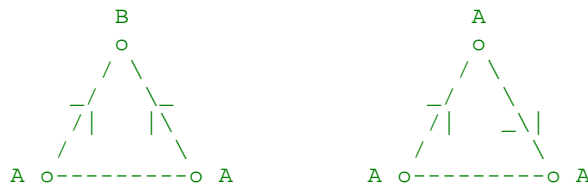


```

/* close_cusps.c
*/
/* This file contains the function
*/
/* void close_cusps(Triangulation *manifold, Boolean fill_cusp[]);
*/
/* which is used by the function fill_cusps() to permanently
/* close the indicated cusps (those for which fill_cusp[cusp->index]
/* is TRUE). The Triangulation *manifold must be a manifold with
/* finite vertices, prepared as in subdivide(). Specifically, we
/* assume that the Tetrahedra incident to the cusps form disjoint
/* regular neighborhoods of the cusps in the manifold. close_cusps()
/* removes the regular neighborhood of each cusp, and fills in the
/* hole in such a way that the given Dehn filling curve (as specified
/* by cusp->m and cusp->l) becomes a trivial curve in the new manifold.
/* I.e. it does the Dehn filling. We assume the Dehn filling coefficients
/* are relatively prime integers (fill_cusps() checks this).
*/
/*
*/
/* The remainder of this comment briefly explains the algorithm used
/* to fill the cusps. It is not a polished exposition, but I hope it
/* is mathematically clear and complete.
*/
/*
*/
/* We consider the 2-dimensional triangulation of a boundary component
/* of the manifold obtained when the regular neighborhood of a cusp
/* (as mentioned above) is stripped off. The basic idea is to fold
/* together adjacent triangles (which are really exposed faces of
/* tetrahedra) using a "close-the-book" move. Proposition 1 below
/* gives a sufficient condition that the folding does not change the
/* topology of either the manifold or its boundary. Proposition 2
/* proves that if we keep folding as long as possible (i.e. as long
/* as we can find a pair of triangles satisfying the hypothesis of
/* Proposition 1), we will end up with no more than six triangles in
/* the (2-dimensional) triangulation. Further fussing around will
/* reduce the number of triangles to two.
*/
/*
*/
/* Before stating Propositions 1 and 2, we need a definition.
*/
/* Consider a triangle, along with one of its neighbors, which may or
/* may not be distinct from the first triangle. The neighbor won't be
/* distinct iff the first triangle is glued to itself along the given edge.
/* Strictly speaking, the illustration below lies in the universal cover.
*/
/*
*/
/*      O
/*     / \
/*    /   \
/*   /       \
/*  A o         o B
/*   \       /
/*    \   /
/*     \ /
/*      O
*/
/*
*/
/* Definition. When discussing two adjacent (but not necessarily
/* distinct) triangles sharing a common edge, the vertices further
/* from the common edge are called the "opposite vertices". In the
/* above illustration, the opposite vertices are labelled A and B.
*/
/*
*/
/* Proposition A. The close-the-book move is valid if the
/* opposite vertices are distinct.
*/
/*
*/
/* Proposition B. If we apply the close-the-book move
/* until there are no more adjacent triangles with
/* distinct opposite vertices, then we will have reached
/* a triangulation with at most 6 triangles. Attaching
/* one or two tetrahedra to the boundary lets us further
/* simplify the triangulation to have exactly two triangles.
*/
/*
*/
/* First a few preparatory lemmas.
*/
/*
*/
/* Lemma 1. If the opposite vertices are distinct,
/* then the triangles are distinct too.
*/

```

* Proof. There are two ways a triangle may be glued
 * to itself (orientation preserving and orientation
 * reversing). It's trivial to check each case, and see
 * that the vertices opposite the identified edge are
 * themselves identified. [In the figure below, I
 * attempted to draw arrows showing the edge identifications.]

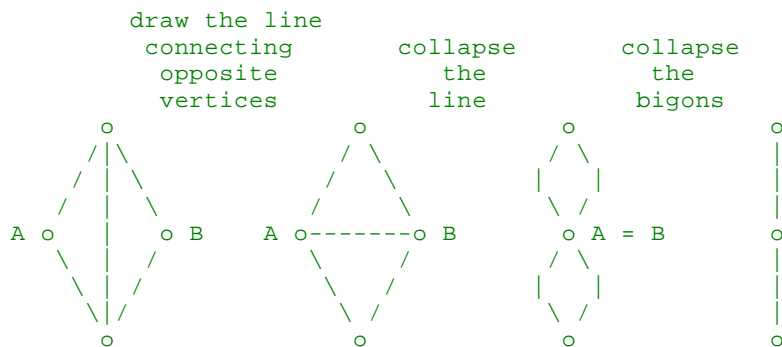


* Lemma 2. The number of vertices in a triangulation of a
 * torus or Klein bottle is half the number of triangles.

* Proof. This is an easy Euler characteristic argument.

$$\begin{aligned} \chi &= 0 = v - e + f = v - (3/2)f + f \\ \Rightarrow v &= f/2. \end{aligned}$$

* Comment: We will think of the close-the-book move in
 * purely two-dimensional terms. We think of it as a two-step
 * process. First we collapse a line segment connecting the
 * opposite vertices, then we collapse the two resulting bigons.



* Proposition A addresses the question of when these operations
 * are valid.

* Proof of Proposition A.

* By Lemma 1 the triangles are distinct, so the
 * segment connecting A to B is an embedded arc with distinct
 * endpoints, and may therefore be collapsed to a point as
 * shown in the above illustration.

* The two edges of the upper bigon (see the third frame
 * of the above illustration) are distinct, since otherwise
 * the top two edges in the second frame of the illustration
 * would be identified, and Lemma 1 would imply $A = B$.
 * Similarly, the two edges of the lower bigon are distinct.
 * It's possible that one edge of the upper bigon is identified
 * with an edge of the lower bigon, but both edges of the upper
 * bigon cannot be identified to edges of the lower, since that
 * would imply that the original triangulation contained only the
 * two triangles shown, and Lemma 2 would then imply that there
 * is only one vertex. It follows that the bigons may be collapsed.
 * Q.E.D.

* Comment. The converse to the Proposition A is almost true. If the
 * opposite vertices are not distinct ($A = B$ in the above
 * illustration) then the line connecting them is a circle.
 * If this circle is homotopically nontrivial, then it certainly
 * cannot be collapsed to a point. However, if it's homotopically
 * trivial, then a modification of the close-the-book move is still
 * possible (but is not used in SnapPea's algorithm).

