AbelianGroup      *homology(Triangulation *manifold);
 *
 *      AbelianGroup      *homology_from_fundamental_group(
 *                               GroupPresentation *group);
 *
 * If all Dehn filling coefficients are integers, homology() returns
 * a pointer to the first homology group of *manifold. Otherwise
 * it returns NULL. Note that homology() will compute homology groups
 * of orbifolds as well as manifolds.
 *
 * 96/12/11 homology() and homology_from_fundamental_group() now
 * check for overflows, and return NULL if any occur.
 *
 * The comments in homology() below describe the algorithm.
 *
 * homology() returns NULL if some Dehn filling coefficients
 * are not integers (or if overflows occur).
 *
 * homology_from_fundamental_group() is a variant of homology which
 * computes the homology by abelianizing a presentation of the
 * fundamental group.
 */

#include "kernel.h"

/*
 * To minimize the possibility of overflows, we use long integers instead
 * of regular integers to do the matrix computations. Take ENTRY_MIN
 * to be -LONG_MAX instead of LONG_MIN, to minimize the (unlikely)
 * possibility that negation causes an overflow (i.e. -LONG_MIN
 * = -(0x80000000) = (0x80000000) = LONG_MIN).
 */
typedef long int MatrixEntry;
#define ENTRY_MAX    LONG_MAX
#define ENTRY_MIN    (-LONG_MAX)

/*
 * The number of meaningful rows and columns in a RelationMatrix are
 * given by num_rows and num_columns, respectively. max_rows
 * records the original number of rows allocated, so we know how many
 * rows to free at the end.
 */
typedef struct
{
    int          num_rows,
                num_columns,
                max_rows;
    MatrixEntry **relations;
} RelationMatrix;

static void      group_to_relation_matrix(GroupPresentation *group, RelationMatrix *
    relation_matrix, Boolean *overflow);
static void      allocate_relation_matrix_from_group(GroupPresentation *group,
    RelationMatrix *relation_matrix);
static void      read_relations_from_group(GroupPresentation *group, RelationMatrix *
    relation_matrix, Boolean *overflow);

static void      find_relations(Triangulation *manifold, RelationMatrix *relation_matrix,
    Boolean *overflow);
static void      allocate_relation_matrix(Triangulation *manifold, RelationMatrix *
    relation_matrix);
static void      initialize_relations(RelationMatrix *relation_matrix);
static void      find_edge_relations(Triangulation *manifold, RelationMatrix *
    relation_matrix);
static void      find_cusp_relations(Triangulation *manifold, RelationMatrix *
    relation_matrix, Boolean *overflow);
static void      eliminate_generators(RelationMatrix *relation_matrix, Boolean *

```

```

        overflow);
static void      delete_empty_relations(RelationMatrix *relation_matrix);
static void      compute_homology_group(RelationMatrix *relation_matrix, AbelianGroup **
g, Boolean *overflow);
static void      add_row_multiple(RelationMatrix *relation_matrix, int src, int dst,
MatrixEntry mult, Boolean *overflow);
static void      add_column_multiple(RelationMatrix *relation_matrix, int src, int dst,
MatrixEntry mult, Boolean *overflow);
static MatrixEntry safe_addition(MatrixEntry a, MatrixEntry b, Boolean *overflow);
static MatrixEntry safe_multiplication(MatrixEntry a, MatrixEntry b, Boolean *overflow);
static void      free_relations(RelationMatrix *relation_matrix);

AbelianGroup *homology(
    Triangulation *manifold)
{
    Boolean      overflow;
    RelationMatrix relation_matrix;
    AbelianGroup *g;

    /*
     * Make sure all the Dehn filling coefficients are integers.
     */
    if ( ! all_DeHN_coefficients_are_integers(manifold) )
        return NULL;

    /*
     * Compute a set of generators.
     */
    choose_generators(manifold, FALSE, FALSE);

    /*
     * The overflow flag keeps track of whether an overflow has occurred.
     */
    overflow = FALSE;

