

```

/*
 * normal_surface_splitting.c
 *
 * This file contains the function
 *
 *      FuncResult split_along_normal_surface(
 *
 *                      NormalSurfaceList  *surface_list,
 *                      int                 index,
 *                      Triangulation      *pieces[2]);
 *
 * which splits the surface_list->triangulation along the normal surface
 * of the given index. All of the surface_list->triangulation's cusps
 * must be complete (no Dehn fillings) and the normal surface must be a
 * connected surface of nonnegative Euler characteristic. If the normal
 * surface is a 2-sided projective plane, split_along_normal_surface()
 * returns func_bad_input; otherwise it returns func_OK. If the normal
 * surface is a sphere or 1-sided projective plane, the resulting
 * spherical boundary component(s) are capped off with 3-ball(s);
 * otherwise the new torus or Klein bottle boundary component(s) become
 * cusp(s). If the normal surface is nonseparating, the result is
 * returned in pieces[0], and pieces[1] is set to NULL. If the normal
 * surface is separating, the two pieces are returned in pieces[0] and
 * pieces[1]. The original surface_list->triangulation is left unchanged,
 * except for the fact that the normal surface data is copied into the
 * parallel_edge, num_squares and num_triangles fields of its Tetrahedra.
 */

/*
 * The Algorithm
 *
 * To understand this documentation, you should first read normal_surfaces.h
 * to learn what a normal surface is, and how its squares and triangles
 * sit in a tetrahedron. Then you should make yourself a drawing of an
 * ideal tetrahedron with a couple squares across its middle and a
 * triangle or two at each ideal vertex. Draw the squares in red and
 * the triangles in blue, to highlight the differences between the pieces
 * they bound.
 *
 * The red squares and blue triangles cut the tetrahedron into pieces.
 * Keeping in mind that the red squares and blue triangles will be
 * "pulled to infinity" to become ideal vertices, the pieces may be
 * classified as follows.
 *
 *      tetrahedra
 *      If there are no red squares cutting across the ideal tetrahedron,
 *      then the central piece will itself be an ideal tetrahedron.
 *
 *      pillows
 *      A piece incident to a red square and two blue triangles has
 *      four faces: two ideal triangles and two "bigons". We call
 *      this piece a pillow. (It would be a true triangular pillow
 *      if each "bigonal" face were collapsed to a single edge.)
 *
 *      square prisms
 *      A piece incident to two red squares is a square prism.
 *      (When the two red squares are pulled to infinity this piece
 *      will become long and skinny, but don't let that distract you.)
 *
 *      triangular prisms
 *      A piece incident to two blue triangles is a triangular prism.
 *      (When the two blue triangles are pulled to infinity this piece
 *      will become long and skinny, but don't let that distract you.)
 *
 * In the remainder of this explanation, "manifold" will refer to the
 * manifold obtained by splitting the original manifold along the given
 * normal surface. The manifold will have either one or two connected
 * components, according to whether the normal surface was separating.
 *
 * Initially the manifold has the cell division consisting of the four
 * types of pieces described above. We will subdivide it into tetrahedra.
 * The subdivision will introduce finite (non-ideal) vertices, so the
 * tetrahedra will be "hybrids", with some ideal vertices and some finite
 * vertices.