* Proof of Proposition B.

```

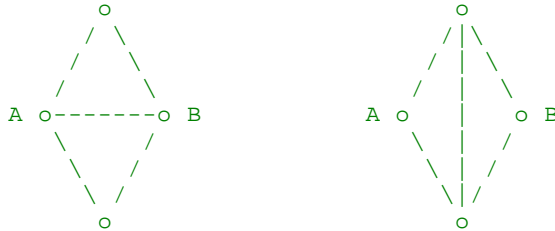
* If opposite vertices are equivalent for each pair of adjacent
* triangles, then all triangles will have the same set of vertices
* (the torus or Klein bottle is connected).
* Therefore the triangulation contains at most three vertices,
* and, by Lemma 2, at most six triangles.
* We now address the question of how to reduce the triangulation
* to only two vertices. The basic idea is to find an edge connecting
* two inequivalent vertices, and attach a (solid!) tetrahedron
* so as to implement a "two-to-two move" in the triangulation of the
* boundary.

```

```

*
*          attach a
*          tetrahedron
*          to alter the
*          triangulation
*          as shown

```



```

* Clearly an edge connecting inequivalent vertices must exist
* (by the connectivity of the torus or Klein bottle). The only
* question is whether the two incident triangles are distinct.
* If they weren't distinct, then the edge identifications would
* have to have the pattern shown on the left side of the
* illustration accompanying the proof of Lemma 1. (They couldn't
* have the pattern shown on the right side of that illustration,
* because then there wouldn't be two inequivalent vertices.)
* Vertex B in the left side of the illustration accompanying
* the proof of Lemma 1 would be isolated from the rest of the
* boundary manifold. Therefore it would be opposite a distinct
* vertex, and the close-the-book move would still be possible.
* Q.E.D.

```

```

* Technical note: close_cusps() always performs the two-to-two
* move in such a way that the subsequent close-the-book move does
* not fold together two faces of the same tetrahedron. This
* avoids needlessly creating an EdgeClass of order one in the
* manifold. Maybe such an EdgeClass would do no harm, but who knows?
*/

```

```
#include "kernel.h"
```

```
struct extra
```

```
{
    VertexIndex ideal_vertex_index;
    int         Dehn_filling_curve[4];
};
```

```

static void transfer_to_short_list(Triangulation *manifold, Boolean fill_cusp[],
    Tetrahedron *short_list_begin, Tetrahedron *short_list_end);
static Boolean incident_to_filled_cusp(Tetrahedron *tet, Boolean fill_cusp[]);
static void simplify_cusps(Triangulation *manifold, Tetrahedron *short_list_begin,
    Tetrahedron *short_list_end);
static void fold_boundary(Tetrahedron *short_list_begin, Tetrahedron *short_list_end);
static Boolean cancel_triangles(Tetrahedron *tet, FaceIndex f0);
static Boolean further_simplification(Triangulation *manifold, Tetrahedron *
    short_list_begin, Tetrahedron *short_list_end);
static Boolean two_to_two(Triangulation *manifold, Tetrahedron *tet, FaceIndex f0, Boolean
    require_distinct_edges);
static void transfer_curves(Tetrahedron *short_list_begin, Tetrahedron *short_list_end)
;
static void standard_form(Triangulation *manifold, Tetrahedron *short_list_begin,
    Tetrahedron *short_list_end);
static void standard_torus_form(Triangulation *manifold, Tetrahedron *tet);
static int max_abs_intersection_number(Tetrahedron *tet);
static void apply_two_to_two_to_eliminate(Triangulation *manifold, Tetrahedron *tet,
    int target);

```

```
static void standard_Klein_bottle_form(Triangulation *manifold, Tetrahedron *tet);
static void fold_cusps(Triangulation *manifold, Tetrahedron *short_list_begin,
    Tetrahedron *short_list_end);
static void fold_one_cusp(Triangulation *manifold, Tetrahedron *tet0);
static void replace_fake_cusps(Triangulation *manifold);
static void renumber_real_cusps(Triangulation *manifold);
```

```
void close_cusps(
Triangulation    *manifold,
Boolean          fill_cusp[])
{
Tetrahedron short_list_begin,
              short_list_end;

/*
 * Move the Tetrahedra incident to cusps-to-be-filled onto a
 * separate short list, so we don't have to be constantly sifting
 * through vast numbers of irrelevant Tetrahedra. Attach and
 * initialize an Extra structure on each Tetrahedron on the
 * short list.
 */
transfer_to_short_list(manifold, fill_cusp, &short_list_begin, &short_list_end);

/*
 * Simplify the cusps to be filled until each is triangulated
 * by at most two triangles. Ignore (1) fake "Cusps" at finite
 * vertices and (2) EdgeClasses not incident to a real Cusp;
 * we'll fix them up at the end. We maintain the tet->edge_class()
 * fields for edges incident to real Cusps so that we can tell
 * whether two EdgeClasses are distinct. The fields within an
 * EdgeClass are not maintained.
 */
simplify_cusps(manifold, &short_list_begin, &short_list_end);

/*
 * Transfer the Dehn filling curves to tet->extra->Dehn_filling_curve[].
 */
transfer_curves(&short_list_begin, &short_list_end);

/*
 * Put each boundary triangulation at each cusp to be filled into
 * the standard form:
 */


where the line of @'s is the Dehn filling curve.


standard_form(manifold, &short_list_begin, &short_list_end);

/*
 * Collapse each cusp by folding along the diagonal in
 * the above illustrations.
 */
fold_cusps(manifold, &short_list_begin, &short_list_end);

/*
 * Get rid of the old EdgeClasses and install new ones.
 */
}
```

```

    replace_edge_classes(manifold);

    /*
     * Get rid of the old fake Cusps and install new ones.
     */
    replace_fake_cusps(manifold);

    /*
     * Renumber the remaining real Cusps, so the indices
     * are contiguous.
     */
    renumber_real_cusps(manifold);

    /*
     * 96/9/28 I haven't actually observed any incorrect behavior
     * (and I don't think there is any) but I was looking through
     * the kernel code to see how orient() was being used, and I
     * got to wondering whether close_cusps() is guaranteed to preserve
     * the orientation. Just to make sure it does, call orient() now,
     * to transfer the orientation from one of the untouched tetrahedra
     * (i.e. an original tetrahedron not incident to one of the
     * cusps-to-be-filled) to all remaining tetrahedra, including the
     * new ones.
     *
     * 96/9/30 After adding the call to orient() I rechecked all
     * Chern-Simons value for the cusped census, and they are all correct.
     */
    orient(manifold);
}

static void transfer_to_short_list(
    Triangulation *manifold,
    Boolean fill_cusp[],
    Tetrahedron *short_list_begin,
    Tetrahedron *short_list_end)
{
    Tetrahedron *tet,
                *this_tet;

    /*
     * Initialize the short list.
     */

    short_list_begin->prev = NULL;
    short_list_begin->next = short_list_end;
    short_list_end->prev = short_list_begin;
    short_list_end->next = NULL;

    /*
     * Transfer Tetrahedra incident to cusps-to-be-filled
     * to the short list.
     */

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

        if (incident_to_filled_cusp(tet, fill_cusp) == TRUE)
        {
            this_tet = tet;
            tet = tet->prev; /* so the loop proceeds correctly */
            REMOVE_NODE(this_tet);
            INSERT_BEFORE(this_tet, short_list_end);
            manifold->num_tetrahedra--;
        }
}

static Boolean incident_to_filled_cusp(
    Tetrahedron *tet,
    Boolean fill_cusp[])
{
    int i;