    /*
     * Read the edge and cusp relations out of the manifold.
     * Each row of the relation_matrix represents a relation;
     * each column corresponds to a generator. For example,
     * the relation  $X_0 - X_5 + 2X_8 = 0$  would be encoded as the
     * row 1 0 0 0 0 -1 0 0 2 0. The edge relations are read
     * into the matrix before the cusp relations because they
     * are simpler (typically 3 to 5 nonzero entries); this
     * minimizes the fill-in during the presimplification phase
     * below.
     */
    find_relations(manifold, &relation_matrix, &overflow);
    if (overflow == TRUE)
    {
        free_relations(&relation_matrix);
        return NULL;
    }

    /*
     * Presimplify the relations.
     *
     * First check each relation to see whether some generator
     * appears with coefficient +1 or -1. If one does,
     * use the relation to substitute out the generator.
     * For example, in the matrix
     *
     *   0  0  1  0  0 -1  2  0  0  1  0
     *   0  1  0  0  0  0  0  1  1  0  0
     *   1  0  1  0  0  0  1  0  0  0  1
     *   0  1 -1  0  1  0  1  0  0  0  0
     *
     * we can use the first row to eliminate the third generator,
     * yielding
     *
     *   0  0  1  0  0 -1  2  0  0  1  0
     *   0  1  0  0  0  0  0  1  1  0  0    <-- no change
     *   1  0  0  0  0  1 -1  0  0 -1  1    <-- first row was subtracted
     *   0  1  0  0  1 -1  3  0  0  1  0    <-- first row was added

```

```

*
* We can now eliminate both the first row and the third column.
* In practice, the third column would be overwritten with the
* contents of the last column to keep the data contiguous.
* For the moment the first row is left with all zeros.
*
*   0  0  0  0  0  0  0  0  0  0
*   0  1  0  0  0  0  0  1  1  0
*   1  0  1  0  0  1 -1  0  0 -1
*   0  1  0  0  1 -1  3  0  0  1
*
* The cusp relations (which typically have many nonzero entries)
* appear at the bottom of the matrix, after the gluing relations
* (which typically have 3 to 5 nonzero entries, regardless of the
* size of the manifold). This minimizes the "fill-in" (overwriting
* of zeros with nonzero values) during the presimplification process.
*/
eliminate_generators(&relation_matrix, &overflow);
if (overflow == TRUE)
{
    free_relations(&relation_matrix);
    return NULL;
}

/*
* delete_empty_relations() now removes rows containing only
* zeros. This includes the rows of zeros created by
* eliminate_generators() as well as the rows which were initially
* all zeros.
*/
delete_empty_relations(&relation_matrix);

/*
* Apply a general algorithm to the relations to compute
* the homology group.
*
* The torsion coefficients of the homology group are the
* invariant factors of the relation matrix. (A torsion
* coefficient of 0 indicates an infinite cyclic factor.)
* That is, one performs row and column operations on the
* matrix to put it into diagonal form, then reads the
* torsion coefficients directly from the diagonal entries.
* It's not hard to convince oneself that both row and column
* operations are "legal"; for the full story on invariant
* factors, see Hartley & Hawkes' Rings, Modules and Linear
* Algebra, Chapman & Hall 1970.
*
* compute_homology_group() uses the following algorithm.
* It performs row and column operations as necessary to
* create a matrix element which divides its entire row
* and column. For example, if it starts with the element
* 10 in the first row (see below), it would notice that 10 does not
* divide the 28 in the first row, so it would subtract twice the
* second column from the last column, and continue with the 8
* at the end of the first row as its candidate. But this 8 does
* not divide the 10 in the first row, so it would subtract the
* last column from the second column, and continue with the 2
* as its candidate.
*
*   0 10  0 28           0 10  0  8           0  2  0  8
*   3  0  2  6           3  0  2  6           3 -6  2  6
*   2 10  4 16           2 10  4 -4          2 14  4 -4
*   0  2  0  8           0  2  0  4           0 -2  0  4
*
* The 2 in the first row now divides its entire row, and, as it
* turns out, its entire column as well. If it didn't divide each
* entry in its column, similar operations would be performed
* repeatedly until it divided both row and column. Note that
* the absolute value of the candidate decreases at each step,
* so this process is sure to succeed in a finite number of steps.
*
* Once we have an element that divides its row and column, we
* perform row operations to clear out its column
*

```

