

```

/*
 * simplify_triangulation.c
 *
 * This file contains the following low-level routines for locally
 * modifying a triangulation
 *
 *      FuncResult  cancel_tetrahedra(EdgeClass *edge, EdgeClass **where_to_resume, int *
num_tetrahedra_ptr);
 *      FuncResult  three_to_two(EdgeClass *edge, EdgeClass **where_to_resume, int *
num_tetrahedra_ptr);
 *      FuncResult  two_to_three(Tetrahedron *tet0, FaceIndex f, int *num_tetrahedra_ptr);
 *
 * as well as the following high-level routines which call them:
 *
 *      void        basic_simplification(Triangulation *manifold);
 *      void        randomize_triangulation(Triangulation *manifold);
 *
 * It also includes the low-level routine
 *
 *      void        one_to_four(Tetrahedron *tet, int *num_tetrahedra_ptr, int new_cusp_index);
 *
 * which is not used by basic_simplification() or randomize_triangulation(),
 * but is called from canonize_part_2.c.
 *
 * The low-level routines are as follows
 *
 *      cancel_tetrahedra() cancels two Tetrahedra which share
 *      a common edge of order 2.
 *
 *      three_to_two() replaces three Tetrahedra surrounding a common
 *      edge with two Tetrahedra sharing a common face.
 *
 *      two_to_three() replaces two Tetrahedra sharing a common face
 *      with three Tetrahedra surrounding a common edge.
 *
 *      one_to_four() replaces one Tetrahedron with four Tetrahedra
 *      meeting at a point.
 *
 * If an operation cannot be performed because of a topological or
 * geometric obstruction, the function does nothing and returns func_failed.
 * Otherwise, it performs the operation and returns func_OK.
 *
 * The function one_to_four() will always succeed, and therefore returns
 * void. It introduces a finite vertex at the center of the Tetrahedron,
 * and therefore cannot be used when a hyperbolic structure is present.
 *
 * The three_to_two(), two_to_three() and one_to_four() operations each
 * correspond to a projection of a 4-simplex.
 *
 * For further details, please see the comments preceding each low-level
 * function.
 *
 * In practice, other SnapPea routines will most likely call the
 * high-level functions basic_simplification() and randomize_triangulation().
 *
 * basic_simplification() first does easy simplifications
 * (namely retriangulating neighborhoods of EdgeClasses of
 * order 1, 2 and 3 to reduce the number of Tetrahedra whenever
 * possible, and retriangulating suspended pentagons using five
 * Tetrahedra instead of six), and then retriangulates octahedra
 * (choosing a different one of the three possible axes for the
 * subdivision into four Tetrahedra) in hopes of making further
 * easy simplifications possible.
 *
 * randomize_triangulation() randomizes the Triangulation, and then
 * resimplifies it.
 *
 * basic_simplification() and randomize_triangulation() may be called
 * for manifolds with or without a hyperbolic structure present.
 * The final Triangulation may depend on whether or not the hyperbolic
 * structure is present, because when a hyperbolic structure is present
 * the low-level routines will refuse to create degenerate Tetrahedra.
 */

```

```

* Most routines in SnapPea keep track of edge angles "mod 0" rather
* than just "mod 2 pi", so that, e.g., a ComplexWithLog with
* log.imag equal to  $(3/2)\pi$  is different than one with log.imag
* equal to  $(-1/2)\pi$ . Unfortunately, the mod 0 angles for a given
* Triangulation are somewhat arbitrary, in the sense that the following
* procedure converts one mod 0 solution to a different mod 0 solution.
*
* Pick an EdgeClass ec in the Triangulation, and consider all the
* Tetrahedra incident to it. If the incident edges don't all belong
* to distinct Tetrahedra, work in the universal cover, so that the
* Tetrahedra will at least appear distinct. For each Tetrahedron,
* call the angle incident to the EdgeClass ec gamma, and call the
* opposite angle gamma as well (they will of course be equal, due to
* the symmetry of the ideal tetrahedron). Call one remaining pair
* of opposite edges alpha, and the other pair beta. Make the choice
* of alphas and betas consistent for all the Tetrahedra incident to
* the EdgeClass ec; that is, each alpha of one Tetrahedron should
* be incident to a beta of an adjacent Tetrahedron. Now add  $2\pi i$ 
* to the log of each alpha edge angle, and subtract  $2\pi i$  from the
* log of each beta edge angle. Note that
*
* (1) The sum of the logs of the edge angles remains  $\pi i$  for
*     each Tetrahedron.
*
* (2) The sum of the logs of the angles surrounding each EdgeClass
*     remains  $2\pi i$ .
*
* (3) The holonomies of the cusps are unaltered. (At least in the
*     generic case -- I haven't thought through what happens when
*     the Tetrahedra incident to the EdgeClass ec are not all distinct.)
*
* The point of all this is that the mod 0 edge angles in a Triangulation
* are not uniquely defined. If all the Tetrahedra are positively
* oriented, then one typically expects to find a solution with mod 0
* edge angles in the range  $[0, \pi]$ , but if some of the Tetrahedra
* are negatively oriented, then the choice of edge angles becomes
* murkier.
*
* When I first started writing the low-level routines in this file
* (i.e. two_to_three(), three_to_two() and cancel_tetrahedra())
* I naively expected to keep track of the mod 0 edge angles. This
* was no problem in the three_to_two() move. It was a little more
* difficult in the two_to_three() move because some arbitrary choices
* were involved, and I couldn't see how to prove that the angles sums
* would be preserved both at EdgeClasses and in each Tetrahedron.
* The scheme broke down entirely in cancel_tetrahedra(), because
* a pair of allegedly cancelling angles could differ by  $2\pi i$ .
* One could correct the problem locally, but only at the risk of
* creating a solution whose edge angles differed from the "preferred"
* ones by multiples of  $2\pi i$ , as described above. Given that
*
* (1) I couldn't see how to simplify the mod 0 angles in any
*     reasonable and canonical way, and
*
* (2) We are mainly interested in solutions with positively
*     oriented Tetrahedra, or at worst with angles in the
*     range  $[(-1/2)\pi, (3/2)\pi]$ ,
*
* I decided that the low-level routines in this file should only
* keep track of the mod  $2\pi$  angles, choosing values in the
* range  $[(-1/2)\pi, (3/2)\pi]$ . Just before returning,
* basic_simplification() calls polish_solution() (that's polish,
* not Polish). In the generic case (when the mod 0 angles are
* valid, but the TetShapes have lost some accuracy) the effect of
* polish_solution() is to recover the lost accuracy, without
* substantially changing the solution. In the exceptional case
* that the edge angles don't add up correctly around a Tetrahedron
* or EdgeClass, polish_solution() will find an entirely new
* solution to the gluing equations.
*
* randomize_triangulation() calls basic_simplification(), so its
* solutions also get polished.
*
* 97/2/3 Modified to strip off the geometric structure (if any)

```

```

* at the start of basic_simplification() and randomize_triangulation(),
* and (if there was a geometric structure) recompute it at the end.
* The old system was working fine for hyperbolic manifolds, but now that
* SnapPea is working with degenerate solutions (to split along normal
* surfaces) one wants to be able to randomize and simplify them too.
* 98/5/20 Modified *not* to strip off the geometric structure
* when the cusp_nbhd_position is present. The low-level routines
* need the hyperbolic structure to maintain the cusp_nbhd_position.
*
* 97/2/4 Modified to handle Triangulations containing finite vertices.
* The 4-1 move, in which four tetrahedra surrounding a common finite
* vertex are replaced by a single tetrahedron, is handled implicitly
* as a 3-2 move (on one of the edge classes incident to the finite
* vertex) followed by a 2-0 move on a pair of tetrahedra having three
* faces and a finite vertex in common. The code in cancel_tetrahedra()
* was modified to accomodate this. When the finite vertex is removed,
* a gap remains in the (negative) numbering of the Cusps structures
* for finite vertices, but this isn't a problem.
*/

#include "kernel.h"
#include <stdlib.h>      /* needed for rand() */

/*
* ORDER_FOUR_ITERATIONS_IN_SIMPLIFY tells how many times
* basic_simplification() should pass unsuccessfully down
* the list of EdgeClasses before giving up.
*/

#define ORDER_FOUR_ITERATIONS_IN_SIMPLIFY    6

/*
* RANDOMIZATION_MULTIPLE tells how long randomize_triangulation()
* should keep randomizing before it resimplifies the manifold.
* It will attempt RANDOMIZATION_MULTIPLE * manifold->num_tetrahedra
* two-to-three moves, each followed by some rudimentary resimplification
* to avoid wasting time in degenerate situations.
*/

#define RANDOMIZATION_MULTIPLE                4

static Tetrahedron *get_tet(Triangulation *manifold, int desired_index);
static void check_for_cancellation(Triangulation *manifold);
static Boolean easy_simplification(Triangulation *manifold);
static FuncResult remove_edge_of_order_one(EdgeClass *edge, EdgeClass **where_to_resume,
int *num_tetrahedra_ptr);
static Boolean this_way_works(Tetrahedron *tet, FaceIndex left_face, FaceIndex
right_face, FaceIndex bottom_face);
static FuncResult cancel_tetrahedra_with_finite_vertex(Tetrahedron *tet, VertexIndex
finite_vertex, EdgeClass *edge, EdgeClass **where_to_resume, int *num_tetrahedra_ptr);
static FuncResult edges_of_order_four(EdgeClass *edge, EdgeClass **where_to_resume, int *
num_tetrahedra_ptr);
static FuncResult try_adjacent_fours(Tetrahedron *tet0, FaceIndex f0, FaceIndex f1,
EdgeClass **where_to_resume, int *num_tetrahedra_ptr);
static FuncResult create_new_order_four(EdgeClass *edge, EdgeClass **where_to_resume, int
*num_tetrahedra_ptr);
static Boolean four_tetrahedra_are_distinct(PositionedTet ptet);
static void set_inverse_neighbor_and_gluings(Tetrahedron *tet, FaceIndex f);

void basic_simplification(
    Triangulation *manifold)
{
    SolutionType original_solution_type[2];
    int iter;
    EdgeClass *edge,
    *where_to_resume;
    Boolean hyperbolic_structure_was_removed;

    /*
    * 97/2/3 Strip off the geometric structure if there is one.
    *
    * 98/5/20 Oops. We don't want to strip off the hyperbolic