```

```

for (i = 0; i < 4; i++)

    if (tet->cusp[i]->is_finite == FALSE
        && fill_cusp[tet->cusp[i]->index] == TRUE)
    {
        /*
         *   Make sure no other routine is using the "extra"
         *   field in the Tetrahedron data structure.
         */
        if (tet->extra != NULL)
            uFatalError("incident_to_filled_cusp", "close_cusps");

        /*
         *   Attach the locally defined struct extra.
         */
        tet->extra = NEW_STRUCT(Extra);

        /*
         *   Record for posterity the index of the ideal vertex.
         */
        tet->extra->ideal_vertex_index = i;

        return TRUE;
    }

return FALSE;
}

static void simplify_cusps(
Triangulation      *manifold,
Tetrahedron        *short_list_begin,
Tetrahedron        *short_list_end)
{
    /*
     *   simplify_cusps() simplifies a triangulation until each
     *   boundary component is triangulated by exactly two triangles.
     *
     *   It calls two other functions:
     *
     *   fold_boundary() folds together adjacent boundary
     *   triangles wherever possible, as explained in the proofs
     *   at the top of this file.  It is guaranteed to reduce the
     *   triangulation of each boundary component to at most six
     *   triangles.
     *
     *   further_simplification() look for an edge (in the 2-dimensional
     *   triangulation of the boundary) with distinct endpoints.
     *   When it finds one it does a two-to-two move, as shown in
     *   the illustration below, so that fold_boundary() can
     *   make further progress.  Because the number of vertices is
     *   exactly half the number of triangles, further_simplification()
     *   is guaranteed to make progress as long as the number of
     *   triangles in a given boundary component exceeds two.
     *
     *
     *       before                      after
     *
     *       c                          c
     *      / \                        / \
     *     /   \                      /   \
     *    /_____\                    /_____ \
     *   /          \                  |          |
     *  /              \                |          |
     * /                \               |          |
     /                  \             /           \
/a-----b            a-----b
\                  /             \           /
 \              /                 \        /
  \          /                   \       /
   \       /                     \      /
    \___/                         \___/
     c                            c
     */
    Note that this algorithm risks the creation of edges of order one
    in the (3-dimentional) triangulation of the manifold. But we
    bravely press on, confident that if we get into trouble further
    down the road, we can call a general purpose simplification
    routine to remove the offending edges.
    */

```

```

do
{
    fold_boundary(short_list_begin, short_list_end);
} while (further_simplification(manifold, short_list_begin, short_list_end) == TRUE);
}

static void fold_boundary(
    Tetrahedron *short_list_begin,
    Tetrahedron *short_list_end)
{
    Tetrahedron *tet;
    FaceIndex    f;

    /*
     * Scan down the list of boundary Tetrahedra, looking for
     * one which can be cancelled with one of its neighbors.
     * (One expects almost any pair of adjacent boundary Tetrahedra
     * to be cancellable.) When one is found, do the cancellation
     * and resume the search from the start of the list.
     */

    for (tet = short_list_begin->next;
         tet != short_list_end;
         tet = tet->next)

        for (f = 0; f < 4; f++)
        {
            if (f == tet->extra->ideal_vertex_index)
                continue;

            if (cancel_triangles(tet, f) == TRUE)
            {
                tet = short_list_begin;
                break;
            }
        }
    }

static Boolean cancel_triangles(
    Tetrahedron *tet,
    FaceIndex    f0)
{
    Tetrahedron    *nbr_tet,
                   *t,
                   *nbr_t;
    FaceIndex      f[4],
                   nbr_f[4],
                   v[4],
                   nbr_v[4];
    EdgeIndex      edge,
                   nbr_edge;
    EdgeClass      *e_class,
                   *nbr_class;
    int            i,
                   ii,
                   j;
    int            b[2],
                   c[2],
                   delta[2][2];
    PositionedTet   ptet,
                   ptet0;

    /*
     * f0 will be part of an array.
     */
    f[0] = f0;

    /*
     * Find the neighbor adjacent to face f[0].
     */
    nbr_tet      = tet->neighbor[f[0]];

```

```

nbr_f[0]      = EVALUATE(tet->gluing[f[0]], f[0]);

/*
 * Note the base of each Tetrahedron.
 */
f[1]          = tet->extra->ideal_vertex_index;
nbr_f[1]      = nbr_tet->extra->ideal_vertex_index;

/*
 * Do a quick error check.
 */
if (nbr_f[1] != EVALUATE(tet->gluing[f[0]], f[1]))
    uFatalError("cancel_triangles", "close_cusps");

/*
 * Note the EdgeIndices of the "vertical" edges farthest from
 * the common face. According to the propositions at the top
 * of this file, the Tetrahedra may be cancelled iff these
 * two edges belong to different EdgeClasses.
 */

edge          = edge_between_vertices[f[0]][f[1]];
nbr_edge      = edge_between_vertices[nbr_f[0]][nbr_f[1]];

e_class       = tet->edge_class[edge];
nbr_class     = nbr_tet->edge_class[nbr_edge];

if (e_class == nbr_class)
    return FALSE;

/*
 * According to the propositions at the top of this file,
 * the fact that the EdgeClasses are distinct implies that
 * the Tetrahedra are distinct as well. Let's check, just
 * to be sure.
 */
if (tet == nbr_tet)
    uFatalError("cancel_triangles", "close_cusps");

/*
 * The following line isn't really necessary, but it serves to avoid
 * creating EdgeClasses of order one. If further_simplification()
 * has just laid down a Tetrahedron to implement a two-to-two
 * move, we want to avoid folding that Tetrahedron onto itself.
 * By checking cases (cf. the documentation and illustrations
 * in two_to_two() below) it's easy to see that some other
 * call to cancel_triangles() (not involving gluing a Tetrahedron
 * to itself) must succeed.
 */
if (tet->neighbor[f[1]] == nbr_tet->neighbor[nbr_f[1]])
    return FALSE;

/*
 * Adjust the peripheral curves so that when we collapse
 * the Tetrahedra, the curves match up correctly.
 *
 *
 *
 *
 *
 *
 *
 *
 *