```

*      0  2  0   8
*      3  0  2  30 <--   3 times the first row was added
*      2  0  4 -60 <--  -7 times the first row was added
*      0  0  0  12 <--   1 times the first row was added
*
* and column operations to clear out its row
*
*      0  2  0   0
*      3  0  2  30
*      2  0  4 -60
*      0  0  0  12
*
* We can now read off a torsion coefficient (in this case, 2),
* and eliminate a row and column. In practice, the deleted
* row is swapped with the last row, and the deleted column
* is overwritten with the contents of the last column, to keep
* the matrix contiguous.
*
*      swap rows:           overwrite column:   reduce matrix dimensions:
*
*      0  0  0  12           0  12  0  12           0  12  0
*      3  0  2  30           3  30  2  30           3  30  2
*      2  0  4 -60           2 -60  4 -60           2 -60  4
*      0  2  0   0           0   0  0   0
*
* We repeat this process until no relations remain. If there
* are any generators left over, they will correspond to infinite
* cyclic factors of the group (represented as torsion coefficients
* of 0).
*/
compute_homology_group(&relation_matrix, &g, &overflow);
if (overflow == TRUE)
{
    free_relations(&relation_matrix);
    free_abelian_group(g);
    return NULL;
}

/*
* Clean up.
*/
free_relations(&relation_matrix);

return g;
}

AbelianGroup *homology_from_fundamental_group(
    GroupPresentation *group)
{
    /*
    * This function is a variation on the above homology() function.
    * The difference is that here we use a presentation of the fundamental
    * group as our starting point, instead of a triangulation.
    * Please see homology() for a complete explanation of the algorithm.
    */

    Boolean        overflow;
    RelationMatrix relation_matrix;
    AbelianGroup   *g;

    overflow = FALSE;

    group_to_relation_matrix(group, &relation_matrix, &overflow);
    if (overflow == TRUE)
    {
        free_relations(&relation_matrix);
        return NULL;
    }

    eliminate_generators(&relation_matrix, &overflow);
    if (overflow == TRUE)
    {
        free_relations(&relation_matrix);

```

```

        return NULL;
    }

    delete_empty_relations(&relation_matrix);

    compute_homology_group(&relation_matrix, &g, &overflow);
    if (overflow == TRUE)
    {
        free_relations(&relation_matrix);
        free_abelian_group(g);
        return NULL;
    }

    free_relations(&relation_matrix);

    return g;
}

static void group_to_relation_matrix(
    GroupPresentation *group,
    RelationMatrix *relation_matrix,
    Boolean *overflow)
{
    allocate_relation_matrix_from_group(group, relation_matrix);
    initialize_relations(relation_matrix);
    read_relations_from_group(group, relation_matrix, overflow);
}

static void allocate_relation_matrix_from_group(
    GroupPresentation *group,
    RelationMatrix *relation_matrix)
{
    int i;

    relation_matrix->max_rows = fg_get_num_relations (group);
    relation_matrix->num_rows = fg_get_num_relations (group);
    relation_matrix->num_columns = fg_get_num_generators(group);

    if (relation_matrix->max_rows > 0)
        relation_matrix->relations = NEW_ARRAY(relation_matrix->max_rows, MatrixEntry *);
    else
        relation_matrix->relations = NULL;

    for (i = 0; i < relation_matrix->max_rows; i++)
        relation_matrix->relations[i] = NEW_ARRAY(relation_matrix->num_columns,
MatrixEntry);
}