```

```

    * structure if the cusp_nbhd_position is present, because the
    * low-level routines need the hyperbolic structure to maintain
    * cusp_nbhd_position.
    */
if (manifold->tet_list_begin.next->cusp_nbhd_position == NULL)
{
    original_solution_type[complete] = manifold->solution_type[complete];
    original_solution_type[filled] = manifold->solution_type[filled];
    remove_hyperbolic_structures(manifold);
    hyperbolic_structure_was_removed = TRUE;
}
else
    hyperbolic_structure_was_removed = FALSE;

/*
 * First do all the easy simplifications, namely removing
 * EdgeClasses of order 1, 2 and 3 when possible, and
 * retriangulating suspended pentagons with five Tetrahedra
 * instead of six.
 */

easy_simplification(manifold);

/*
 * Go down the list retriangulating the octahedra surrounding
 * EdgeClasses of order 4, in the hope of creating new, more
 * useful EdgeClasses of order 4. Keep doing this until we've
 * gone through the list ORDER_FOUR_ITERATIONS_IN_SIMPLIFY times
 * with no further progress.
 *
 * The operation of the inner loop is complicated by the
 * appearance and disappearance of EdgeClasses as the
 * algorithm proceeds. To avoid possible infinite loops,
 * and also to avoid possible "resonance" phenomena, we
 * pseudorandomly decide whether or not to perform each
 * potential retriangulation we encounter.
 */

for (iter = 0; iter < ORDER_FOUR_ITERATIONS_IN_SIMPLIFY; iter++)

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)

        if ((rand() & 3) > 0 /* proceed with probability 3/4 */
            && create_new_order_four(edge, &where_to_resume, &manifold->num_tetrahedra) == func_OK)
        {
            if (easy_simplification(manifold) == TRUE)
            {
                iter = -1;
                break;
            }
            else
                edge = where_to_resume;
        }

/*
 * Clean up.
 *
 * 97/2/3 If we trashed the tet shapes, reinitialize
 * them and then call polish_hyperbolic_structure(). Obviously
 * polish_hyperbolic_structure() will be recomputing the geometric
 * structure from scratch, not just "polishing" it.
 */

tidy_peripheral_curves(manifold);
if (hyperbolic_structure_was_removed
    && original_solution_type[complete] != not_attempted)
{
    manifold->solution_type[complete] = original_solution_type[complete];
    manifold->solution_type[filled] = original_solution_type[filled];
    initialize_tet_shapes(manifold);
    polish_hyperbolic_structures(manifold);
}

```

```

    }

    /*
     * The Chern-Simons invariant of the manifold is still the
     * same, but the fudge factor may have changed.
     */

    compute_CS_fudge_from_value(manifold);
}

void randomize_triangulation(
    Triangulation *manifold)
{
    SolutionType    original_solution_type[2];
    int             count;
    Boolean         hyperbolic_structure_was_removed;

    /*
     * 97/2/3 Strip off the geometric structure if there is one.
     *
     * 98/5/20 Oops. We don't want to strip off the hyperbolic
     * structure if the cusp_nbhd_position is present, because the
     * low-level routines need the hyperbolic structure to maintain
     * cusp_nbhd_position.
     */
    if (manifold->tet_list_begin.next->cusp_nbhd_position == NULL)
    {
        original_solution_type[complete] = manifold->solution_type[complete];
        original_solution_type[filled]   = manifold->solution_type[filled];
        remove_hyperbolic_structures(manifold);
        hyperbolic_structure_was_removed = TRUE;
    }
    else
        hyperbolic_structure_was_removed = FALSE;

    /*
     * Randomize the triangulation, doing only minimal
     * simplifications along the way. The minimal simplifications
     * are crucial -- otherwise the algorithm would create,
     * say, a pair of potentially cancelling Tetrahedra, and
     * then waste all it's remaining efforts making the union
     * of those two Tetrahedra more and more complex.
     *
     * By the way, not all the calls to two_to_three() will
     * succeed (e.g. because some Tetrahedra may be glued to
     * themselves), but that's OK.
     */

    for (count = RANDOMIZATION_MULTIPLE * manifold->num_tetrahedra; --count >= 0; )
    {
        if (two_to_three(
            get_tet(manifold, rand() % manifold->num_tetrahedra),
            rand() % 4,
            &manifold->num_tetrahedra)
            == func_OK)

            check_for_cancellation(manifold);

        /*
         * Resimplify the manifold.
         * basic_simplification() will tidy up the peripheral curves,
         * recompute the hyperbolic structure (if one is present),
         * and recompute the CS_fudge.
         */

        if (hyperbolic_structure_was_removed
            && original_solution_type[complete] != not_attempted)
        {
            manifold->solution_type[complete] = original_solution_type[complete];
            manifold->solution_type[filled]   = original_solution_type[filled];
            initialize_tet_shapes(manifold); /* unnecessary, but robust */
        }
    }
}

```

```

    basic_simplification(manifold);
}

static Tetrahedron *get_tet(
    Triangulation *manifold,
    int desired_index)
{
    int i;
    Tetrahedron *tet;

    /*
     * Return a pointer to the i-th Tetrahedron on the list,
     * with implicit numbering 0 through (num_tetrahedra - 1).
     */
    for (i = 0, tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         i++, tet = tet->next)

        if (i == desired_index)
            return tet;

    /*
     * If we get to here, something went wrong.
     */
    uFatalError("get_tet", "simplify_triangulation");

    /*
     * The C++ compiler would like a return value, even though
     * we never return from the uFatalError() call.
     */
    return NULL;
}

static void check_for_cancellation(
    Triangulation *manifold)
{
    Boolean progress;
    EdgeClass *edge,
              *where_to_resume;

    /*
     * This function is similar to easy_simplification() (see below),
     * except that it checks only for EdgeClasses of order 1 or 2.
     */

    do
    {
        progress = FALSE;

        for (edge = manifold->edge_list_begin.next;
             edge != &manifold->edge_list_end;
             edge = edge->next)

            switch (edge->order)
            {
                case 1:
                    if (remove_edge_of_order_one(edge, &where_to_resume, &manifold->
num_tetrahedra) == func_OK)
                    {
                        progress = TRUE;
                        edge = where_to_resume;
                    }
                    break;

                case 2:
                    if (cancel_tetrahedra(edge, &where_to_resume, &manifold->
num_tetrahedra) == func_OK)
                    {
                        progress = TRUE;
                        edge = where_to_resume;
                    }
                    break;
            }
    } while (progress);
}

```



```

        edge = where_to_resume;
    }
    break;

    case 3:
        if (three_to_two(edge, &where_to_resume, &manifold->num_tetrahedra) == func_OK)
        {
            progress = TRUE;
            triangulation_was_simplified = TRUE;
            edge = where_to_resume;
        }
        break;

    case 4:
        if (edges_of_order_four(edge, &where_to_resume, &manifold->num_tetrahedra) == func_OK)
        {
            progress = TRUE;
            triangulation_was_simplified = TRUE;
            edge = where_to_resume;
        }
        break;

    default:
        break;
}

} while (progress == TRUE);

return triangulation_was_simplified;
}

static FuncResult remove_edge_of_order_one(
    EdgeClass *edge,
    EdgeClass **where_to_resume,
    int *num_tetrahedra_ptr)
{
    Tetrahedron *tet;
    FaceIndex left_face,
              right_face,
              bottom_face;

    /*
     * remove_edge_of_order_one() contains no explicit low-level
     * retriangulation. Instead, each call to remove_edge_of_order_one()
     * calls two_to_three() to increase the order of EdgeClass *edge from
     * one to two, and then calls cancel_tetrahedra() to remove *edge.
     * Because two_to_three() increases the number of Tetrahedra by
     * one and cancel_tetrahedra() decreases it by two, there is a net
     * loss of one Tetrahedron.
     *
     * remove_edge_of_order_one() checks ahead of time whether the
     * calls to two_to_three() and cancel_tetrahedra() will be able
     * to succeed; if not (e.g. because of an embedded annulus),
     * remove_edge_of_order_one() does nothing and returns func_failed.
     *
     * The new EdgeClass created in the call to two_to_three() has
     * its order reduced to one in the call to cancel_tetrahedra().
     * Thus, remove_edge_of_order_one() always leave a new EdgeClass
     * of order one. Eventually remove_edge_of_order_one() will be
     * blocked by an annulus. Typically this annulus is trivial,
     * and opposite the EdgeClass of order 1 there is an EdgeClass of
     * order two, whose removal (by an independent call to
     * cancel_tetrahedra()) also destroys the EdgeClass of order 1.
     *
     * I'd like to draw some illustrations, but it just isn't possible
     * in a text-only file. So I'll leave it as an exercise for the
     * reader to illustrate what happens in the successive calls
     * to remove_edge_of_order_one().
     */
}

```



```

    * Label the Tetrahedron and the two faces incident to the
    * EdgeClass of order one.
    */
tet = edge->incident_tet;
left_face = one_face_at_edge[edge->incident_edge_index];
right_face = other_face_at_edge[edge->incident_edge_index];

/*
 * EdgeClasses of order 1 should never occur when a hyperbolic
 * structure is present.
 */

if (tet->shape[complete] != NULL)
    uFatalError("remove_edge_of_order_one", "simplify_triangulation");

/*
 * Let bottom_face be a candidate face for performing the
 * two-to-three move. Check ahead of time whether the calls
 * to two-to-three() and cancel_tetrahedra() will succeed.
 */

if (this_way_works(tet, left_face, right_face, remaining_face[left_face][right_face]) =✓
= TRUE)
    bottom_face = remaining_face[left_face][right_face];
else
if (this_way_works(tet, left_face, right_face, remaining_face[right_face][left_face]) =✓
= TRUE)
    bottom_face = remaining_face[right_face][left_face];
else
    return func_failed;

/*
 * Call two_to_three() and cancel_tetrahedra().
 */

if (
    two_to_three(tet, bottom_face, num_tetrahedra_ptr) == func_failed
    || edge->order != 2
    || cancel_tetrahedra(edge, where_to_resume, num_tetrahedra_ptr) == func_failed
)
    uFatalError("remove_edge_of_order_one", "simplify_triangulation");

return func_OK;
}

static Boolean this_way_works(
    Tetrahedron *tet,
    FaceIndex left_face,
    FaceIndex right_face,
    FaceIndex bottom_face)
{
    Tetrahedron *tet1;
    FaceIndex left1,
              right1,
              bottom1;
    EdgeClass *edgeA,
              *edgeB;

    /*
     * The left_ and right_faces fold together to form the EdgeClass
     * of order one.
     * The bottom_face cannot be glued to the remaining face of tet,
     * because if it were we'd have a manifold with only one Tetrahedron
     * but at least two EdgeClasses, which violates the proposition
     * that in a manifold with cusp cross sections of Euler characteristic
     * zero, the number of EdgeClasses must equal the number of
     * Tetrahedra.
     */
    /*
     * Oops! The reasoning in the preceding paragraph fails us
     * for finite triangulations (with honest vertices instead of
     * ideal vertices). In such cases it suffices simply to report
     * that the triangulation cannot be simplified. JRW 2002/08/26
     */

```

```

    */
    if (tet->neighbor[bottom_face] == tet)
/*      uFatalError("this_way_works", "simplify_triangulation");      */
        return FALSE;

/*
 * We want to locate the two EdgeClasses which would be combined
 * when remove_edge_of_order_one() calls cancel_tetrahedra().
 */

    tet1 = tet->neighbor[bottom_face];
    left1  = EVALUATE(tet->gluing[bottom_face], left_face);
    right1 = EVALUATE(tet->gluing[bottom_face], right_face);
    bottom1 = EVALUATE(tet->gluing[bottom_face], bottom_face);

    edgeA = tet1->edge_class[edge_between_vertices[bottom1][ left1]];
    edgeB = tet1->edge_class[edge_between_vertices[bottom1][right1]];

    return (edgeA != edgeB);
}

/*
 * cancel_tetrahedra() checks whether the two Tetrahedra
 * incident to the EdgeClass edge contain an annulus or
 * Moebius strip, and if they do not, it cancels them and
 * returns func_OK. If they do contain an annulus or Moebius
 * strip, cancel_tetrahedra() does nothing and returns func_failed.
 *
 * Comments in the code below explain how the cancellation
 * occurs, and why no other degenerate situations can occur.
 *
 * The imaginary parts of the logarithmic forms of the TetShapes
 * are computed mod 2 pi i, as explained at the top of this file.
 *
 * 97/2/4 Modified to allow for the possibility of two Tetrahedra
 * sharing three faces and the enclosed finite vertex. In this
 * case the annulus referred to above encloses a solid cylinder.
 * The tetrahedra are cancelled and the finite vertex is removed.
 */

FuncResult cancel_tetrahedra(
    EdgeClass *edge,
    EdgeClass **where_to_resume,
    int *num_tetrahedra_ptr)
{
    Tetrahedron *tet[2],
                *nbr[2],
                *t;
    VertexIndex v[2][4],
                w[2][4];
    Orientation orientation[2];
    EdgeClass *outer_edge[2];
    Boolean are_whole_manifold;
    int c,
        i,
        ii,
        j,
        k;
    int delta[2][2][2];
    VertexIndex active_vertex;
    Boolean tet_orientations_agree,
            edge_orientations_agree,
            edge_class_orientations_agree;
    PositionedTet ptet,
                  ptet0;
    EdgeIndex left_edge;
    Permutation gluing[2];

/*
 * Just to be safe . . .
 */

    if (edge->order != 2)