```

```

 *
 *              /\
 *             / \
 *            a   b
 *           /-----\
 *          /         \
 *         d           c
 *        /\
 *       E
 *
 * We want to insure that  $b = -c$ , which automatically imples  $a = -d$ .
 * Geometrically, all strands at  $c$  should pass to  $b$ , and all strands
 * at  $d$  should pass to  $a$ . Nothing should cut across the middle
 * from  $c$  to  $a$ , or from  $d$  to  $b$ . To accomplish this, we subtract
 * ( $b + c$ ) all the way around the (vertical) EdgeClass  $E$ .
 *
 * As long as we're travelling around the EdgeClass, set the

```



```

* tet->edge_class pointers to the address of the EdgeClass
* this one is merging into.
*/

f[2]      = remaining_face[f[1]][f[0]];
f[3]      = remaining_face[f[0]][f[1]];
nbr_f[2]   = EVALUATE(tet->gluing[f[0]], f[2]);
nbr_f[3]   = EVALUATE(tet->gluing[f[0]], f[3]);

for (i = 0; i < 2; i++)      /* which sheet */
{
    ii = (parity[tet->gluing[f[0]]] == orientation_preserving) ? i : !i;

    for (j = 0; j < 2; j++)    /* which curve */
    {
        b[j] = nbr_tet->curve[j][ii][nbr_f[1]][nbr_f[2]];
        c[j] =      tet->curve[j][i ][      f[1]][      f[2]];

        delta[j][ii] = b[j] + c[j];
    }
}

ptet0.tet      = tet;
ptet0.near_face = f[2];
ptet0.left_face = f[3];
ptet0.right_face = f[0];
ptet0.bottom_face = f[1];
ptet0.orientation = right_handed;

ptet = ptet0;
do
{
    for (i = 0; i < 2; i++)
    {
        ii = (ptet.orientation == ptet0.orientation) ? i : !i;
        for (j = 0; j < 2; j++)    /* which curve */
        {
            ptet.tet->curve[j][ii][ptet.bottom_face][ptet.left_face] += delta[j][ii];
            ptet.tet->curve[j][ii][ptet.bottom_face][ptet.near_face] -= delta[j][ii];
        }
    }
    ptet.tet->edge_class[edge_between_faces[ptet.near_face][ptet.left_face]] =
nbr_class;
    veer_left(&ptet);
} while ( ! same_positioned_tet(&ptet, &ptet0));

/*
* Imagine removing tet and nbr_tet from the manifold.
* Glues together the three exposed pairs of faces.
*
* Miraculously, this code is correct even in degenerate
* cases (corresponding to when the bigons collapse in
* the proofs at the top of this file).
*/

for (i = 1; i < 4; i++)
{
    t      = tet->neighbor[f[i]];
    nbr_t  = nbr_tet->neighbor[nbr_f[i]];

    for (j = 0; j < 4; j++)
    {
        v[j]      = EVALUATE(tet->gluing[f[i]], f[j]);
        nbr_v[j]   = EVALUATE(nbr_tet->gluing[nbr_f[i]], nbr_f[j]);
    }

    t->neighbor[v[i]]      = nbr_t;
    nbr_t->neighbor[nbr_v[i]] = t;

    t->gluing[v[i]]      = CREATE_PERMUTATION(v[0], nbr_v[0], v[1], nbr_v[1], v[2],
nbr_v[2], v[3], nbr_v[3]);
    nbr_t->gluing[nbr_v[i]] = CREATE_PERMUTATION(nbr_v[0], v[0], nbr_v[1], v[1], nbr_v[
2], v[2], nbr_v[3], v[3]);

```

```

    }

    /*
     * Free tet and nbr_tet.
     */
    REMOVE_NODE(tet);
    REMOVE_NODE(nbr_tet);
    free_tetrahedron(tet);
    free_tetrahedron(nbr_tet);

    return TRUE;
}

static Boolean further_simplification(
    Triangulation *manifold,
    Tetrahedron *short_list_begin,
    Tetrahedron *short_list_end)
{
    Tetrahedron *tet;
    FaceIndex f;

    /*
     * Scan down the list of boundary Tetrahedra, looking for
     * an edge (in the 2-dimensional triangulation of the boundary)
     * with distinct endpoints. If one is found, do the necessary
     * retriangulation (as illustrated in simplify_cusps() above)
     * and return TRUE. If none are found, return FALSE.
     *
     * Note that at most one retriangulation will be performed in
     * a single call to further_simplification().
     */

    for (tet = short_list_begin->next;
         tet != short_list_end;
         tet = tet->next)

        for (f = 0; f < 4; f++)
        {
            if (f == tet->extra->ideal_vertex_index)
                continue;

            if (two_to_two(manifold, tet, f, TRUE) == TRUE)
                return TRUE;
        }

    return FALSE;
}

/*
 * two_to_two() is called from two different parts of close_cusps().
 *
 * (1) further_simplification() calls it to alter a triangulation
 * so that the number of triangles in the triangulation of
 * a boundary component can be reduced from 4 or 6 to 2.
 * In this case, two_to_two() should do nothing and return
 * FALSE if a certain pair of EdgeClasses are not distinct.
 *
 * (2) standard_form() calls it (indirectly) to put 2-triangle
 * triangulations into the standard form. Here there is no
 * need for any EdgeClasses to be distinct (in fact, they never
 * will be).
 *
 * The argument require_distinct_edges says whether distinct EdgeClasses
 * should be required.
 */

static Boolean two_to_two(
    Triangulation *manifold,
    Tetrahedron *tet,
    FaceIndex f0,
    Boolean require_distinct_edges)
{