static void read_relations_from_group(
    GroupPresentation *group,
    RelationMatrix *relation_matrix,
    Boolean *overflow)
{
    int i,
        j,
        *relation;

    for (i = 0; i < relation_matrix->num_rows; i++)
    {
        relation = fg_get_relation(group, i);

        for (j = 0; relation[j] != 0; j++)
        {
            if (ABS(relation[j]) > relation_matrix->num_columns)
                uFatalError("read_relations_from_group", "homology");

            if (relation[j] > 0)
            {
                if (relation_matrix->relations[i][ relation[j] - 1] < ENTRY_MAX)
                    relation_matrix->relations[i][ relation[j] - 1]++;
                else

```

```

        *overflow = TRUE;
    }
    else /* relation[j] < 0 */
    {
        if (relation_matrix->relations[i][-relation[j] - 1] > ENTRY_MIN)
            relation_matrix->relations[i][-relation[j] - 1]--;
        else
            *overflow = TRUE;
    }
}

fg_free_relation(relation);
}
}

static void find_relations(
    Triangulation *manifold,
    RelationMatrix *relation_matrix,
    Boolean *overflow)
{
    allocate_relation_matrix(manifold, relation_matrix);
    initialize_relations(relation_matrix);
    find_edge_relations(manifold, relation_matrix);
    find_cusp_relations(manifold, relation_matrix, overflow);
}

static void allocate_relation_matrix(
    Triangulation *manifold,
    RelationMatrix *relation_matrix)
{
    int i;

    /*
     * There will be, at most, one relation for each EdgeClass and one
     * relation for each Cusp. We'll worry about the exact number of
     * relations later. By an Euler characteristic argument,
     * the number of EdgeClasses equals the number of Tetrahedra.
     *
     * relation_matrix->num_rows records the number of active rows,
     * which is initially zero.
     *
     * The number of generators is found in the manifold->num_generators field.
     */

    relation_matrix->max_rows = manifold->num_tetrahedra + manifold->num_cusps;
    relation_matrix->num_rows = 0;
    relation_matrix->num_columns = manifold->num_generators;

    /*
     * Allocate storage for the relations.
     */

    relation_matrix->relations = NEW_ARRAY(relation_matrix->max_rows, MatrixEntry *);

    for (i = 0; i < relation_matrix->max_rows; i++)
        relation_matrix->relations[i] = NEW_ARRAY(relation_matrix->num_columns,
MatrixEntry);
}

static void initialize_relations(
    RelationMatrix *relation_matrix)
{
    int i,
        j;

    for (i = 0; i < relation_matrix->max_rows; i++)
        for (j = 0; j < relation_matrix->num_columns; j++)
            relation_matrix->relations[i][j] = 0;
}

```

```

static void find_edge_relations(
    Triangulation    *manifold,
    RelationMatrix   *relation_matrix)
{
    EdgeClass        *edge;
    PositionedTet     ptet,
                    ptet0;
    MatrixEntry       *entry;

    /*
     * We compute a relation for each EdgeClass in the manifold.
     * The functions set_left_edge(), veer_left() and same_positioned_tet()
     * (from positioned_tet.c -- see documentation in kernel_prototypes.h)
     * walk a PositionedTet around an EdgeClass. At each step, we examine
     * the generator (see choose_generators.c) dual to ptet.near_face.
     */

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)
    {
        set_left_edge(edge, &ptet0);
        ptet = ptet0;
        do
        {
            entry = &relation_matrix->relations
                    [relation_matrix->num_rows]
                    [ptet.tet->generator_index[ptet.near_face]];

            switch (ptet.tet->generator_status[ptet.near_face])
            {
                case outbound_generator:
                    *entry += 1;
                    break;

                case inbound_generator:
                    *entry -= 1;
                    break;

                case not_a_generator:
                    /* do nothing */
                    break;

                default:
                    uFatalError("find_edge_relations", "homology");
            }

            veer_left(&ptet);