```

```

    uFatalError("cancel_tetrahedra", "simplify_triangulation");

/*
 * Let tet[0] and tet[1] be the two Tetrahedra incident
 * to EdgeClass *edge, and v[i][j] be their vertices.
 * Vertex v[0][i] is glued to vertex v[1][i].
 * Vertices v[i][0] and v[i][1] are incident to the
 * EdgeClass *edge.
 */

tet[0] = edge->incident_tet;
v[0][0] = one_vertex_at_edge[edge->incident_edge_index];
v[0][1] = other_vertex_at_edge[edge->incident_edge_index];
v[0][2] = remaining_face[v[0][1]][v[0][0]];
v[0][3] = remaining_face[v[0][0]][v[0][1]];
orientation[0] = right_handed;

if (tet[0]->neighbor[v[0][2]] != tet[0]->neighbor[v[0][3]]
    || tet[0]->gluing [v[0][2]] != tet[0]->gluing [v[0][3]])
    uFatalError("cancel_tetrahedra", "simplify_triangulation");

tet[1] = tet[0]->neighbor[v[0][2]];

for (i = 0; i < 4; i++)
    v[1][i] = EVALUATE(tet[0]->gluing[v[0][2]], v[0][i]);

orientation[1] = (parity[tet[0]->gluing[v[0][2]]] == orientation_preserving) ?
    orientation[0] :
    ! orientation[0];

/*
 * It's easy to prove that if the manifold has only torus and Klein
 * bottle cusp cross sections, then tet[0] and tet[1] are distinct.
 *
 * 97/2/4 I assume that the presence of at least one torus or
 * Klein bottle cusp is enough to guarantee that tet[0] != tet[1],
 * but I haven't thought through the details. Even if we eventually
 * wanted to use this code to simplify non-ideal triangulations
 * of closed manifolds, we could simply replace the uFatalError()
 * call with func_failed.
 *
 * 99/06/04 Indeed we do want this code to simplify non-ideal
 * triangulations of closed manifolds, so I replaced
 * uFatalError("cancel_tetrahedra", "simplify_triangulation");
 * with
 * return func_failed;
 */

if (tet[0] == tet[1])
    return func_failed;

/*
 * If the edge connecting v[0][2] to v[0][3] belongs to the same
 * EdgeClass as the edge connecting v[1][2] to v[1][3], then the
 * union of tet[0] and tet[1] contains an embedded annulus or
 * Moebius strip, and we should return func_failed.
 *
 * 97/2/4 Check whether tet[0] and tet[1] share three faces,
 * and enclose a finite vertex. If so, we may cancel the Tetrahedra,
 * and also remove the finite vertex. Obviously these changes will
 * never be invoked for ideal triangulations (i.e. with no finite
 * vertices).
 *
 * 2000/03/14 Oops! In the 97/2/4 change I overlooked the possibility
 * that tet[0] and tet[1] comprise the entire manifold (in which
 * case the manifold is a 3-sphere or L(3,1)). The code now tests
 * for this possibility, and returns func_failed when it occurs.
 */

for (i = 0; i < 2; i++)
    outer_edge[i] = tet[i]->edge_class[edge_between_vertices[v[i][2]][v[i][3]]];

if (outer_edge[0] == outer_edge[1])
{

```

```

    for (i = 0; i < 2; i++)

        if (tet[0]->cusp[v[0][i]]->is_finite == TRUE
            && tet[0]->neighbor[v[0][!i]] == tet[1]
            && tet[0]->neighbor[v[0][ i]] != tet[1]
            && tet[0]->gluing[v[0][!i]] == tet[0]->gluing[v[0][2]])

            return cancel_tetrahedra_with_finite_vertex(tet[0], v[0][i], edge,
where_to_resume, num_tetrahedra_ptr);

    return func_failed;
}

/*
 * The plan is to flatten the two Tetrahedra. To prove rigorously
 * that this does not change the topology of the manifold, first
 * imagine compressing the strip lying between the edge from
 * v[0][2] to v[0][3] and the edge from v[1][2] to v[1][3].
 * This is valid iff the two edges are in distinct EdgeClasses,
 * and we just checked that they are. Then imagine flattening
 * the two triangular pillows. This is OK iff the two triangular
 * pillows don't make up the whole manifold, which they don't
 * because otherwise the boundary would contain a sphere.
 * Q.E.D.
 *
 * 2000/03/14 Test explicitly whether the two triangular pillows
 * make up the whole manifold.
 */

are_whole_manifold = TRUE;

for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
        if (tet[i]->neighbor[v[i][j]] != tet[0]
            && tet[i]->neighbor[v[i][j]] != tet[1])
            are_whole_manifold = FALSE;

if (are_whole_manifold == TRUE)
    return func_failed;

/*
 * Before compressing the aforementioned strip, we need to clear
 * the peripheral curves away from the strip we're going to
 * collapse. While we're at it, we'll relabel all edges in
 * EdgeClass outer_edge[1] as EdgeClass outer_edge[0], and adjust
 * their edge_orientation if necessary, in preparation for merging
 * the two classes.
 */

for (c = 0; c < 2; c++)
    for (j = 0; j < 2; j++)
        for (i = 0; i < 2; i++)
            {
                ii = (orientation[0] == orientation[1]) ? i : !i;
                delta[c][j][i] = tet[1]->curve[c][ ii][v[1][j+2]][v[1][0]]
                    + tet[0]->curve[c][ii][v[0][j+2]][v[0][0]];
            }

tet_orientations_agree = (orientation[0] == orientation[1]);
edge_orientations_agree = (tet[0]->edge_orientation[edge_between_faces[v[0][0]][v[0]
[1]]]
                        == tet[1]->edge_orientation[edge_between_faces[v[1][0]][v[1]
[1]]]);
edge_class_orientations_agree = (tet_orientations_agree == edge_orientations_agree);

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

ptet = ptet0;
do

```

```

{
    /*
     * Adjust the peripheral curves.
     */
    for (c = 0; c < 2; c++)
        for (j = 0; j < 2; j++)
        {
            active_vertex = (j == 0) ? ptet.bottom_face : ptet.right_face;
            for (i = 0; i < 2; i++)
            {
                ii = (ptet.orientation == ptet0.orientation) ? i : !i;
                ptet.tet->curve[c][i][active_vertex][ptet.left_face] -= delta[c][j][ii]✎
;
                ptet.tet->curve[c][i][active_vertex][ptet.near_face] += delta[c][j][ii]✎
;
            }
        }

    /*
     * For convenience, note the EdgeIndex of the left edge.
     */
    left_edge = edge_between_faces[ptet.near_face][ptet.left_face];

    /*
     * Adjust the EdgeClass.
     */
    ptet.tet->edge_class[left_edge] = outer_edge[0];

    /*
     * Adjust the edge_orientation.
     */

    if ( ! edge_class_orientations_agree )
        ptet.tet->edge_orientation[left_edge] = ! ptet.tet->edge_orientation[left_edge]✎
;

    /*
     * Move on.
     */
    veer_left(&ptet);

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

/*
 * Adjust the EdgeClass sizes.
 */

outer_edge[0]->order += outer_edge[1]->order;

for (i = 0; i < 2; i++)
    for (j = 0; j < 6; j++)
        tet[i]->edge_class[j]->order--;

/*
 * We are about to delete EdgeClasses edge and outer_edge[1].
 * Set *where_to_resume to point to the EdgeClass just
 * just before the spot where edge was.
 */

if (edge->prev != outer_edge[1])
    *where_to_resume = edge->prev;
else
    *where_to_resume = outer_edge[1]->prev;

/*
 * Free the unused EdgeClasses.
 */

REMOVE_NODE(edge);
REMOVE_NODE(outer_edge[1]);
my_free(edge);
my_free(outer_edge[1]);

/*

```

```

    * Set the incident_tet and incident_edge_index fields
    * for all EdgeClasses which lost members.
    */

    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
        {
            t = tet[i]->neighbor[v[i][j]];
            if (t != tet[0] && t != tet[1])
                for (k = 0; k < 6; k++)
                {
                    t->edge_class[k]->incident_tet      = t;
                    t->edge_class[k]->incident_edge_index = k;
                }
        }

    /*
    * Set neighbors and gluings.
    */

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++) {
            nbr[j] = tet[j]->neighbor[v[j][i]];
            gluing[j] = tet[j]->gluing [v[j][i]];
            for (k = 0; k < 4; k++) /* which vertex */
                w[j][k] = EVALUATE(gluing[j], v[j][k]);
        }
        for (j = 0; j < 2; j++) /* which Tetrahedron */
        {
            nbr[j]->neighbor[w[j][i]] = nbr[!j];
            nbr[j]->gluing [w[j][i]] = CREATE_PERMUTATION(
                w[j][0], w[!j][0],
                w[j][1], w[!j][1],
                w[j][2], w[!j][2],
                w[j][3], w[!j][3]);
        }
    }

    /*
    * Free the collapsed Tetrahedra.
    */

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

    *num_tetrahedra_ptr -= 2;

    return func_OK;
}

static FuncResult cancel_tetrahedra_with_finite_vertex(
    Tetrahedron *tet,
    VertexIndex finite_vertex,
    EdgeClass *edge, /* needed only for setting *where_to_resume */
    EdgeClass **where_to_resume,
    int *num_tetrahedra_ptr)
{
    Tetrahedron *nbr,
                *tet_outer,
                *nbr_outer;
    Permutation gluing;
    FaceIndex f,
              ff,
              tet_outer_f,
              nbr_outer_f;
    VertexIndex v,
                nbr_finite;
    EdgeIndex e;
    Cusp *dead_cusp;