```

```

int      i,
        ii,
        j;
FaceIndex f[4],
        nbr_f[4];
Tetrahedron *nbr_tet,
        *new_tet,
        *tetA,
        *tetB;
EdgeClass  *e_class[4];

/*
 * Get set up as in cancel_triangles() above.
 */

f[0] = f0;

nbr_tet      = tet->neighbor[f[0]];
nbr_f[0]      = EVALUATE(tet->gluing[f[0]], f[0]);

f[1]         = tet->extra->ideal_vertex_index;
nbr_f[1]      = nbr_tet->extra->ideal_vertex_index;

if (nbr_f[1] != EVALUATE(tet->gluing[f[0]], f[1]))
    uFatalError("two_to_two", "close_cusps");

f[2]         = remaining_face[f[1]][f[0]];
f[3]         = remaining_face[f[0]][f[1]];
nbr_f[2]      = EVALUATE(tet->gluing[f[0]], f[2]);
nbr_f[3]      = EVALUATE(tet->gluing[f[0]], f[3]);

/*
 * Note the EdgeClasses.
 */
e_class[0] = tet->edge_class[ edge_between_faces[ f[2]][ f[3]] ];
e_class[1] = nbr_tet->edge_class[ edge_between_faces[nbr_f[2]][nbr_f[3]] ];
e_class[2] = tet->edge_class[ edge_between_faces[ f[0]][ f[2]] ];
e_class[3] = tet->edge_class[ edge_between_faces[ f[0]][ f[3]] ];

/*
 * If require_distinct_edges is TRUE, check the EdgeClasses.
 * (See comment preceeding this function.)
 * further_simplification() can make progress iff e_class[2] != e_class[3].
 */
if (require_distinct_edges == TRUE
    && e_class[2] == e_class[3])

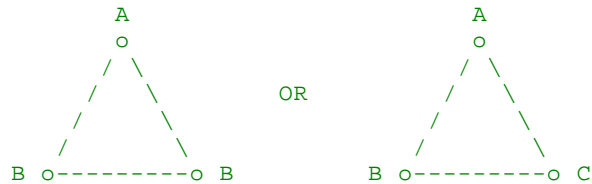
    return FALSE;

/*
 * In further_simplification() . . .
 *
 * We now know that no triangle in the
 * 2-dimensional triangulation of the boundary is
 * glued to itself, because if it were glued to itself with
 *
 * an orientation preserving gluing, then there would
 * be an isolated vertex, and further progress would
 * have been possible in fold_boundary()
 *
 * an orientation reversing gluing, then there would be
 * only one vertex in the triangulation of the boundary,
 * and class2 would have equalled class3. (Recall
 * that when fold_boundary() can make no more
 * progress, all boundary triangles have the same set
 * of vertices, including multiplicity.)
 *
 * In standard_form() . . .
 *
 * We know that no triangle can be glued to itself, because
 * otherwise the boundary would be a Klein bottle already in
 * standard form.
 */
if (tet == nbr_tet)

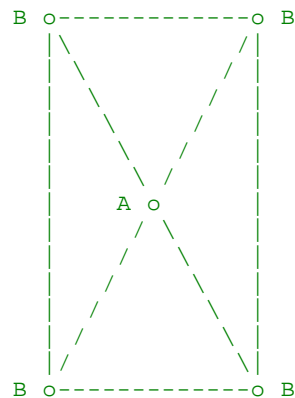
```

```
uFatalError("two_to_two", "close_cusps");
```

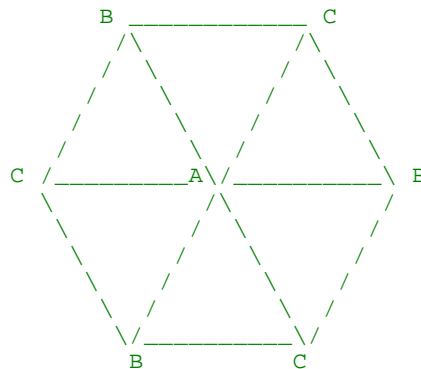
```
/*
 * [This comment applies on if we were called from
 * further_simplification().]
 *
 * Continuing with the idea that all boundary triangles
 * have the same vertex set, it follows that each triangle
 * must be of the form
```



```
/*
 * In the first case, the boundary component is formed
 * by identifying sides of the square
```



```
/*
 * and in the second case by identifying sides of the
 * hexagon
```



```
/*
 * It's easy to figure out that there are three possible
 * gluing patterns for the square (xyXY, xyXy and xxyy)
 * and two for the hexagon (xyzXYZ and xyzXzy). In all
 * but one case (the xxyy gluing of the square) we may
 * conclude that tet and nbr_tet's neighbors are distinct
 * from tet and nbr_tet (except for obvious place they meet).
 * Even in the exceptional case, there are other places
 * where we could do the two-to-two move where the
 * neighbors are distinct from tet and nbr_tet. So
 * if the neighbors aren't distinct from tet and nbr_tet,
 * we simply return FALSE and wait for one of those
 * more convenient places to show up.
 */
```

```
/*
 * First label everything in sight.
 * Use macros rather than writing quantities into
 * arrays, so that in cases where tet and nbr_tet are glued
 * nontrivially to each other, the results will be correct.
```

```

    * That is, we want the quantities to be reevaluated every
    * time they are used.
    */

```

```

#define TET_T(i)      tet->neighbor[f[i]]
#define NBR_T(i)      nbr_tet->neighbor[nbr_f[i]]
#define TET_V(i,j)    EVALUATE(tet->gluing[f[i]], f[j])
#define NBR_V(i,j)    EVALUATE(nbr_tet->gluing[nbr_f[i]], nbr_f[j])

```

```

/*
 * In the event we were called from further_simplification,
 * check that the TET_T(i) and NBR_T(i) are distinct from
 * tet and nbr_tet. This avoids unnecessarily creating an
 * EdgeClass of order one (maybe it doesn't matter, but
 * why risk it?).
 */

```

```

if (require_distinct_edges == TRUE)

```

```

    for (i = 2; i < 4; i++)

```

```

        if (TET_T(i) == tet
            || TET_T(i) == nbr_tet
            || NBR_T(i) == tet
            || NBR_T(i) == nbr_tet)

```

```

            return FALSE;

```

```

/*
 * We introduce a new Tetrahedron which realizes the
 * two-to-two move on the boundary triangulation, and
 * adjust tet and nbr_tet to sit correctly above it.
 *
 * We could just reuse the tet and nbr_tet structures, but
 * then there'd be problems if either is glued to itself or
 * the other (aside from the obvious place they're glued).
 * So we replace tet and nbr_tet with tetA and tetB
 * respectively.
 *
 * You might want to draw a picture to keep track of what's
 * going on.
 *
 * Vertex 0 of the new_tet sits at vertex (not face)      f[0] of      tet.
 * Vertex 1 of the new_tet sits at vertex (not face) nbr_f[0] of nbr_tet.
 * Vertex 2 of the new_tet sits at vertex (not face)      f[3] of      tet.
 * Vertex 3 of the new_tet sits at vertex (not face)      f[2] of      tet.
 *
 * Vertex 0 of tetA will be over vertex 0 of new_tet.
 * Vertex 1 of tetA will be at the cusp.
 * Vertex 2 of tetA will be over vertex 1 of new_tet.
 * Vertex 3 of tetA will be over vertex 2 of new_tet.
 *
 * Vertex 0 of tetB will be over vertex 1 of new_tet.
 * Vertex 1 of tetB will be at the cusp.
 * Vertex 2 of tetB will be over vertex 3 of new_tet.
 * Vertex 3 of tetB will be over vertex 0 of new_tet.
 *
 * Take a deep breath and set all the necessary fields . . .
 */

```

```

new_tet = NEW_STRUCT(Tetrahedron);
tetA    = NEW_STRUCT(Tetrahedron);
tetB    = NEW_STRUCT(Tetrahedron);

```

```

initialize_tetrahedron(new_tet);
initialize_tetrahedron(tetA);
initialize_tetrahedron(tetB);