        } while ( ! same_positioned_tet(&ptet, &ptet0) );

        relation_matrix->num_rows++;
    }
}

static void find_cusp_relations(
    Triangulation    *manifold,
    RelationMatrix   *relation_matrix,
    Boolean          *overflow)
{
    Tetrahedron      *tet;
    VertexIndex       vertex;
    FaceIndex         side;
    Orientation        orientation;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)

        for (vertex = 0; vertex < 4; vertex++)
        {
            if (tet->cusp[vertex]->is_complete)
                continue;

```

```

    for (side = 0; side < 4; side++)
    {
        if (side == vertex)
            continue;

        if (tet->generator_status[side] != inbound_generator)
            continue;

        for (orientation = 0; orientation < 2; orientation++) /* orientation =
right_handed, left_handed */
        /*
         * Here's the version of the code which didn't
         * check for overflows:
         */
        {
            relation_matrix->relations
                [relation_matrix->num_rows + tet->cuspid[vertex]->index]
                [tet->generator_index[side]]
                += (MatrixEntry) tet->cuspid[vertex]->m * tet->curve[M][orientation]
[vertex][side]
                + (MatrixEntry) tet->cuspid[vertex]->l * tet->curve[L][orientation]
[vertex][side];
            *
            * Here's the current version:
            */
        {
            relation_matrix->relations
                [relation_matrix->num_rows + tet->cuspid[vertex]->index]
                [tet->generator_index[side]]
            = safe_addition(
                relation_matrix->relations
                    [relation_matrix->num_rows + tet->cuspid[vertex]->index]
                    [tet->generator_index[side]],
                safe_multiplication((MatrixEntry) tet->cuspid[vertex]->m,
(MatrixEntry) tet->curve[M][orientation][vertex][side], overflow),
                overflow);

            relation_matrix->relations
                [relation_matrix->num_rows + tet->cuspid[vertex]->index]
                [tet->generator_index[side]]
            = safe_addition(
                relation_matrix->relations
                    [relation_matrix->num_rows + tet->cuspid[vertex]->index]
                    [tet->generator_index[side]],
                safe_multiplication((MatrixEntry) tet->cuspid[vertex]->l,
(MatrixEntry) tet->curve[L][orientation][vertex][side], overflow),
                overflow);

            /*
             * If an overflow occurs, don't worry about it here.
             * Nothing bad will happen. Just keep going, and let
             * the calling function deal with it.
             */
        }
    }
}

relation_matrix->num_rows += manifold->num_cusps;
}

static void eliminate_generators(
    RelationMatrix *relation_matrix,
    Boolean *overflow)
{
    int i,
        j,
        ii,
        jj;
    MatrixEntry mult;

    /*
     * eliminate_generators() tries to substitute out as many generators
     * as possible. To do so, it looks at the rows of the matrix one

```



```

    * at a time, and whenever it finds a row containing +1 or -1 times
    * a generator, it eliminates that generator.
    */

    for (i = 0; i < relation_matrix->num_rows; i++)
        for (j = 0; j < relation_matrix->num_columns; j++)
            if (relation_matrix->relations[i][j] == 1
                || relation_matrix->relations[i][j] == -1)
            {
                /*
                 * Substitute this generator out of all the other rows.
                 */

                for (ii = 0; ii < relation_matrix->num_rows; ii++)
                    if (ii != i && relation_matrix->relations[ii][j])
                    {
                        mult = (relation_matrix->relations[i][j] == -1) ?
                            relation_matrix->relations[ii][j] :
                            - relation_matrix->relations[ii][j];
                        add_row_multiple(relation_matrix, i, ii, mult, overflow);
                        if (*overflow == TRUE)
                            return;
                    }

                /*
                 * Make this row all zeros.
                 */

                for (jj = 0; jj < relation_matrix->num_columns; jj++)
                    relation_matrix->relations[i][jj] = 0;