```

```

* subdividing the 1-skeleton
*   Introduce a finite vertex in the interior of each line in
*   the 1-skeleton. You should draw the finite vertex at the
*   midpoint of each line in your drawing, even though
*   the concept of "midpoint" has no intrinsic meaning in an
*   infinite line which runs from one ideal vertex to another.
*
* subdividing the 2-skeleton
*   Subdivide each bigon in the 2-skeleton by introducing a line
*   segment connecting the finite vertices at the "midpoints"
*   of its two edges. Subdivide each triangle in the 2-skeleton
*   by introducing three line segments connecting the finite
*   vertices at the "midpoints" of the triangle's edges.
*
* subdividing the 3-skeleton
*   Each type of piece (triangular prism, square prism, pillow
*   and tetrahedron) is subdivided differently. The details are
*   explained in the code itself. The important points are that
*   each piece is subdivided into tetrahedra, and the subdivision
*   is consistent with the subdivision of the 2-skeleton described
*   above. At least one tetrahedron in each piece is guaranteed
*   to have the correct orientation; split_along_normal_surface()
*   eventually extends that orientation to the whole connected
*   component of the triangulation. (In case you're wondering why
*   all tetrahedra aren't given the correct orientation right way,
*   the reason is that it would require a more complicated and less
*   natural indexing system.)
*
* We'll define the indexing system (i.e. the assignment of VertexIndices
* to the vertices in the subdivision) only on the boundary of each
* (original) tetrahedron. The SubdivisionData structure will assign to
* each triangle (in the subdivision of each face of each original
* tetrahedron) a pointer to the tetrahedron (in the subdivision of the
* original Tetrahedron's interior) which is borders, as well as a
* permutation mapping the boundary triangle's VertexIndices (defined
* in a canonical way in the following paragraph) to the tetrahedron's
* more or less arbitrary VertexIndices.
*
* You'll want to make yourself a drawing as we go along. First draw
* an ideal triangle representing a face of one of the original
* tetrahedra. Label its vertices 'a', 'b' and 'c'. (These are the
* VertexIndices which come with the original Tetrahedron; they take
* values in {0,1,2,3}.) Draw a couple red lines showing where the
* red squares meet this face, and draw blue lines showing where the
* blue triangles meet it. Then, in black, draw the subdivision of
* the 2-skeleton as defined above in the section "subdividing the
* 2-skeleton". Each ideal vertex in the subdivision (remember, ideal
* vertices include the red and blue line segments, which will eventually
* be "pulled to infinity") gets the VertexIndex of the nearest ideal
* vertex of the large triangle. Each finite vertex (each black dot
* along an edge of the large triangle) gets the VertexIndex of the
* vertex of the large triangle opposite the edge the black dot's on.
* Note that each finite vertex is incident to two faces of the original
* tetrahedron, and it gets assigned a different VertexIndex on each;
* this inconsistency is harmless.
*/

/*
* Note: In a previous attempt at a splitting algorithm, I had hoped
* to let the square and triangular prisms become lines, let the pillows
* become triangles, and use the tetrahedra to triangulate the resulting
* manifold(s). For an explanation of why this works for complicated
* manifolds but fails for simple ones, please see
* normal_surface_splitting.old.c.
*/

#include "kernel.h"
#include "normal_surfaces.h"

/*
* A TetReference says which (new, small) Tetrahedron borders each triangle
* in the subdivision of a boundary face of the (original, large) Tetrahedron.
*/
typedef struct

```

```

{
    /*
     * Which Tetrahedron do we see?
     */
    Tetrahedron *tet;

    /*
     * How do the canonical VertexIndices of the subdivision (defined above)
     * map to the actual vertices of the tet?
     */
    Permutation gluing;
} TetReference;

typedef struct
{
    /*
     * A bigon (on a face of an original large tetrahedron) get subdivided
     * into a pair of semi-ideal triangles. The triangles closer to the
     * cusp is called the "outer" triangle, and the other one is called
     * the "inner" triangles.
     */
    TetReference    outer,
                   inner;
} TetReferencePair;

/*
 * Each (original, large) Tetrahedron has lots of (new, small) triangles
 * on its boundary. The SubdivisionData structure organizes the
 * TetReferences assigned to them.
 */
typedef struct
{
    /*
     * central[f][v] holds the TetReference for the central triangle
     * on face f. central[f][v] holds the TetReference for the the
     * triangle bordering the central triangle on the side closest
     * to vertex v of the original large triangle.
     */
    TetReference    central[4][4];

    /*
     * At each ideal vertex of each original large face, there may be
     * any number of bigons (each divided into two semi-ideal triangles).
     * side_array_length[f][v] tells how many such bigons there are at
     * ideal vertex v of face f, and side_array[f][v] is an array of
     * TetReferencePairs for the new, small Tetrahedra they see.
     * (side_array_length[f][f] and side_array[f][f] are unused.)
     */
    int             side_array_length[4][4];
    TetReferencePair *side_array[4][4];
} SubdivisionData;

/*
 * The IdealVertexReference structure is used only in distinguish_cusps().
 */
typedef struct
{
    Tetrahedron *tet;
    VertexIndex v;
} IdealVertexReference;

static Boolean    is_two_sided_projective_plane(NormalSurfaceList *surface_list, int index);
static void      install_normal_surface(NormalSurfaceList *surface_list, int index);
static Triangulation *subdivide_manifold(Triangulation *manifold, Boolean is_two_sided, int Euler_characteristic);
static SubdivisionData *allocate_subdivision_data(Triangulation *manifold);
static void      free_subdivision_data(SubdivisionData *data, int num_old_tetrahedra);
static void      copy_cusps(Triangulation *manifold, Triangulation *subdivision);