```

```

new_tet->cusp[0] =      tet->cusp[  f[0]];
new_tet->cusp[1] = nbr_tet->cusp[nbr_f[0]];
new_tet->cusp[2] =      tet->cusp[  f[3]];
new_tet->cusp[3] =      tet->cusp[  f[2]];

```

```

new_tet->neighbor[0] = NBR_T(1);

```

```

new_tet->neighbor[1] = TET_T(1);
new_tet->gluing[0] = CREATE_PERMUTATION(
    0, NBR_V(1, 1),
    1, NBR_V(1, 0),
    2, NBR_V(1, 3),
    3, NBR_V(1, 2));
new_tet->gluing[1] = CREATE_PERMUTATION(
    0, TET_V(1, 0),
    1, TET_V(1, 1),
    2, TET_V(1, 3),
    3, TET_V(1, 2));

NBR_T(1)->neighbor[NBR_V(1, 1)] = new_tet;
TET_T(1)->neighbor[TET_V(1, 1)] = new_tet;
NBR_T(1)->gluing[NBR_V(1, 1)] = inverse_permutation[new_tet->gluing[0]];
TET_T(1)->gluing[TET_V(1, 1)] = inverse_permutation[new_tet->gluing[1]];

new_tet->neighbor[2] = tetB;
new_tet->neighbor[3] = tetA;
new_tet->gluing[2] = CREATE_PERMUTATION(0, 3, 1, 0, 2, 1, 3, 2);
new_tet->gluing[3] = CREATE_PERMUTATION(0, 0, 1, 2, 2, 3, 3, 1);

tetA->neighbor[1] = new_tet;
tetB->neighbor[1] = new_tet;
tetA->gluing[1] = inverse_permutation[new_tet->gluing[3]];
tetB->gluing[1] = inverse_permutation[new_tet->gluing[2]];

tetA->neighbor[2] = TET_T(2);
tetA->gluing[2] = CREATE_PERMUTATION(0, TET_V(2, 0), 1, TET_V(2, 1), 2, TET_V(2, 2)
, 3, TET_V(2, 3));
TET_T(2)->neighbor[TET_V(2, 2)] = tetA;
TET_T(2)->gluing[TET_V(2, 2)] = inverse_permutation[tetA->gluing[2]];

tetA->neighbor[0] = NBR_T(2);
tetA->gluing[0] = CREATE_PERMUTATION(0, NBR_V(2, 2), 1, NBR_V(2, 1), 2, NBR_V(2, 0)
, 3, NBR_V(2, 3));
NBR_T(2)->neighbor[NBR_V(2, 2)] = tetA;
NBR_T(2)->gluing[NBR_V(2, 2)] = inverse_permutation[tetA->gluing[0]];

tetB->neighbor[0] = TET_T(3);
tetB->gluing[0] = CREATE_PERMUTATION(0, TET_V(3, 3), 1, TET_V(3, 1), 2, TET_V(3, 2)
, 3, TET_V(3, 0));
TET_T(3)->neighbor[TET_V(3, 3)] = tetB;
TET_T(3)->gluing[TET_V(3, 3)] = inverse_permutation[tetB->gluing[0]];

tetB->neighbor[3] = NBR_T(3);
tetB->gluing[3] = CREATE_PERMUTATION(0, NBR_V(3, 0), 1, NBR_V(3, 1), 2, NBR_V(3, 2)
, 3, NBR_V(3, 3));
NBR_T(3)->neighbor[NBR_V(3, 3)] = tetB;
NBR_T(3)->gluing[NBR_V(3, 3)] = inverse_permutation[tetB->gluing[3]];

tetA->neighbor[3] = tetB;
tetB->neighbor[2] = tetA;
tetA->gluing[3] = CREATE_PERMUTATION(0, 3, 1, 1, 2, 0, 3, 2);
tetB->gluing[2] = inverse_permutation[tetA->gluing[3]];

for (j = 0; j < 2; j++) /* which curve */
    for (i = 0; i < 2; i++) /* which sheet */
    {
        ii = (parity[CREATE_PERMUTATION(0, f[0], 1, f[1], 2, f[2], 3, f[3])] == 0) ? i
: !i;
        tetA->curve[j][i][1][2] = tet->curve[j][ii][f[1]][f[2]];

        ii = (parity[CREATE_PERMUTATION(0, nbr_f[2], 1, nbr_f[1], 2, nbr_f[0], 3, nbr_f[
3])] == 0) ? i : !i;
        tetA->curve[j][i][1][0] = nbr_tet->curve[j][ii][nbr_f[1]][nbr_f[2]];

        tetA->curve[j][i][1][3] = - (tetA->curve[j][i][1][2] + tetA->curve[j][i][1][0])
;

        ii = (parity[CREATE_PERMUTATION(0, f[3], 1, f[1], 2, f[2], 3, f[0])] == 0) ? i
: !i;
        tetB->curve[j][i][1][0] = tet->curve[j][ii][f[1]][f[3]];
    }

```

```

    ii = (parity[CREATE_PERMUTATION(0, nbr_f[0], 1, nbr_f[1], 2, nbr_f[2], 3, nbr_f[3])
[3]]) == 0) ? i : !i;
    tetB->curve[j][i][1][3] = nbr_tet->curve[j][ii][nbr_f[1]][nbr_f[3]];

    tetB->curve[j][i][1][2] = - (tetB->curve[j][i][1][0] + tetB->curve[j][i][1][3]);
;
}

tetA->edge_class[edge_between_faces[2][3]] = e_class[0];
tetA->edge_class[edge_between_faces[3][0]] = e_class[1];
tetA->edge_class[edge_between_faces[0][2]] = e_class[2];
tetB->edge_class[edge_between_faces[0][2]] = e_class[0];
tetB->edge_class[edge_between_faces[2][3]] = e_class[1];
tetB->edge_class[edge_between_faces[3][0]] = e_class[3];

tetA->cusp[1] = tet->cusp[f[1]];
tetB->cusp[1] = tet->cusp[f[1]];

tetA->cusp[0] = new_tet->cusp[0];
tetA->cusp[2] = new_tet->cusp[1];
tetA->cusp[3] = new_tet->cusp[2];
tetB->cusp[0] = new_tet->cusp[1];
tetB->cusp[2] = new_tet->cusp[3];
tetB->cusp[3] = new_tet->cusp[0];

tetA->extra = NEW_STRUCT(Extra);
tetB->extra = NEW_STRUCT(Extra);

tetA->extra->ideal_vertex_index = 1;
tetB->extra->ideal_vertex_index = 1;

if (require_distinct_edges == FALSE)
{
    tetA->extra->Dehn_filling_curve[2] = tet->extra->Dehn_filling_curve[f[2]];
    tetB->extra->Dehn_filling_curve[0] = tet->extra->Dehn_filling_curve[f[3]];
    tetA->extra->Dehn_filling_curve[0] = nbr_tet->extra->Dehn_filling_curve[nbr_f[2]];
    tetB->extra->Dehn_filling_curve[3] = nbr_tet->extra->Dehn_filling_curve[nbr_f[3]];
    tetA->extra->Dehn_filling_curve[3] = - (tetA->extra->Dehn_filling_curve[2] + tetA->
extra->Dehn_filling_curve[0]);
    tetB->extra->Dehn_filling_curve[2] = - (tetB->extra->Dehn_filling_curve[0] + tetB->
extra->Dehn_filling_curve[3]);
}

INSERT_BEFORE(new_tet, &manifold->tet_list_end);

/*
 * To avoid screwing up linked lists of Tetrahedra
 * (i.e. the short list, which standard_form() traverses
 * in a for(;;) loop), we copy tetA and tetB onto the
 * storage formerly used by tet and nbr_tet.
 * In spirit they are new Tetrahedra, but we want
 * them to occupy the same physical memory as the old
 * Tetrahedra, so as not to screw up a higher level
 * function which holds a pointer to one of the old
 * Tetrahedra.
 */

tetA->prev = tet->prev;
tetB->prev = nbr_tet->prev;
tetA->next = tet->next;
tetB->next = nbr_tet->next;

my_free(tet->extra);
my_free(nbr_tet->extra);

*tet = *tetA;
*nbr_tet = *tetB;

/*
 * If they are glued to themselves, correct the
 * neighbor fields.
 */

for (i = 0; i < 4; i++)