                /*
                 * Generator j may now be eliminated. Let the highest
                 * numbered generator inherit its index (i.e. overwrite
                 * column j with contents of the last column), and
                 * decrement num_columns.
                 */

                relation_matrix->num_columns--;

                for (ii = 0; ii < relation_matrix->num_rows; ii++)
                    relation_matrix->relations[ii][j] = relation_matrix->relations[ii]
[relation_matrix->num_columns];

                /*
                 * Break out of the j loop, and move on to the next i.
                 */

                break;
            }
    }

}

static void delete_empty_relations(
    RelationMatrix *relation_matrix)
{
    int i, j;
    Boolean all_zeros;
    MatrixEntry *temp;

    /*
     * Eliminate rows consisting entirely of zeros.
     */

    /*
     * For each row . . .
     */
    for (i = 0; i < relation_matrix->num_rows; i++)
    {
        /*
         * . . . check whether all entries are zero . . .

```

```

    */
    all_zeros = TRUE;
    for (j = 0; j < relation_matrix->num_columns; j++)
        if (relation_matrix->relations[i][j])
        {
            all_zeros = FALSE;
            break;
        }

    /*
     * . . . and if so, eliminate the row.
     */
    if (all_zeros)
    {
        /*
         * Decrement num_rows . . .
         */
        relation_matrix->num_rows--;

        /*
         * . . . and swap the zero row with the last active row
         * of the matrix. (Because num_rows was decremented,
         * the zero row will be "invisible" in its new location,
         * but its storage will still be there to be freed when
         * the time comes.)
         */
        temp
            = relation_matrix->relations[i];
        relation_matrix->relations[i]
            = relation_matrix->relations[relation_matrix->num_rows];
        relation_matrix->relations[relation_matrix->num_rows]
            = temp;

        /*
         * We want to consider the new row we just swapped into position i.
         * The following i-- will "cancel" the i++ in the for(;;) loop.
         */
        i--;
    }
}

static void compute_homology_group(
    RelationMatrix *relation_matrix,
    AbelianGroup **g,
    Boolean *overflow)
{
    int i, /* i and j are dummy variables, as usual. */
        j,
        ii, /* ii and jj are the indices of the entry which is */
        jj; /* supposed to divide all the other entries in its */
           /* row and column. */
    Boolean all_zeros,
            desired_entry_has_been_found;
    MatrixEntry **m,
                *temp,
                mult;

    /*
     * Allocate space for the AbelianGroup data structure.
     */

    *g = NEW_STRUCT(AbelianGroup);

    /*
     * Initialize (*g)->num_torsion_coefficients to zero, and
     * allocate enough space for the array of torsion coefficients.
     * We probably won't need all the space we're allocating, but
     * given that the relation_matrix is presimplified, we shouldn't
     * be overshooting by too much.
     *
     * Note that NEW_ARRAY uses my_malloc(), which will gracefully handle
     * a request for zero bytes when relation_matrix->num_columns is zero.

```

```

    */

    (*g)->num_torsion_coefficients = 0;
    (*g)->torsion_coefficients      = NEW_ARRAY(relation_matrix->num_columns, long int);

    /*
    * Let "m" (for "matrix") be a synonym for relation_matrix->relations,
    * to make the following code more legible.
    */

    m = relation_matrix->relations;

    /*
    * Note that the following code will work fine even if
    * relation_matrix->num_columns == 0. (It will keep
    * decrementing relation_matrix->num_rows until it
    * reaches zero.)
    */

    while (relation_matrix->num_rows > 0)
    {
        /*
        * If the last row contains all zeros, eliminate it.
        * Otherwise, perform matrix operations as necessary
        * to create a nonzero MatrixEntry which divides both
        * its row and its column.
        */

        all_zeros = TRUE;
        ii = relation_matrix->num_rows - 1;
        for (jj = 0; jj < relation_matrix->num_columns; jj++)
            if (m[ii][jj])
            {
                all_zeros = FALSE;
                break;
            }

        if (all_zeros)

            relation_matrix->num_rows--;

        else
        {
            /*
            * Find an entry (ii,jj) which divides every entry
            * in its row and column.
            */

            do
            {
                desired_entry_has_been_found = TRUE;