```

```

EdgeClass    *dead_edge;

/*
 * The three faces of the tet surrounding the finite_vertex
 * are glued to the neighboring Tetrahedron in the obvious way,
 * forming a triangular pillow with a finite vertex and three
 * EdgeClasses in its interior.
 */

nbr          = tet->neighbor[!finite_vertex];
gluing       = tet->gluing [!finite_vertex];

if (tet->cusp[finite_vertex]->is_finite != TRUE)
    uFatalError("cancel_tetrahedra_with_finite_vertex", "simplify_triangulation");

for (f = 0; f < 4; f++)
    if (f != finite_vertex)
        if (tet->neighbor[f] != nbr
            || tet->gluing [f] != gluing)
            uFatalError("cancel_tetrahedra_with_finite_vertex", "simplify_triangulation");

/*
 * If tet and nbr had four faces in common, then the manifold
 * couldn't have any torus or Klein bottle boundary components.
 * We assume this isn't the case.
 */
if (tet->neighbor[finite_vertex] == nbr)
    uFatalError("cancel_tetrahedra_with_finite_vertex", "simplify_triangulation");

/*
 * The peripheral curves will match up correctly after the cancellation.
 * No explicit preparation is required.
 */

/*
 * Remove the Cusp structure representing the finite vertex
 * in the triangular pillow's interior. Finite vertices aren't
 * counted in a Triangulation's num_cusps field. By removing
 * this Cusp we may leave a gap in the (negative) indexing of
 * finite vertex Cusps, but that's OK.
 */
dead_cusp = tet->cusp[finite_vertex];
REMOVE_NODE(dead_cusp);
my_free(dead_cusp);

/*
 * Remove the three EdgeClasses from the pillow's interior.
 *
 * Make sure the calling program is left with a valid pointer
 * to an EdgeClass, to continue its loop where it left off.
 */
*where_to_resume = edge->prev;
for (v = 0; v < 4; v++)
    if (v != finite_vertex)
    {
        dead_edge = tet->edge_class[edge_between_vertices[v][finite_vertex]];
        if (dead_edge == *where_to_resume)
            *where_to_resume = dead_edge->prev;
        REMOVE_NODE(dead_edge);
        my_free(dead_edge);
    }

/*
 * Note which Tetrahedra border the triangular pillow's outer faces.
 */

tet_outer    = tet->neighbor[finite_vertex];
tet_outer_f  = EVALUATE(tet->gluing[finite_vertex], finite_vertex);

nbr_finite   = EVALUATE(gluing, finite_vertex);
nbr_outer    = nbr->neighbor[nbr_finite];
nbr_outer_f  = EVALUATE(nbr->gluing[nbr_finite], nbr_finite);

```

```

/*
 * Make sure the three EdgeClasses around the pillow's boundary
 * "see" Tetrahedra other than the ones we are about to cancel.
 * Reduce the order of each such EdgeClass by two.
 */
for (f = 0; f < 4; f++)
    if (f != finite_vertex)
    {
        ff = EVALUATE(tet->gluing[finite_vertex], f);
        e = edge_between_faces[tet_outer_f][ff];
        tet_outer->edge_class[e]->incident_tet = tet_outer;
        tet_outer->edge_class[e]->incident_edge_index = e;
        tet_outer->edge_class[e]->order -= 2;
    }

/*
 * Glue tet_outer and nbr_outer to one another.
 * (Note: compose_permutations() composes right-to-left.)
 */

tet_outer->neighbor[tet_outer_f] = nbr_outer;
tet_outer->gluing[tet_outer_f] = compose_permutations(gluing, tet_outer->gluing
[tet_outer_f]);
tet_outer->gluing[tet_outer_f] = compose_permutations(nbr->gluing[nbr_finite],
tet_outer->gluing[tet_outer_f]);

nbr_outer->neighbor[nbr_outer_f] = tet_outer;
nbr_outer->gluing[nbr_outer_f] = compose_permutations(inverse_permutation[gluing],
nbr_outer->gluing[nbr_outer_f]);
nbr_outer->gluing[nbr_outer_f] = compose_permutations(tet->gluing[finite_vertex],
nbr_outer->gluing[nbr_outer_f]);

if (nbr_outer->gluing[nbr_outer_f] != inverse_permutation[tet_outer->gluing
[tet_outer_f]]
    || EVALUATE(tet_outer->gluing[tet_outer_f], tet_outer_f) != nbr_outer_f)
    uFatalError("cancel_tetrahedra_with_finite_vertex", "simplify_triangulation");

/*
 * Remove tet and nbr.
 */
REMOVE_NODE(tet);
REMOVE_NODE(nbr);
free_tetrahedron(tet);
free_tetrahedron(nbr);
*num_tetrahedra_ptr -= 2;

return func_OK;
}

/*
 * If the three Tetrahedra surrounding the EdgeClass *edge are distinct,
 * three_to_two() replaces them with two Tetrahedra sharing a common
 * face, and returns func_OK. Otherwise it does nothing and returns
 * func_failed.
 *
 * The Orientations of the two new Tetrahedra are set to match the
 * Orientation of one of the three old ones, so that the Orientability
 * of the Triangulation (if there is one) will be preserved.
 *
 * The two new Tetrahedra created by three_to_two() take the place
 * of one of the doomed ones in the list of Tetrahedra. The doomed
 * Tetrahedron are removed from the list before being destroyed.
 * Similarly, the EdgeClass edge is removed from its list before
 * being destroyed.
 *
 * If the three original Tetrahedra are nondegenerate, the two
 * two new ones must perforce be nondegenerate as well. Proof:
 * if a pair of ideal vertices coincides in a new Tetrahedra,
 * that pair must have coincided in one of the three original
 * Tetrahedra as well.
 *
 * The imaginary parts of the logarithmic forms of the TetShapes

```



```

* are computed mod 2 pi i, as explained at the top of this file.
*/

FuncResult three_to_two(
    EdgeClass *edge,
    EdgeClass **where_to_resume,
    int *num_tetrahedra_ptr)
{
    int c,
        h,
        hh,
        i,
        j,
        j1,
        j2;
    Tetrahedron *tet[3],
        *new_tet[2];
    VertexIndex v[3][4],
        w[2][4];
    Orientation old_orientation[3];
    Permutation gluing;
    EdgeIndex old_h_edge_index,
        old_v_edge_index,
        new_h_edge_index,
        new_v_edge_index;

    /*
     * Just to be safe . . .
     */

    if (edge->order != 3)
        uFatalError("three_to_two", "simplify_triangulation");

    /*
     * The three Tetrahedra incident to the EdgeClass *edge will be
     * called tet[0], tet[1] and tet[2]. The vertices of tet[i] will
     * be v[i][0-3].
     *
     * I recommend making a sketch of tet[0-2] to consult as you
     * read through the following code. The EdgeClass *edge is
     * vertical. Vertex v[i][0] of each tet[i] is at the bottom
     * of the edge, and vertex v[i][1] is at the top. Vertices
     * v[i][2] and v[i][3] are on the "equator", with v[i][3]
     * being counterclockwise from v[i][2] as viewed from above.
     */

    /*
     * Locate one Tetrahedron incident to EdgeClass *edge.
     * Choose the v[0][j] so that tet[0] is viewed with
     * the right_handed Orientation.
     */

    tet[0] = edge->incident_tet;
    v[0][0] = one_vertex_at_edge[edge->incident_edge_index];
    v[0][1] = other_vertex_at_edge[edge->incident_edge_index];
    v[0][2] = remaining_face[v[0][0]][v[0][1]];
    v[0][3] = remaining_face[v[0][1]][v[0][0]];
    old_orientation[0] = right_handed;

    /*
     * Locate the two remaining Tetrahedra.
     * If the Triangulation is oriented, they will also be positioned
     * with the right_handed Orientation.
     */

    for (i = 0; i < 2; i++)
    {
        tet[i+1] = tet[i]->neighbor[v[i][2]];
        gluing = tet[i]->gluing[v[i][2]];
        v[i+1][0] = EVALUATE(gluing, v[i][0]);
        v[i+1][1] = EVALUATE(gluing, v[i][1]);
        v[i+1][2] = EVALUATE(gluing, v[i][3]);
        v[i+1][3] = EVALUATE(gluing, v[i][2]);
        old_orientation[i+1] = (parity[gluing] == orientation_preserving) ?

```

```

        old_orientation[i] :
        ! old_orientation[i];
    }

/*
 * If the three Tetrahedra are not distinct, we can't do any
 * simplification, so return func_failed.
 */

for (i = 0; i < 3; i++)
    if (tet[i] == tet[(i+1)%3])
        return func_failed;

/*
 * This function should never be invoked when canonize_info is present.
 */

if (tet[0]->canonize_info != NULL)
    uFatalError("three_to_two", "simplify_triangulation");

/*
 * Create the new Tetrahedra.
 *
 * new_tet[0] occupies the northern half of the picture as described
 * above, and new_tet[1] occupies the southern half. Vertex w[0][3] of
 * new_tet[0] is at the north pole, and vertex w[1][3] of new_tet[1] is at
 * the south pole. Face w[i][j] (j = 0,1,2) of new_tet[i] coincides with
 * face v[j][i] of tet[j]. The actual values of w[][] give both
 * new_tet[0] and new_tet[1] the right_handed Orientation.
 */

for (i = 0; i < 2; i++)
{
    new_tet[i] = NEW_STRUCT(Tetrahedron);
    initialize_tetrahedron(new_tet[i]);
}

w[0][0] = 0;    w[0][1] = 1;    w[0][2] = 3;    w[0][3] = 2;
w[1][0] = 0;    w[1][1] = 1;    w[1][2] = 2;    w[1][3] = 3;

/*
 * Set the gluing and neighbor fields.
 *
 * Note that this code works correctly even if some of the faces
 * of the tet[i] were glued to each other.
 */

for (i = 0; i < 2; i++)
{
    for (j = 0; j < 3; j++)
    {
        new_tet[i]->neighbor[w[i][j]] = tet[j]->neighbor[v[j][i]];
        new_tet[i]->gluing [w[i][j]]
            = CREATE_PERMUTATION(
                w[i][j],          EVALUATE(tet[j]->gluing[v[j][i]], v[j][i]),
                w[i][(j+1)%3],    EVALUATE(tet[j]->gluing[v[j][i]], v[j][2]),
                w[i][(j+2)%3],    EVALUATE(tet[j]->gluing[v[j][i]], v[j][3]),
                w[i][3],          EVALUATE(tet[j]->gluing[v[j][i]], v[j][!i])
            );
        set_inverse_neighbor_and_gluing(new_tet[i], w[i][j]);
    }
    new_tet[i]->neighbor[w[i][3]] = new_tet[!i];
    new_tet[i]->gluing [w[i][3]] = CREATE_PERMUTATION(
        w[i][0], w[!i][0],
        w[i][1], w[!i][1],
        w[i][2], w[!i][2],
        w[i][3], w[!i][3]);
}

/*
 * Set the cusp fields.
 */

for (i = 0; i < 2; i++)