```

```

{
    if (tet->neighbor[i] == tetA)
        tet->neighbor[i] = tet;
    if (tet->neighbor[i] == tetB)
        tet->neighbor[i] = nbr_tet;
    if (nbr_tet->neighbor[i] == tetA)
        nbr_tet->neighbor[i] = tet;
    if (nbr_tet->neighbor[i] == tetB)
        nbr_tet->neighbor[i] = nbr_tet;
}

/*
 * Correct the neighbor fields for remaining neighbors.
 */

for (i = 0; i < 4; i++)
{
    tet->neighbor[i]->neighbor[EVALUATE(tet->gluing[i],i)] = tet;
    nbr_tet->neighbor[i]->neighbor[EVALUATE(nbr_tet->gluing[i],i)] = nbr_tet;
}

my_free(tetA);
my_free(tetB);

manifold->num_tetrahedra++;

return TRUE;
}

static void transfer_curves(
    Tetrahedron *short_list_begin,
    Tetrahedron *short_list_end)
{
    Tetrahedron *tet;
    VertexIndex v;
    Cusp *cusp;
    int i,
        j;

    /*
     * Transfer the Dehn filling curves to tet->extra->Dehn_filling_curve[].
     */

    for (tet = short_list_begin->next;
         tet != short_list_end;
         tet = tet->next)
    {
        v = tet->extra->ideal_vertex_index;
        cusp = tet->cusp[v];

        for (i = 0; i < 4; i++)
        {
            if (i == v)
                continue;

            tet->extra->Dehn_filling_curve[i] = 0;

            for (j = 0; j < 2; j++)
                tet->extra->Dehn_filling_curve[i]
                    += (int)cusp->m * tet->curve[M][j][v][i]
                    + (int)cusp->l * tet->curve[L][j][v][i];
        }
    }
}

static void standard_form(
    Triangulation *manifold,
    Tetrahedron *short_list_begin,
    Tetrahedron *short_list_end)
{
    Tetrahedron *tet;

```



```

/*
 * See the documentation in close_cusps() for an illustration of
 * the standard forms.
 */

for (tet = short_list_begin->next;
     tet != short_list_end;
     tet = tet->next)

    if (tet->cusps[tet->extra->ideal_vertex_index]->topology == torus_cusp)
        standard_torus_form(manifold, tet);
    else
        standard_Klein_bottle_form(manifold, tet);
}

static void standard_torus_form(
    Triangulation *manifold,
    Tetrahedron *tet)
{
    int max;

    /*
     * The idea here is to modify the triangulation of the
     * boundary torus so that the Dehn filling curve
     * looks simpler. Geometrically, we are going to do Dehn
     * twists which realize the Euclidean algorithm, but you
     * needn't think in terms of Dehn twists as you read the
     * following code.
     *
     * The Dehn filling curve will intersect the sides of a
     * boundary triangle with intersection numbers a, b and c,
     * where  $a + b + c = 0$ . If, say, c has the greatest absolute
     * value, then a and b will have the same sign, and  $c = -(a + b)$ .
     * The intersection numbers on the other triangle in the triangulation
     * are of course the negatives of these.
     *
     * If we do a two-to-two move across the edge with intersection
     * number c, then the new intersection numbers will be a, -b and
     * (b - a). Each time we do this we reduce the absolute value of
     * the largest intersection number, until we reach a state where
     * one of the intersection numbers is zero and the other two are
     * negatives of each other. The latter two must be +1 and -1,
     * because the Dehn filling curve is a simple closed curve.
     * Thus, we eventually reach a state where the intersection
     * numbers are {0, +1, -1}.
     *
     * The state just before this (if any) must have been {1, 1, 2}.
     * {1, 1, 2} is the standard form.
     *
     * So . . . the algorithm is
     *
     * if (state = {0, +1, -1})
     *     back up to {1, 1, 2}
     * else
     *     while (state is not {1, 1, 2})
     *         apply a two-to-two move to reduce the absolute value
     *         of the largest intersection number
     *
     * Technical comment: in the case where we have to back up
     * to {1, 1, 2}, when we seal the cusp we'll be creating an
     * EdgeClass of order 1 in the 3-manifold.
     */

    max = max_abs_intersection_number(tet);

    if (max == 1)
        apply_two_to_two_to_eliminate(manifold, tet, 0);
    else
        while (max > 2)
        {
            apply_two_to_two_to_eliminate(manifold, tet, max);
            max = max_abs_intersection_number(tet);
        }
}