                /* Does entry (ii,jj) divide its row? */
                for (j = 0; j < relation_matrix->num_columns; j++)
                    if (m[ii][j] % m[ii][jj])
                    {
                        mult = - (m[ii][j] / m[ii][jj]);
                        add_column_multiple(relation_matrix, jj, j, mult, overflow);
                        if (*overflow == TRUE)
                            return;
                        jj = j;
                        desired_entry_has_been_found = FALSE;
                        break;
                    }

                /* Does entry (ii,jj) divide its column? */
                for (i = 0; i < relation_matrix->num_rows; i++)
                    if (m[i][jj] % m[ii][jj])
                    {
                        mult = - (m[i][jj] / m[ii][jj]);
                        add_row_multiple(relation_matrix, ii, i, mult, overflow);
                        if (*overflow == TRUE)
                            return;
                    }
            }
        }
    }

```

```

        ii = i;
        desired_entry_has_been_found = FALSE;
        break;
    }

    while ( ! desired_entry_has_been_found);

    /*
     * Use row ii to eliminate generator jj from all other rows.
     */

    for (i = 0; i < relation_matrix->num_rows; i++)
        if (i != ii)
        {
            mult = - (m[i][jj] / m[ii][jj]);
            add_row_multiple(relation_matrix, ii, i, mult, overflow);
            if (*overflow == TRUE)
                return;
        }

    /*
     * Pretend we also zeroed out the entries in row ii,
     * except for entry (ii,jj) itself. (There is no need
     * to do actually write the zeros.)
     */

    /*
     * Write the torsion coefficient iff it isn't 1,
     * and increment num_torsion_coefficients.
     */

    if (ABS(m[ii][jj]) != 1)
        (*g)->torsion_coefficients[(*g)->num_torsion_coefficients++] = ABS(m[ii]
[jj]);

    /*
     * Overwrite column jj with the last column, and decrement num_columns.
     */

    relation_matrix->num_columns--;
    for (i = 0; i < relation_matrix->num_rows; i++)
        m[i][jj] = m[i][relation_matrix->num_columns];

    /*
     * Eliminate row ii by swapping it with the last active row,
     * and decrementing num_rows.
     */
    relation_matrix->num_rows--;
    temp = m[ii];
    m[ii] = m[relation_matrix->num_rows];
    m[relation_matrix->num_rows] = temp;
}

/*
 * Now that all the relations are gone, any remaining
 * generators represent torsion coefficients of zero.
 */

while (relation_matrix->num_columns--)
    (*g)->torsion_coefficients[(*g)->num_torsion_coefficients++] = 0L;
}

static void add_row_multiple(
    RelationMatrix *relation_matrix,
    int src,
    int dst,
    MatrixEntry mult,
    Boolean *overflow)
{
    MatrixEntry factor_max,
        term0,
        term1,

```

```

        sum;
    int        j;

    /*
     * If we weren't concerned about overflows,
     * this function could be written
     */
    for (j = 0; j < relation_matrix->num_columns; j++)
        relation_matrix->relations[dst][j] += mult * relation_matrix->relations[src][j];
    /*

    /*
     * If mult == 0 there's no work to be done,
     * so return now and avoid having to worry about special cases.
     */
    if (mult == 0)
        return;