```

```

{
    for (j = 0; j < 3; j++)
        new_tet[i]->cuspl[w[i][j]] = tet[(j+1)%3]->cuspl[v[(j+1)%3][3]];
    new_tet[i]->cuspl[w[i][3]] = tet[0]->cuspl[v[0][!i]];
}

/*
 * Set the peripheral curves.
 */

for (c = 0; c < 2; c++)                /* which curve          */
    for (h = 0; h < 2; h++)              /* which sheet          */
        for (i = 0; i < 2; i++) {        /* which tetrahedron    */

            /*
             * Set the equatorial vertices.
             */
            for (j = 0; j < 3; j++)        /* which vertex          */
            {
                j1 = (j+1) % 3;
                j2 = (j+2) % 3;

                hh = (old_orientation[j1] == right_handed) ? h : !h;
                new_tet[i]->curve[c][h][w[i][j]][w[i][j1]] = tet[j1]->curve[c][hh][v
                [j1][3]][v[j1][i]];

                hh = (old_orientation[j2] == right_handed) ? h : !h;
                new_tet[i]->curve[c][h][w[i][j]][w[i][j2]] = tet[j2]->curve[c][hh][v
                [j2][2]][v[j2][i]];

                new_tet[i]->curve[c][h][w[i][j]][w[i][3]]
                    = - (new_tet[i]->curve[c][h][w[i][j]][w[i][j1]]
                        + new_tet[i]->curve[c][h][w[i][j]][w[i][j2]]);
            }

            /*
             * Set the polar vertices.
             */
            for (j = 0; j < 3; j++) {        /* which side of vertex 3 */
                hh = (old_orientation[j] == right_handed) ? h : !h;
                new_tet[i]->curve[c][h][w[i][3]][w[i][j]] = tet[j]->curve[c][hh][v[j][!
                i]][v[j][i]];
            }
        }

/*
 * Set where_to_resume to the predecessor of the EdgeClass about
 * to be killed, so that the loop in the calling function can
 * continue at the correct spot in the list.
 */

*where_to_resume = edge->prev;

/*
 * Kill the EdgeClass at the center of the three old Tetrahedra.
 */

REMOVE_NODE(edge);
my_free(edge);

/*
 * Update the surviving EdgeClasses.
 */

for (i = 0; i < 2; i++)
    for (j = 0; j < 3; j++)
    {
        j1 = (j+1) % 3;
        j2 = (j+2) % 3;

        old_h_edge_index = edge_between_vertices[v[j2][2]][v[j2][ 3]];
        old_v_edge_index = edge_between_vertices[v[j2][2]][v[j2][!i]];
        new_h_edge_index = edge_between_vertices[w[ i][j]][w[ i][j1]];
        new_v_edge_index = edge_between_vertices[w[ i][j]][w[ i][ 3]];
    }
}

```

```

    new_tet[i]->edge_class[new_h_edge_index]
        = tet[j2]->edge_class[old_h_edge_index];
    new_tet[i]->edge_class[new_v_edge_index]
        = tet[j2]->edge_class[old_v_edge_index];

    if (old_orientation[j2] == right_handed)
    {
        new_tet[i]->edge_orientation[new_h_edge_index]
            = tet[j2]->edge_orientation[old_h_edge_index];
        new_tet[i]->edge_orientation[new_v_edge_index]
            = tet[j2]->edge_orientation[old_v_edge_index];
    }
    else
    {
        new_tet[i]->edge_orientation[new_h_edge_index]
            = ! tet[j2]->edge_orientation[old_h_edge_index];
        new_tet[i]->edge_orientation[new_v_edge_index]
            = ! tet[j2]->edge_orientation[old_v_edge_index];
    }

    new_tet[i]->edge_class[new_v_edge_index]->order--;
    if (i == 0)
        new_tet[i]->edge_class[new_h_edge_index]->order++;

    new_tet[i]->edge_class[new_h_edge_index]->incident_tet = new_tet[i];
    new_tet[i]->edge_class[new_v_edge_index]->incident_tet = new_tet[i];
    new_tet[i]->edge_class[new_h_edge_index]->incident_edge_index =
new_h_edge_index;
    new_tet[i]->edge_class[new_v_edge_index]->incident_edge_index =
new_v_edge_index;
    }

/*
 * Compute the shapes of the new Tetrahedra iff
 * the old tetrahedra had shapes.
 */

if (tet[0]->shape[complete] != NULL)
{
    /*
     * Allocate space for the TetShapes of the new Tetrahedra.
     */
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            new_tet[i]->shape[j] = NEW_STRUCT(TetShape);

    /*
     * Add the complex edge angles of the old Tetrahedra
     * to get those of the new Tetrahedra. Use the
     * edge_orientation[] to get the orientations correct.
     * Note that add_edge_angles chooses angles in the range
     *  $[(-1/2)\pi, (3/2)\pi]$ , regardless of the angles of
     * summands.
     */
    for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
            add_edge_angles(
                tet[(j+1)%3], edge_between_vertices[v[(j+1)%3][3]][v[(j+1)%3][!i]],
                tet[(j+2)%3], edge_between_vertices[v[(j+2)%3][2]][v[(j+2)%3][!i]],
                new_tet[i],   edge_between_vertices[w[i][j]][w[i][3]]
            );
}

/*
 * Compute VertexCrossSections for the new Tetrahedra
 * iff the old Tetrahedra had VertexCrossSections.
 */

if (tet[0]->cross_section != NULL)
{
    /*
     * Begin with a quick error check.
     */

```

```

if (new_tet[0]->shape[complete] == NULL)
    uFatalError("three_to_two", "simplify_triangulation");

/*
 * Allocate space for the VertexCrossSections of the new Tetrahedra.
 */
for (i = 0; i < 2; i++)
    new_tet[i]->cross_section = NEW_STRUCT(VertexCrossSections);

/*
 * Compute the VertexCrossSections for each of the two new Tetrahedra.
 */
for (i = 0; i < 2; i++)
{
    /*
     * Compute the polar VertexCrossSections.
     */
    for (j = 0; j < 3; j++)
        new_tet[i]->cross_section->edge_length[w[i][3]][w[i][j]]
            = tet[j]->cross_section->edge_length[v[j][!i]][v[j][i]];
    new_tet[i]->cross_section->has_been_set[w[i][3]] = TRUE;

    /*
     * Compute the equatorial VertexCrossSections.
     */
    for (j = 0; j < 3; j++)
        compute_three_edge_lengths(new_tet[i], w[i][(j+1)%3], w[i][j],
            tet[j]->cross_section->edge_length[v[j][2]][v[j][i]]);
}

/*
 * Update the tilts.
 */
for (i = 0; i < 2; i++)
    compute_tilts_for_one_tet(new_tet[i]);
}

/*
 * Compute CuspNbhdPositions for the new Tetrahedra iff
 * the old Tetrahedra had CuspNbhdPositions.
 */
if (tet[0]->cusp_nbhd_position != NULL)
{
    /*
     * Begin with a quick error check.
     */
    if (new_tet[0]->shape[complete] == NULL)
        uFatalError("three_to_two", "simplify_triangulation");

    /*
     * Allocate space for the CuspNbhdPositions of the new Tetrahedra.
     */
    for (i = 0; i < 2; i++)
        new_tet[i]->cusp_nbhd_position = NEW_STRUCT(CuspNbhdPosition);

    /*
     * Compute the CuspNbhdPositions for each of the two new Tetrahedra.
     */
    for (i = 0; i < 2; i++)
    {
        /*
         * Consider both the right_handed and left_handed sheets.
         */
        for (h = 0; h < 2; h++)
        {
            /*
             * Compute the polar CuspNbhdPositions.
             *
             * The first approach which comes to mind is simply to copy
             * the relevant coordinates from the old Tetrahedra to the
             * new ones. Unfortunately the CuspNbhdPositions of the
             * three old tetrahedra may differ by covering translations.
             *

```

```

    * This problem is not insurmountable, but the code will be
    * cleaner if we simply copy the coordinates of two corners,
    * and then call cn_find_third_corner() to compute the
    * remaining corner. Recall that both new Tetrahedra are
    * seen in the right_handed Orientation, as is old tet[0].
    */
    if (tet[0]->cuspid_nbhd_position->in_use[h][v[0][!i]] == TRUE)
    {
        new_tet[i]->cuspid_nbhd_position->x[h][w[i][3]][w[i][1]]
            = tet[0]->cuspid_nbhd_position->x[h][v[0][!i]][v[0][2]];
        new_tet[i]->cuspid_nbhd_position->x[h][w[i][3]][w[i][2]]
            = tet[0]->cuspid_nbhd_position->x[h][v[0][!i]][v[0][3]];
        cn_find_third_corner(new_tet[i], h, w[i][3], w[i][1], w[i][2], w[i][0])
    }

    new_tet[i]->cuspid_nbhd_position->in_use[h][w[i][3]] = TRUE;
}
else
{
    new_tet[i]->cuspid_nbhd_position->x[h][w[i][3]][w[i][1]] = Zero;
    new_tet[i]->cuspid_nbhd_position->x[h][w[i][3]][w[i][2]] = Zero;
    new_tet[i]->cuspid_nbhd_position->x[h][w[i][3]][w[i][0]] = Zero;

    new_tet[i]->cuspid_nbhd_position->in_use[h][w[i][3]] = FALSE;
}

/*
 * Compute the equatorial CuspidNbhdPositions.
 */
/*
 * Technical note: The new_tets are both seen with the
 * right_handed Orientation. So when old_orientation[]
 * is also right_handed, we want to read the new sheet h
 * from the old sheet h. But when old_orientation[] is
 * left_handed, we want to read the new sheet h from the
 * old sheet !h. Because left_handed == 1, the expression
 * (old_orientation[] ^ h) gives the correct old sheet
 * to read from.
 */
for (j = 0; j < 3; j++)
{
    if (tet[j]->cuspid_nbhd_position->in_use[old_orientation[j]^h][v[j][2]] ==
= TRUE)
    {
        new_tet[i]->cuspid_nbhd_position->x[h][w[i][(j+1)%3]][w[i][(j+2)%3]]
            = tet[j]->cuspid_nbhd_position->x[old_orientation[j]^h][v[j][2]]
[v[j][3]];
        new_tet[i]->cuspid_nbhd_position->x[h][w[i][(j+1)%3]][w[i][3]]
            = tet[j]->cuspid_nbhd_position->x[old_orientation[j]^h][v[j][2]]
[v[j][!i]];
        cn_find_third_corner(new_tet[i], h, w[i][(j+1)%3], w[i][(j+2)%3], w
[i][3], w[i][j]);

        new_tet[i]->cuspid_nbhd_position->in_use[h][w[i][(j+1)%3]] = TRUE;
    }
    else
    {
        new_tet[i]->cuspid_nbhd_position->x[h][w[i][(j+1)%3]][w[i][(j+2)%3]]
= Zero;
        new_tet[i]->cuspid_nbhd_position->x[h][w[i][(j+1)%3]][w[i][3]]
= Zero;
        new_tet[i]->cuspid_nbhd_position->x[h][w[i][(j+1)%3]][w[i][j]]
= Zero;

        new_tet[i]->cuspid_nbhd_position->in_use[h][w[i][(j+1)%3]] = FALSE;
    }
}
}

/*
 * Put the new Tetrahedra on the list, and remove and free
 * the old ones.
 */

```

```

    for (i = 0; i < 2; i++)
        INSERT_BEFORE(new_tet[i], tet[0]);

    for (i = 0; i < 3; i++)
    {
        REMOVE_NODE(tet[i]);
        free_tetrahedron(tet[i]);
    }

    *num_tetrahedra_ptr -= 1;

    return func_OK;
}

/*
 * The three new Tetrahedra created by two_to_three() take tet0's place
 * in the list of Tetrahedra. Tet0 and the other doomed Tetrahedron are
 * removed from the list before being destroyed. Similarly, the new
 * EdgeClass is added to the list of EdgeClasses just in front of one
 * of the existing EdgeClasses.
 *
 * The Orientations of the three new Tetrahedra are set to match the
 * Orientation of one of the two old ones, so that the Orientability
 * of the Triangulation (if there is one) will be preserved.
 *
 * two_to_three() returns func_failed if either
 *
 * (1) the two initial Tetrahedra are not not distinct (i.e. tet0
 *     is glued to itself at face f), or
 *
 * (2) a hyperbolic structure is present, and even though the two
 *     initial Tetrahedra are combinatorially distinct, they are
 *     superimposed in hyperbolic space (i.e. the vertices opposite
 *     their common face, though combinatorially distinct, lie at the
 *     same point on the sphere at infinity). In this case, performing
 *     the two_to_three() move would create degenerate Tetrahedra.
 *
 * The imaginary parts of the logarithmic forms of the TetShapes
 * are computed mod 2 pi i, as explained at the top of this file.
 */

FuncResult two_to_three(
    Tetrahedron *tet0,
    FaceIndex f,
    int *num_tetrahedra_ptr)
{
    Tetrahedron *tet[2],
                *new_tet[3];
    VertexIndex v[2][4];
    Orientation old_orientation[2];
    int c,
        h,
        hh,
        i,
        i1,
        i2,
        j,
        k;
    EdgeClass *new_class;

    /*
     * two_to_three() is the inverse of three_to_two(), and
     * is implemented similarly. In particular, the picture to
     * imagine (or, better yet, draw on a scrap of paper before
     * diving into this code) is virtually identical to that from
     * three_to_two(), only what was tet[] there is new_tet[] here,
     * and vice versa.
     */

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