```

```

}

static int max_abs_intersection_number(
    Tetrahedron *tet)
{
    VertexIndex v;
    int max;
    int i;

    v = tet->extra->ideal_vertex_index;

    max = 0;

    for (i = 0; i < 4; i++)
    {
        if (i == v)
            continue;

        if (ABS(tet->extra->Dehn_filling_curve[i]) > max)
            max = ABS(tet->extra->Dehn_filling_curve[i]);
    }

    return max;
}

static void apply_two_to_two_to_eliminate(
    Triangulation *manifold,
    Tetrahedron *tet,
    int target)
{
    VertexIndex v;
    FaceIndex f;

    /*
     * apply_two_to_two_to_eliminate() applies a two-to-two move
     * to alter the boundary triangulation in such a way as to
     * eliminate the edge whose intersection number with the
     * Dehn filling curve has absolute value target.
     */

    v = tet->extra->ideal_vertex_index;

    /*
     * Find the FaceIndex f of the face of tet corresponding to the
     * 2-d edge we want to eliminate.
     */

    for (f = 0; f < 4; f++)
    {
        if (f == v)
            continue;

        if (ABS(tet->extra->Dehn_filling_curve[f]) == target)
            break;
    }

    if (f == 4) /* didn't find the right f */
        uFatalError("apply_two_to_two_to_eliminate", "close_cusps");

    (void) two_to_two(manifold, tet, f, FALSE);
}

static void standard_Klein_bottle_form(
    Triangulation *manifold,
    Tetrahedron *tet)
{
    VertexIndex v;
    FaceIndex f;

    v = tet->extra->ideal_vertex_index;

```

```

/*
 * Consider the triangle corresponding to tet in the
 * (2-dimensional) triangulation of the boundary Klein bottle.
 * The triangulation of the Klein bottle is in the standard
 * form iff this triangle has two sides glued to each other.
 */

for (f = 0; f < 4; f++)
{
    if (f == v)
        continue;

    if (tet->neighbor[f] == tet)
        return;
}

/*
 * The boundary triangulation must have one of the following
 * two forms, where the line of @'s is the meridian.
 * (In fact the two forms represent the same triangulation.
 * I've drawn them separately to convince the reader--and
 * myself--that this is the only triangulation other than
 * than the standard one.)
 */
      *
      *          ----->>-----
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^@@@@@/^@@@@@^
      *          |         \      |
      *          |        \       |
      *          |       \        |
      *          |      \         |
      *          |     \          |
      *          |    \           |
      *          |   \            |
      *          |  \             |
      *          | \              |
      *          |  \             |
      *          |   \            |
      *          |    \           |
      *          |     \          |
      *          |      \         |
      *          |       \        |
      *          |        \       |
      *          |         \      |
      *          |          \     |
      *          |           \    |
      *          |            \   |
      *          |             \  |
      *          |              \|
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
      *          |          /     |
      *          |         /      |
      *          |        /       |
      *          |       /        |
      *          |      /         |
      *          |     /          |
      *          |    /           |
      *          |   /            |
      *          |  /             |
      *          | /              |
      *          |/               |
      *          ^               |
      *          |               /|
      *          |              / |
      *          |             /  |
      *          |            /   |
      *          |           /    |
     
```

```

int      i,
        j;
int      abs_int_num;
Cusp     *dead_cusp;

/*
 * The illustrations in close_cusps() show the standard forms
 * for torus and Klein bottle cusps. We are going to fold
 * along the diagonal. Note that the absolute value of the
 * intersection number of the Dehn filling curve with the edge
 * we're folding along is 2 for a torus and 0 for a Klein bottle.
 * For all other edges it's 1.
 */

/*
 * f[0][0] will be the FaceIndex of the bottom face of tet[0] (the
 * one furthest from the ideal vertex). f[0][1] will be the face
 * incident to the edge we're folding along. f[0][2] and f[0][3]
 * will be the remaining faces. f[1][0-3] will be the corresponding
 * faces of tet[1], the other Tetrahedron at this cusp.
 */

tet[0] = tet0;

f[0][0] = tet[0]->extra->ideal_vertex_index;

for (f[0][1] = 0; f[0][1] < 4; f[0][1]++)
{
    if (f[0][1] == f[0][0])
        continue;

    abs_int_num = ABS(tet[0]->extra->Dehn_filling_curve[f[0][1]]);

    if (abs_int_num == 2 || abs_int_num == 0)
        break;
}

if (f[0][1] == 4)
    uFatalError("fold_one_cusp", "close_cusps");

f[0][2] = remaining_face[f[0][0]][f[0][1]];
f[0][3] = remaining_face[f[0][1]][f[0][0]];

tet[1] = tet[0]->neighbor[f[0][1]];

for (i = 0; i < 4; i++)
    f[1][i] = EVALUATE(tet[0]->gluing[f[0][1]], f[0][i]);

/*
 * nbr[0] (resp. nbr[1]) is the Tetrahedron (with all finite vertices)
 * which sits underneath tet[0] (resp. tet[1]). Their FaceIndices
 * are nf[0][0] and nf[1][0], and are indexed in the natural way
 * relative to tet[0] and tet[1].
 */

for (i = 0; i < 2; i++)
{
    nbr[i] = tet[i]->neighbor[f[i][0]];

    for (j = 0; j < 4; j++)
        nf[i][j] = EVALUATE(tet[i]->gluing[f[i][0]], f[i][j]);
}

/*
 * To fold the cusp, we simply identify similarly indexed
 * vertices of nbr[0] and nbr[1].
 */

for (i = 0; i < 2; i++)
{
    nbr[i]->neighbor[nf[i][0]] = nbr[!i];

    nbr[i]->gluing[nf[i][0]] = CREATE_PERMUTATION(
        nf[i][0], nf[!i][0],

```

```

        nf[i][1], nf[!i][1],
        nf[i][2], nf[!i][2],
        nf[i][3], nf[!i][3]);
    }

    /*
     * Discard tet[0] and tet[1].
     */

    dead_cusp = tet[0]->cusp[f[0][0]];
    if (dead_cusp->topology == torus_cusp)
        manifold->num_or_cusps--;
    else
        manifold->num_nonor_cusps--;
    manifold->num_cusps--;
    REMOVE_NODE(dead_cusp);
    my_free(dead_cusp);

    for (i = 0; i < 2; i++)
    {
        REMOVE_NODE(tet[i]);
        free_tetrahedron(tet[i]);
    }
}

static void replace_fake_cusps(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i;
    Cusp *cusp,
        *dead_cusp;

    /*
     * Set to NULL all tet->cusp pointers which point to fake Cusps.
     */

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

        for (i = 0; i < 4; i++)

            if (tet->cusp[i]->is_finite == TRUE)

                tet->cusp[i] = NULL;

    /*
     * Free the fake Cusps.
     */

    for (cusp = manifold->cusp_list_begin.next;
        cusp != &manifold->cusp_list_end;
        cusp = cusp->next)

        if (cusp->is_finite == TRUE)
        {
            dead_cusp = cusp;
            cusp = cusp->prev; /* so the loop will proceed correctly */
            REMOVE_NODE(dead_cusp);
            my_free(dead_cusp);
        }

    /*
     * Assign new fake Cusps.
     */

    create_fake_cusps(manifold);
}

static void renumber_real_cusps(
    Triangulation *manifold)

```

```
{
    Cusp      *cusp;
    int        cusp_count;

    cusp_count = 0;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)

        if (cusp->is_finite == FALSE)

            cusp->index = cusp_count++;
}
```