```

```

static void      subdivide_old_tetrahedron(Tetrahedron *old_tet, Triangulation *
subdivision, SubdivisionData *tet_data, Cusp *cusp_at_split);
static void      subdivide_triangular_prism(Tetrahedron *old_tet, VertexIndex old_v,
int index, Triangulation *subdivision, SubdivisionData *tet_data, Cusp *cusp_at_split)
;
static void      subdivide_central_tetrahedron(Tetrahedron *old_tet, Triangulation *
subdivision, SubdivisionData *tet_data, Cusp *cusp_at_split);
static void      subdivide_pillow(Tetrahedron *old_tet, EdgeIndex defining_edge,
Triangulation *subdivision, SubdivisionData *tet_data, Cusp *cusp_at_split);
static void      subdivide_square_prism(Tetrahedron *old_tet, int index,
Triangulation *subdivision, SubdivisionData *tet_data, Cusp *cusp_at_split);
static void      glue_external_faces(Triangulation *manifold, SubdivisionData *data)
;
static Permutation compute_external_gluings(Permutation perm0_inverse, Permutation
perm1, Permutation perm2);
static void      distinguish_cusps(Triangulation *subdivision, Cusp *new_cusps[2]);
static void      separate_connected_components(Triangulation *subdivision,
Triangulation *pieces[2]);
static Tetrahedron *find_correctly_oriented_tet(Triangulation *manifold);

```

```

FuncResult split_along_normal_surface(
NormalSurfaceList *surface_list,
int index,
Triangulation *pieces[2])
{
Triangulation *subdivision;
int i;

/*
 * Dispose of a rare special case, before getting on to the
 * main algorithm.
 *
 * The correct way to handle a 2-sided projective plane would be
 * to split along it and cap off each boundary surface by coning
 * to a point, thereby producing an orbifold with two singular points.
 * SnapPea isn't prepared to do this.
 */
if (is_two_sided_projective_plane(surface_list, index) == TRUE)
{
uAcknowledge("Can't cut along 2-sided projective planes.");
pieces[0] = NULL;
pieces[1] = NULL;
return func_bad_input;
}

/*
 * Copy the requested normal surface into surface_list->triangulation.
 */
install_normal_surface(surface_list, index);

/*
 * The present version of the software assumes all cusps are complete.
 */
if (all_cusps_are_complete(surface_list->triangulation) == FALSE)
uFatalError("split_along_normal_surface", "normal_surface_splitting");

/*
 * Subdivide the manifold. The result may or may not be connected.
 * subdivide_manifold() creates Tetrahedra and real Cusps,
 * but not EdgeClasses or fake Cusps ("fake Cusps" are Cusp structures
 * for finite vertices).
 */
subdivision = subdivide_manifold(
surface_list->triangulation,
surface_list->list[index].is_two_sided,
surface_list->list[index].Euler_characteristic);

/*
 * Separate the subdivision into its connected components.
 * If the subdivision is connected, pieces[1] will be set to NULL.
 */
separate_connected_components(subdivision, pieces);