```

```

tet[0] = tet0;
v[0][3] = f;
v[0][0] = !f;          /* v[0][0] is some vertex other than v[0][3] */
v[0][1] = remaining_face[v[0][3]][v[0][0]]; /* tet[0] will be seen */
v[0][2] = remaining_face[v[0][0]][v[0][3]]; /* as left_handed */
old_orientation[0] = left_handed;

tet[1] = tet[0]->neighbor[f];
for (i = 0; i < 4; i++)
    v[1][i] = EVALUATE(tet[0]->gluing[f], v[0][i]);
old_orientation[1] = (parity[tet[0]->gluing[f]] == orientation_preserving) ?
    old_orientation[0] : ! old_orientation[0];

/*
 * If tet[0] and tet[1] are not distinct, we cannot proceed.
 */

if (tet[0] == tet[1])
    return func_failed;

/*
 * If a hyperbolic structure is present and the 2-3 move would create
 * degenerate Tetrahedra, we do not want to proceed. Degenerate
 * Tetrahedra will be created iff vertices v[0][3] and v[1][3] coincide.
 * (People usually think of degeneracy in terms of the dihedral angles
 * approaching {0, 1, infinity}, but in this context we also think in
 * terms of the equivalent definition that a Tetrahedron is degenerate
 * iff two or more vertices coincide.)
 *
 * angles_sum_to_zero() check whether the angles sum to zero mod 2 pi.
 */

if (tet[0]->shape[complete] != NULL)
    if (angles_sum_to_zero(
        tet[0], edge_between_vertices[v[0][0]][v[0][1]],
        tet[1], edge_between_vertices[v[1][0]][v[1][1]]))

        return func_failed;

/*
 * Allocate the three new Tetrahedra.
 */

for (i = 0; i < 3; i++)
{
    new_tet[i] = NEW_STRUCT(Tetrahedron);
    initialize_tetrahedron(new_tet[i]);
}

/*
 * Note that here we can refer to the VertexIndices of the new_tet[i]
 * directly, because a symmetrical indexing scheme is consistent
 * with a fixed orientation. In three_to_two(), the symmetrical
 * indexing scheme was not consistent with a fixed orientation,
 * so we had to use the w[][] to store the true indices of the
 * new Tetrahedra.
 */

/*
 * Set "internal" neighbors and gluings.
 */
for (i = 0; i < 3; i++)
{
    i1 = (i+1) % 3;
    i2 = (i+2) % 3;
    new_tet[i]->neighbor[2] = new_tet[i1];
    new_tet[i]->neighbor[3] = new_tet[i2];
    new_tet[i]->gluing[2] = CREATE_PERMUTATION(0, 0, 1, 1, 2, 3, 3, 2);
    new_tet[i]->gluing[3] = CREATE_PERMUTATION(0, 0, 1, 1, 2, 3, 3, 2);
}

/*
 * Set "external" neighbors and gluings.

```



```

* This code works even if some of the external faces of tet[0]
* and tet[1] are glued to each other.
*/

for (i = 0; i < 3; i++)          /* which new Tetrahedron */
    for (j = 0; j < 2; j++)      /* which face */
    {
        new_tet[i]->neighbor[j] = tet[j]->neighbor[v[j][i]];
        new_tet[i]->gluing[j]
            = CREATE_PERMUTATION(
                j, EVALUATE(tet[j]->gluing[v[j][i]], v[j][i]),
                !j, EVALUATE(tet[j]->gluing[v[j][i]], v[j][3]),
                2, EVALUATE(tet[j]->gluing[v[j][i]], v[j][(i+1)%3]),
                3, EVALUATE(tet[j]->gluing[v[j][i]], v[j][(i+2)%3])
            );
        set_inverse_neighbor_and_gluing(new_tet[i], j);
    }

/*
* Set the cusp fields.
*/

for (i = 0; i < 3; i++)
{
    new_tet[i]->cusp[0] = tet[1]->cusp[v[1][3]];
    new_tet[i]->cusp[1] = tet[0]->cusp[v[0][3]];
    new_tet[i]->cusp[2] = tet[0]->cusp[v[0][(i+1)%3]];
    new_tet[i]->cusp[3] = tet[0]->cusp[v[0][(i+2)%3]];
}

/*
* Set the peripheral curves.
*/

for (c = 0; c < 2; c++)          /* which curve */
    for (h = 0; h < 2; h++) {    /* which sheet */

        /*
        * Set the exterior sides of the polar vertices.
        */
        for (i = 0; i < 3; i++)    /* which tetrahedron */
            for (j = 0; j < 2; j++) { /* which vertex */
                hh = (old_orientation[!j] == left_handed) ? h : !h;
                new_tet[i]->curve[c][h][j][!j] = tet[!j]->curve[c][hh][v[!j][3]][v[!j]
[i]];
            }

        /*
        * Set the interior sides of the polar vertices.
        */
        for (i = 0; i < 3; i++)    /* which tetrahedron */
            for (j = 0; j < 2; j++) /* which vertex */
                for (k = 2; k < 4; k++) /* which side */
                    new_tet[i]->curve[c][h][j][k] = - FLOW(
                        new_tet[ i ]->curve[c][h][j][!j],
                        new_tet[(i+k-1)%3]->curve[c][h][j][!j]);

        /*
        * Set the equatorial vertices.
        */
        for (i = 0; i < 3; i++)    /* which tetrahedron */
            for (j = 2; j < 4; j++) { /* which vertex */
                for (k = 0; k < 2; k++) /* which side */
                {
                    hh = (old_orientation[k] == left_handed) ? h : !h;
                    new_tet[i]->curve[c][h][j][k] = tet[k]->curve[c][hh][v[k][(i+j-1)%
3]][v[k][i]];
                }
                new_tet[i]->curve[c][h][j][5-j] = - (new_tet[i]->curve[c][h][j][0] +
new_tet[i]->curve[c][h][j][1]);
            }
    }

/*

```

```

    * Create the new EdgeClass.
    */

new_class = NEW_STRUCT(EdgeClass);
initialize_edge_class(new_class);
new_class->order = 3;
new_class->incident_tet = new_tet[0];
new_class->incident_edge_index = edge_between_vertices[0][1];

/*
 * Insert the new EdgeClass at an arbitrary spot in the linked list.
 */

INSERT_BEFORE(new_class, tet[0]->edge_class[0]);

/*
 * Set the EdgeClasses.
 */

for (i = 0; i < 3; i++)
{
    i1 = (i+1) % 3;
    i2 = (i+2) % 3;

    new_tet[i]->edge_class[edge_between_vertices[0][1]] = new_class;
    new_tet[i]->edge_class[edge_between_vertices[0][2]] = tet[1]->edge_class
    [edge_between_vertices[v[1][3]][v[1][i1]]];
    new_tet[i]->edge_class[edge_between_vertices[0][3]] = tet[1]->edge_class
    [edge_between_vertices[v[1][3]][v[1][i2]]];
    new_tet[i]->edge_class[edge_between_vertices[1][2]] = tet[0]->edge_class
    [edge_between_vertices[v[0][3]][v[0][i1]]];
    new_tet[i]->edge_class[edge_between_vertices[1][3]] = tet[0]->edge_class
    [edge_between_vertices[v[0][3]][v[0][i2]]];
    new_tet[i]->edge_class[edge_between_vertices[2][3]] = tet[0]->edge_class
    [edge_between_vertices[v[0][i1]][v[0][i2]]];
}

/*
 * Set the edge_orientations.
 */

for (i = 0; i < 3; i++)
{
    i1 = (i+1) % 3;
    i2 = (i+2) % 3;

    new_tet[i]->edge_orientation[edge_between_vertices[0][1]]
        = right_handed;
    new_tet[i]->edge_orientation[edge_between_vertices[0][2]]
        = (old_orientation[1] == left_handed) ?
            tet[1]->edge_orientation[edge_between_vertices[v[1][3]][v[1][i1]]] :
            ! tet[1]->edge_orientation[edge_between_vertices[v[1][3]][v[1][i1]]];
    new_tet[i]->edge_orientation[edge_between_vertices[0][3]]
        = (old_orientation[1] == left_handed) ?
            tet[1]->edge_orientation[edge_between_vertices[v[1][3]][v[1][i2]]] :
            ! tet[1]->edge_orientation[edge_between_vertices[v[1][3]][v[1][i2]]];
    new_tet[i]->edge_orientation[edge_between_vertices[1][2]]
        = (old_orientation[0] == left_handed) ?
            tet[0]->edge_orientation[edge_between_vertices[v[0][3]][v[0][i1]]] :
            ! tet[0]->edge_orientation[edge_between_vertices[v[0][3]][v[0][i1]]];
    new_tet[i]->edge_orientation[edge_between_vertices[1][3]]
        = (old_orientation[0] == left_handed) ?
            tet[0]->edge_orientation[edge_between_vertices[v[0][3]][v[0][i2]]] :
            ! tet[0]->edge_orientation[edge_between_vertices[v[0][3]][v[0][i2]]];
    new_tet[i]->edge_orientation[edge_between_vertices[2][3]]
        = (old_orientation[0] == left_handed) ?
            tet[0]->edge_orientation[edge_between_vertices[v[0][i1]][v[0][i2]]] :
            ! tet[0]->edge_orientation[edge_between_vertices[v[0][i1]][v[0][i2]]];
}

/*
 * Adjust the EdgeClass orders.
 */