    /*
     * Let factor_max be the largest number you can multiply
     * times "mult" without getting an overflow.
     * (Division is slow compared to multiplication, but we only
     * do one division for the whole row, so it won't be noticeable
     * for large matrices.)
     */
    factor_max = ENTRY_MAX / ABS(mult);

    for (j = 0; j < relation_matrix->num_columns; j++)
    {
        if (ABS(relation_matrix->relations[src][j]) <= factor_max)
        {
            term0 = relation_matrix->relations[dst][j];
            term1 = mult * relation_matrix->relations[src][j];
            sum    = term0 + term1;

            if ( (term0 > 0 && term1 > 0 && sum < 0)
                || (term0 < 0 && term1 < 0 && (sum > 0 || sum == LONG_MIN)))
            /*
             * The addition would cause an overflow.
             * Set *overflow to TRUE and let the calling function
             * decide what to do about it.
             */
                *overflow = TRUE;
            else
                relation_matrix->relations[dst][j] = sum;
        }
        else
        {
            /*
             * The multiplication would cause an overflow.
             * Set *overflow to TRUE and let the calling function
             * decide what to do about it.
             */
            *overflow = TRUE;
        }
    }
}

static void add_column_multiple(
    RelationMatrix *relation_matrix,
    int            src,
    int            dst,
    MatrixEntry    mult,
    Boolean        *overflow)
{
    MatrixEntry factor_max,
                term0,
                term1,
                sum;
    int        i;

    /*

```

```

    * If we weren't concerned about overflows,
    * this function could be written
    *
for (i = 0; i < relation_matrix->num_rows; i++)
    relation_matrix->relations[i][dst] += mult * relation_matrix->relations[i][src];
    *
    */

/*
 * If mult == 0 there's no work to be done,
 * so return now and avoid having to worry about special cases.
 */
if (mult == 0)
    return;

/*
 * Let factor_max be the largest number you can multiply
 * times "mult" without getting an overflow.
 * (Division is slow compared to multiplication, but we only
 * do one division for the whole row, so it won't be noticeable
 * for large matrices.)
 */
factor_max = ENTRY_MAX / ABS(mult);

for (i = 0; i < relation_matrix->num_rows; i++)
{
    if (ABS(relation_matrix->relations[i][src]) <= factor_max)
    {
        term0 = relation_matrix->relations[i][dst];
        term1 = mult * relation_matrix->relations[i][src];
        sum = term0 + term1;

        if ( (term0 > 0 && term1 > 0 && sum < 0)
            || (term0 < 0 && term1 < 0 && (sum > 0 || sum == LONG_MIN)))
        /*
         * The addition would cause an overflow.
         * Set *overflow to TRUE and let the calling function
         * decide what to do about it.
         */
        *overflow = TRUE;
        else
            relation_matrix->relations[i][dst] = sum;
    }
    else
    {
        /*
         * The multiplication would cause an overflow.
         * Set *overflow to TRUE and let the calling function
         * decide what to do about it.
         */
        *overflow = TRUE;
    }
}
}

static MatrixEntry safe_addition(
    MatrixEntry a,
    MatrixEntry b,
    Boolean *overflow)
{
    /*
     * If we weren't concerned about overflows,
     * this function would be simple indeed:
     *
    return a + b;
    *
    */

    MatrixEntry sum;

    sum = a + b;

    if ( (a > 0 && b > 0 && sum < 0)

```

```
        || (a < 0 && b < 0 && (sum > 0 || sum == LONG_MIN)))
    {
        *overflow = TRUE;
        return 0;
    }
    else
        return sum;
}

static MatrixEntry safe_multiplication(
    MatrixEntry a,
    MatrixEntry b,
    Boolean      *overflow)
{
    /*
     * If we weren't concerned about overflows,
     * this function would be simple indeed:
     */
    return a * b;
    /*
     */

    MatrixEntry factor_max;

    if (a == 0)
        return 0;

    /*
     * Division is slow compared to multiplication,
     * but safe_multiplication() isn't used in any time-critical functions.
     */
    factor_max = ENTRY_MAX / ABS(a);

    if (ABS(b) <= factor_max)
        return a * b;
    else
    {
        *overflow = TRUE;
        return 0;
    }
}

static void free_relations(
    RelationMatrix *relation_matrix)
{
    int i;

    for (i = 0; i < relation_matrix->max_rows; i++)
        my_free(relation_matrix->relations[i]);

    if (relation_matrix->relations != NULL)
        my_free(relation_matrix->relations);
}
```