```

```

for (i = 0; i < 3; i++)
{
    new_tet[i]->edge_class[edge_between_vertices[0][2]]->order++;
    new_tet[i]->edge_class[edge_between_vertices[1][2]]->order++;
    new_tet[i]->edge_class[edge_between_vertices[2][3]]->order--;
}

/*
 * Set incident_tets and incident_edge_indices.
 */

for (i = 0; i < 3; i++)
    for (j = 0; j < 6; j++)
    {
        new_tet[i]->edge_class[j]->incident_tet      = new_tet[i];
        new_tet[i]->edge_class[j]->incident_edge_index = j;
    }

/*
 * Compute the shapes of the new Tetrahedra iff
 * the old tetrahedra had shapes.
 */

if (tet[0]->shape[complete] != NULL)
{
    /*
     * Allocate space for the TetShapes of the new Tetrahedra.
     */
    for (i = 0; i < 3; i++)
        for (j = 0; j < 2; j++)
            new_tet[i]->shape[j] = NEW_STRUCT(TetShape);

    /*
     * First compute the TetShapes for the equatorial angles.
     */
    for (i = 0; i < 3; i++) /* which new Tetrahedron */
        add_edge_angles(
            tet[0],      edge_between_vertices[v[0][(i+1)%3]][v[0][(i+2)%3]],
            tet[1],      edge_between_vertices[v[1][(i+1)%3]][v[1][(i+2)%3]],
            new_tet[i], edge_between_vertices[2][3]
        );

    /*
     * Now compute the remaining complex angles of each Tetrahedron,
     * with log.imag in the range  $[(-1/2)\pi, (3/2)\pi]$ .
     */
    for (i = 0; i < 3; i++) /* which new Tetrahedron */
        compute_remaining_angles(new_tet[i], edge3_between_vertices[2][3]);
}

/*
 * Compute VertexCrossSections for the new Tetrahedra
 * iff the old Tetrahedra had VertexCrossSections.
 */

if (tet[0]->cross_section != NULL)
{
    /*
     * Begin with a quick error check.
     */

    if (new_tet[0]->shape[complete] == NULL)
        uFatalError("two_to_three", "simplify_triangulation");

    /*
     * Allocate space for the VertexCrossSections of the new Tetrahedra.
     */
    for (i = 0; i < 3; i++)
        new_tet[i]->cross_section = NEW_STRUCT(VertexCrossSections);

    /*
     * Compute the VertexCrossSections for each of
     * the three new Tetrahedra.
     */
}

```

```

    for (i = 0; i < 3; i++)
    {
        /*
         * Compute the polar vertices.
         */
        for (j = 0; j < 2; j++)
            compute_three_edge_lengths(new_tet[i], !j, j,
                                       tet[j]->cross_section->edge_length[v[j][3]][v[j][i]]);

        /*
         * Compute the equatorial vertices.
         */
        for (j = 2; j < 4; j++)
            compute_three_edge_lengths(new_tet[i], j, 0,
                                       tet[0]->cross_section->edge_length[v[0][(i+j+2)%3]][v[0][i]]);
    }

    /*
     * Update the tilts.
     */

    for (i = 0; i < 3; i++)
        compute_tilts_for_one_tet(new_tet[i]);
}

/*
 * Provide CanonizeInfo for the new Tetrahedra
 * iff the old Tetrahedra had CanonizeInfo.
 */

if (tet[0]->canonize_info != NULL)
{
    /*
     * Allocate space for the CanonizeInfo of the new Tetrahedra.
     */
    for (i = 0; i < 3; i++)
        new_tet[i]->canonize_info = NEW_STRUCT(CanonizeInfo);

    /*
     * Set part_of_coned_cell to TRUE for each new Tetrahedron.
     */
    for (i = 0; i < 3; i++)
        new_tet[i]->canonize_info->part_of_coned_cell = TRUE;

    /*
     * Set each new "exterior" face to have the same face_status
     * as the corresponding old face.
     */
    for (i = 0; i < 3; i++)
        for (j = 0; j < 2; j++)
            new_tet[i]->canonize_info->face_status[j]
                = tet[j]->canonize_info->face_status[v[j][i]];

    /*
     * Set each new "interior" face to have face_status inside_cone_face.
     */
    for (i = 0; i < 3; i++)
        for (j = 2; j < 4; j++)
            new_tet[i]->canonize_info->face_status[j] = inside_cone_face;
}

/*
 * Compute CuspNbhdPositions for the new Tetrahedra iff
 * the old Tetrahedra had CuspNbhdPositions.
 */

if (tet[0]->cusp_nbhd_position != NULL)
{
    /*
     * Begin with a quick error check.
     */
    if (new_tet[0]->shape[complete] == NULL)
        uFatalError("two_to_three", "simplify_triangulation");
}

```

```

/*
 * Allocate space for the CuspNbhdPositions of the new Tetrahedra.
 */
for (i = 0; i < 3; i++)
    new_tet[i]->cusp_nbhd_position = NEW_STRUCT(CuspNbhdPosition);

/*
 * Compute the CuspNbhdPositions for each of the three new Tetrahedra.
 */
for (i = 0; i < 3; i++)
{
    /*
     * Consider both the right_handed and left_handed sheets.
     */
    for (h = 0; h < 2; h++)
    {
        /*
         * Compute the polar CuspNbhdPositions.
         *
         * Technical note: The new_tets are all seen with the
         * left_handed Orientation. So when old_orientation[]
         * is also left_handed, we want to read the new sheet h
         * from the old sheet h. But when old_orientation[] is
         * right_handed, we want to read the new sheet h from the
         * old sheet !h. Because left_handed == 1, the expression
         * (old_orientation[] == h) gives the correct old sheet
         * to read from.
         */
        for (j = 0; j < 2; j++)
        {
            if (tet[j]->cusp_nbhd_position->in_use[old_orientation[j]==h][v[j][3]]
== TRUE)
            {
                new_tet[i]->cusp_nbhd_position->x[h][!j][2]
                = tet[j]->cusp_nbhd_position->x[old_orientation[j]==h][v[j][3]]
[v[j][(i+1)%3]];
                new_tet[i]->cusp_nbhd_position->x[h][!j][3]
                = tet[j]->cusp_nbhd_position->x[old_orientation[j]==h][v[j][3]]
[v[j][(i+2)%3]];
                cn_find_third_corner(new_tet[i], h, !j, 2, 3, j);

                new_tet[i]->cusp_nbhd_position->in_use[h][!j] = TRUE;
            }
            else
            {
                new_tet[i]->cusp_nbhd_position->x[h][!j][2] = Zero;
                new_tet[i]->cusp_nbhd_position->x[h][!j][3] = Zero;
                new_tet[i]->cusp_nbhd_position->x[h][!j][j] = Zero;

                new_tet[i]->cusp_nbhd_position->in_use[h][!j] = FALSE;
            }
        }

        /*
         * Compute the equatorial CuspNbhdPositions.
         *
         * The three new Tetrahedra are seen in the left_handed
         * Orientation, as is the old tet[0]. So if we coordinates
         * coordinates from tet[0] we know the sheets will match up.
         */
        for (j = 2; j < 4; j++)
        {
            if (tet[0]->cusp_nbhd_position->in_use[h][v[0][(i+j+2)%3]] == TRUE)
            {
                new_tet[i]->cusp_nbhd_position->x[h][j][5-j]
                = tet[0]->cusp_nbhd_position->x[h][v[0][(i+j+2)%3]][v[0][(4+i-
j)%3]];
                new_tet[i]->cusp_nbhd_position->x[h][j][1]
                = tet[0]->cusp_nbhd_position->x[h][v[0][(i+j+2)%3]][v[0][3]];
                cn_find_third_corner(new_tet[i], h, j, 5-j, 1, 0);

                new_tet[i]->cusp_nbhd_position->in_use[h][j] = TRUE;
            }
        }
    }
}

```

```

        else
        {
            new_tet[i]->cuspid_nbhd_position->x[h][j][5-j] = Zero;
            new_tet[i]->cuspid_nbhd_position->x[h][j][ 1 ] = Zero;
            new_tet[i]->cuspid_nbhd_position->x[h][j][ 0 ] = Zero;

            new_tet[i]->cuspid_nbhd_position->in_use[h][j] = FALSE;
        }
    }
}

/*
 * Put the new Tetrahedra on the list, and remove and free
 * the old ones.
 */

for (i = 0; i < 3; i++)
    INSERT_BEFORE(new_tet[i], tet[0]);

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

*num_tetrahedra_ptr += 1;

return func_OK;
}

/*
 * one_to_four() performs a one-to-four move, replacing a Tetrahedron
 * with four new Tetrahedra meeting at a finite vertex. It adds
 * the four new Tetrahedra and the four new EdgeClasses to the
 * appropriate lists.
 */

void one_to_four(
    Tetrahedron *tet,
    int          *num_tetrahedra_ptr,
    int          new_cuspid_index)
{
    int          c,
                h,
                i,
                j,
                k;
    Tetrahedron *new_tet[4];
    Cusp         *new_cuspid;
    EdgeClass    *new_class[4];

    /*
     * It doesn't make sense to call this function when a hyperbolic
     * structure, VertexCrossSections or a CuspidNbhdPosition are present.
     */

    if (tet->shape[complete] != NULL
        || tet->cross_section != NULL
        || tet->cuspid_nbhd_position != NULL)
        uFatalError("one_to_four", "simplify_triangulation");

    /*
     * To understand this code, I recommend you first make a drawing
     * of a truncated ideal tetrahedron, and draw its subdivision into
     * four tetrahedra meeting at the center. The four new Tetrahedra
     * are indexed in the natural way: each vertex which coincides with
     * a vertex of the old Tetrahedron inherits the latter's VertexIndex,
     * while the vertex at the center gets the VertexIndex from the
     * "opposite" vertex of the old Tetrahedron.
     *
     * Note that if the manifold is oriented, this scheme preserves
     * the orientation.
     */

```

```

    */

/*
 * Allocate space for the new Tetrahedra.
 *
 * new_tet[i] will be the new Tetrahedron incident to face i
 * of the old Tetrahedron.
 */

for (i = 0; i < 4; i++)
{
    new_tet[i] = NEW_STRUCT(Tetrahedron);
    initialize_tetrahedron(new_tet[i]);
}

/*
 * Set neighbors and gluings.
 *
 * This code works even if some of tet's external faces
 * are glued to each other.
 */

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

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

        if (j == i)      /* "external" neighbor */
        {
            new_tet[i]->neighbor[j] = tet->neighbor[i];
            new_tet[i]->gluing[j]    = tet->gluing[i];
            set_inverse_neighbor_and_gluing(new_tet[i], j);
        }

        else {           /* "internal" neighbor */
            new_tet[i]->neighbor[j] = new_tet[j];
            new_tet[i]->gluing[j]    = CREATE_PERMUTATION(
                remaining_face[i][j],    remaining_face[i][j],
                remaining_face[j][i],    remaining_face[j][i],
                i,                        j,
                j,                        i);
        }

/*
 * Create a Cusp structure for the finite vertex.
 */

new_cusp = NEW_STRUCT(Cusp);
initialize_cusp(new_cusp);
new_cusp->is_finite = TRUE;
new_cusp->index      = new_cusp_index;
INSERT_BEFORE(new_cusp, tet->cusp[0]);

/*
 * Set the new Tetrahedra's cusp fields.
 */

for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
        new_tet[i]->cusp[j] =
            j == i ?
                new_cusp :      /* finite vertex */
                tet->cusp[j];   /* ideal vertex */

/*
 * Set the peripheral curves.
 */

for (c = 0; c < 2; c++)                /* which curve */
    for (h = 0; h < 2; h++)              /* which sheet */
        for (i = 0; i < 4; i++)          /* which tetrahedron */
            for (j = 0; j < 4; j++)      /* which vertex */

                if (j == i)
                    /*

```

```

        * Set the peripheral curves on the finite vertex
        * to zero.
        */
    for (k = 0; k < 4; k++) /* which side */
        new_tet[i]->curve[c][h][i][k] = 0;

    else
    /*
    * Set the curves on the new ideal vertices in
    * terms of those on the old.
    */
    for (k = 0; k < 4; k++) /* which side */

        if (k == j)
            /*
            * The j-th vertex has no j-th side.
            * Do nothing.
            */
            ;
        else if (k == i)
            /*
            * The i-th side of vertex j on new
            * Tetrahedron i coincides with the
            * i-th side of vertex j on the old
            * Tetrahedron.
            */
            new_tet[i]->curve[c][h][j][i] =
                tet->curve[c][h][j][i];
        else
            /*
            * Compute the new side as a FLOW
            * between two of the old sides.
            */
            new_tet[i]->curve[c][h][j][k] = FLOW(
                tet->curve[c][h][j][k],
                tet->curve[c][h][j][i]);

    /*
    * Create the new EdgeClasses.
    */

    for (i = 0; i < 4; i++)
    {
        new_class[i] = NEW_STRUCT(EdgeClass);
        initialize_edge_class(new_class[i]);
        new_class[i]->order = 3;
        new_class[i]->incident_tet = new_tet[!i];
        new_class[i]->incident_edge_index = edge_between_vertices[i][!i];
    }

    /*
    * Insert the new EdgeClasses at an arbitrary spot in the linked list.
    */

    for (i = 0; i < 4; i++)
        INSERT_BEFORE(new_class[i], tet->edge_class[0]);

    /*
    * Set the edge_class and edge_orientation fields of the new Tetrahedra.
    */

    for (i = 0; i < 4; i++) /* which Tetrahedron */
        for (j = 0; j < 4; j++) /* VertexIndex at one end of edge */
            for (k = j + 1; k < 4; k++) /* VertexIndex at other end of edge */
            {
                new_tet[i]->edge_class[edge_between_vertices[j][k]] =
                    (j == i || k == i) ?
                    (j == i ? new_class[k] : new_class[j]) :
                    tet->edge_class[edge_between_vertices[j][k]];

                new_tet[i]->edge_orientation[edge_between_vertices[j][k]] =
                    (j == i || k == i) ?
                    right_handed :
                    tet->edge_orientation[edge_between_vertices[j][k]];
            }

```



```

    }

    /*
    * Adjust the EdgeClass orders of the preexisting EdgeClasses.
    * (The orders of the new EdgeClasses were set above.)
    */

    for (i = 0; i < 6; i++)
        tet->edge_class[i]->order++;

    /*
    * Set incident_tets and incident_edge_indices for the preexisting
    * EdgeClasses. (Those for the new EdgeClasses were set above.)
    */
    for (i = 0; i < 6; i++)
    {
        tet->edge_class[i]->incident_tet      = new_tet[one_face_at_edge[i]];
        tet->edge_class[i]->incident_edge_index = i;
    }

    /*
    * Provide CanonizeInfo for the new Tetrahedra iff the old Tetrahedron
    * had CanonizeInfo.
    */

    if (tet->canonize_info != NULL)
    {
        /*
        * For each new Tetrahedron . . .
        */
        for (i = 0; i < 4; i++)
        {
            /*
            * Allocate space for the CanonizeInfo.
            */
            new_tet[i]->canonize_info = NEW_STRUCT(CanonizeInfo);

            /*
            * Set part_of_coned_cell to TRUE.
            */
            new_tet[i]->canonize_info->part_of_coned_cell = TRUE;

            /*
            * Set face_status to inside_cone_face for each "interior" face,
            * and have it match the old values for the "exterior" faces.
            */
            for (j = 0; j < 4; j++) /* which face */
                new_tet[i]->canonize_info->face_status[j] =
                    j == i ?
                    tet->canonize_info->face_status[j] : /* exterior face */
                    inside_cone_face; /* interior face */
        }
    }

    /*
    * Put the new Tetrahedra on the list, and remove and free the old one.
    */

    for (i = 0; i < 4; i++)
        INSERT_BEFORE(new_tet[i], tet);

    REMOVE_NODE(tet);
    free_tetrahedron(tet);

    *num_tetrahedra_ptr += 3;
}

static FuncResult edges_of_order_four(
    EdgeClass *edge,
    EdgeClass **where_to_resume,
    int *num_tetrahedra_ptr)
{
    PositionedTet ptet0,

```

```

        ptet;

/*
 * *edge is an EdgeClass of order 4. Look for another EdgeClass
 * of order 4 which shares a triangle with *edge. If the six
 * Tetrahedra incident to the two EdgeClasses are all distinct,
 * then their union is a suspended pentagon which will be
 * retriangulated with only five Tetrahedra.
 */

ptet0.tet = edge->incident_tet;
ptet0.bottom_face = one_vertex_at_edge[edge->incident_edge_index];
ptet0.right_face = other_vertex_at_edge[edge->incident_edge_index];
ptet0.near_face = remaining_face[ptet0.bottom_face][ptet0.right_face];
ptet0.left_face = remaining_face[ptet0.right_face][ptet0.bottom_face];
ptet0.orientation = right_handed;

ptet = ptet0;
do
{
    if (ptet.tet->edge_class[edge_between_faces[ptet.near_face][ptet.right_face]]->
order == 4)
        if (try_adjacent_fours(ptet.tet, ptet.near_face, ptet.bottom_face,
where_to_resume, num_tetrahedra_ptr) == func_OK)
            return func_OK;

    if (ptet.tet->edge_class[edge_between_faces[ptet.near_face][ptet.bottom_face]]->
order == 4)
        if (try_adjacent_fours(ptet.tet, ptet.near_face, ptet.right_face,
where_to_resume, num_tetrahedra_ptr) == func_OK)
            return func_OK;

    veer_left(&ptet);

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

return func_failed;
}

static FuncResult try_adjacent_fours(
Tetrahedron *tet0,
FaceIndex f0,
FaceIndex f1,
EdgeClass **where_to_resume,
int *num_tetrahedra_ptr)
{
Tetrahedron *tet[6];
FaceIndex f2,
f3,
g2,
g3;
int i,
j;
EdgeClass *class0,
*class1;

/*
 * Two nonantipodal EdgeClasses of order 4 lies on tet. The
 * face between them has index f0. The face not incident to
 * either has index f1.
 */

/*
 * Find the six Tetrahedra adjacent to the EdgeClasses of order 4.
 */
tet[0] = tet0;
f2 = remaining_face[f0][f1];
f3 = remaining_face[f1][f0];

tet[1] = tet0->neighbor[f0];
g2 = EVALUATE(tet0->gluing[f0], f2);
g3 = EVALUATE(tet0->gluing[f0], f3);

```

```

tet[2] = tet[0]->neighbor[f2];
tet[3] = tet[0]->neighbor[f3];
tet[4] = tet[1]->neighbor[g2];
tet[5] = tet[1]->neighbor[g3];

/*
 * If the six Tetrahedra aren't all distinct, return func_failed.
 * (Thought question: Might simplification sometimes be possible
 * even if all six Tetrahedra aren't distinct? Hmmm . . . seems
 * unlikely.)
 */
for (i = 0; i < 6; i++)
    for (j = i + 1; j < 6; j++)
        if (tet[i] == tet[j])
            return func_failed;

/*
 * Note the two EdgeClasses which now have order four.
 */
class0 = tet0->edge_class[edge_between_faces[f0][f2]];
class1 = tet0->edge_class[edge_between_faces[f0][f3]];

/*
 * The following two-to-three move increases the number of
 * Tetrahedra by one, but it creates two EdgeClasses of
 * order three . . .
 */
if (two_to_three(tet0, f0, num_tetrahedra_ptr) == func_failed)
{
    /*
     * (There can't be any topological obstruction to the
     * retriangulation, but there might be a geometric obstruction,
     * namely that the two_to_three() move might require creation
     * of degenerate Tetrahedra. So if two_to_three() fails when
     * a hyperbolic structure is present, we assume (potential)
     * degenerate Tetrahedra are the cause, and we return func_failed.
     * Otherwise we call uFatalError().)
     */
    if (tet0->shape[complete] != NULL)
        return func_failed;
    else
        uFatalError("try_adjacent_fours", "simplify_triangulation");
}

/*
 * . . . each of which can be used to reduce the number of
 * Tetrahedra by one.
 */
if (three_to_two(class0, where_to_resume, num_tetrahedra_ptr) == func_failed
    || three_to_two(class1, where_to_resume, num_tetrahedra_ptr) == func_failed)
    uFatalError("try_adjacent_fours", "simplify_triangulation");

/*
 * Note that where_to_resume will come out pointing to some
 * valid EdgeClass. We won't worry too much about just which
 * one it points at.
 */

return func_OK;
}

static FuncResult create_new_order_four(
    EdgeClass *edge,
    EdgeClass **where_to_resume,
    int *num_tetrahedra_ptr)
{
    PositionedTet ptet0,
    ptet;

    if (edge->order != 4)
        return func_failed;

    /*

```

```

    * create_new_order_four() is similar to edges_of_order_four().
    *
    * *edge is an EdgeClass of order 4. Look for another EdgeClass
    * of order 5 or less which shares a triangle with *edge.
    * If the four Tetrahedra incident to *edge are all distinct,
    * then their union is an octagon which will be retriangulated
    * so as to create a new EdgeClass of order 4 or less.
    */

    ptet0.tet = edge->incident_tet;
    ptet0.bottom_face = one_vertex_at_edge[edge->incident_edge_index];
    ptet0.right_face = other_vertex_at_edge[edge->incident_edge_index];
    ptet0.near_face = remaining_face[ptet0.bottom_face][ptet0.right_face];
    ptet0.left_face = remaining_face[ptet0.right_face][ptet0.bottom_face];
    ptet0.orientation = right_handed;

    if (four_tetrahedra_are_distinct(ptet0) == FALSE)
        return func_failed;

    ptet = ptet0;
    do
    {
        if (ptet.tet->edge_class[edge_between_faces[ptet.near_face][ptet.right_face]]->
order <= 5
        || ptet.tet->edge_class[edge_between_faces[ptet.near_face][ptet.bottom_face]]->
order <= 5)
        {
            if (two_to_three(ptet.tet, ptet.near_face, num_tetrahedra_ptr) == func_OK)
            {
                if (three_to_two(edge, where_to_resume, num_tetrahedra_ptr) == func_OK)
                    return func_OK;
                else
                    uFatalError("create_new_order_four", "simplify_triangulation");
            }
            else
            {
                /*
                * The call to two_to_three() failed. It can't fail for
                * topological reasons (we checked that the four Tetrahedra
                * surrounding the EdgeClass of order 4 are distinct), but
                * if a hyperbolic structure is present it might fail
                * because two antipodal vertices of the octahedron
                * coincide. In the latter case, we simply move on in
                * the hope that a different retriangulation will work.
                */
                if (ptet.tet->shape[complete] == NULL)
                    uFatalError("create_new_order_four", "simplify_triangulation");
                /*
                * else continue with do loop
                */
            }
        }

        veer_left(&ptet);

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

    return func_failed;
}

static Boolean four_tetrahedra_are_distinct(
    PositionedTet ptet)
{
    int i,
        j;
    Tetrahedron *tet[4];

    for (i = 0; i < 4; i++)
    {
        tet[i] = ptet.tet;
        veer_left(&ptet);
    }
}

```

```
    for (i = 0; i < 4; i++)
        for (j = i + 1; j < 4; j++)
            if (tet[i] == tet[j])
                return FALSE;

    return TRUE;
}

static void set_inverse_neighbor_and_gluing(
    Tetrahedron *tet,
    FaceIndex    f)
{
    tet->neighbor[f]->neighbor[EVALUATE(tet->gluing[f], f)]
        = tet;
    tet->neighbor[f]->gluing [EVALUATE(tet->gluing[f], f)]
        = inverse_permutation[tet->gluing[f]];
}
```