```

```

/*
 * Spruce up the two pieces.
 */
for (i = 0; i < 2; i++)
    if (pieces[i] != NULL)
    {
        /*
         * The subdivision algorithm promises to provide the correct
         * orientation for at least one tetrahedron in each piece.
         * Extend this orientation to all of pieces[i].
         */
        extend_orientation(pieces[i], find_correctly_oriented_tet(pieces[i]));

        /*
         * Install "fake cusps" for the finite vertices.
         */
        create_fake_cusps(pieces[i]);

        /*
         * Install and orient the EdgeClasses.
         */
        create_edge_classes(pieces[i]);
        orient_edge_classes(pieces[i]);

        /*
         * Retriangulate with no finite vertices.
         *
         * Note: We haven't set the cusp topologies or num_or_cusps
         * and num_nonor_cusps, but remove_finite_vertices()
         * doesn't care.
         *
         * Note: If pieces[i] is a closed manifold,
         * remove_finite_vertices() will drill out an arbitrary cusp.
         */
        remove_finite_vertices(pieces[i]);

        /*
         * Install peripheral curves only for those cusps which
         * don't already have them. For cusps which have them,
         * keep the originals.
         */
        peripheral_curves_as_needed(pieces[i]);
        count_cusps(pieces[i]);

        /*
         * The splitting may have turned a nonorientable manifold
         * into one or more orientable pieces, in which case
         * some of the original {meridian, longitude} pairs might
         * fail to obey the right-hand rule.
         */
        if (pieces[i]->orientability == oriented_manifold)
            fix_peripheral_orientations(pieces[i]);

        /*
         * Find the hyperbolic structure.
         */
        find_complete_hyperbolic_structure(pieces[i]);
    }

/*
 * Free the subdivision, which has no Tetrahedra or Cusps left anyhow.
 */
free_triangulation(subdivision);

/*
 * All done!
 */
return func_OK;
}

```

```

static Boolean is_two_sided_projective_plane(
    NormalSurfaceList *surface_list,
    int index)

```

```

{
    return surface_list->list[index].is_connected      == TRUE
        && surface_list->list[index].is_two_sided      == TRUE
        && surface_list->list[index].Euler_characteristic == 1;
}

static void install_normal_surface(
    NormalSurfaceList *surface_list,
    int index)
{
    Tetrahedron *old_tet;
    VertexIndex v;

    if (index < 0 || index >= surface_list->num_normal_surfaces)
        uFatalError("install_normal_surface", "normal_surface_splitting");

    for (old_tet = surface_list->triangulation->tet_list_begin.next;
         old_tet != &surface_list->triangulation->tet_list_end;
         old_tet = old_tet->next)
    {
        old_tet->parallel_edge = surface_list->list[index].parallel_edge[old_tet->index];
        old_tet->num_squares    = surface_list->list[index].num_squares [old_tet->index];
        for (v = 0; v < 4; v++)
            old_tet->num_triangles[v] = surface_list->list[index].num_triangles[old_tet->
index][v];
    }
}

static Triangulation *subdivide_manifold(
    Triangulation *manifold,
    Boolean is_two_sided,
    int Euler_characteristic)
{
    Triangulation *subdivision;
    Cusp *new_cusps[2];
    SubdivisionData *data;
    Tetrahedron *old_tet;

    /*
     * Create a Triangulation structure to hold the new Tetrahedra.
     */
    subdivision = NEW_STRUCT(Triangulation);
    initialize_triangulation(subdivision);

    /*
     * Create copies of the old Cusps, for use in the subdivision.
     * Each old Cusp's matching_cusp field is set to point to its
     * corresponding new Cusp in the subdivision.
     */
    copy_cusps(manifold, subdivision);

    /*
     * Allocate new Cusps as necessary.
     */
    switch (Euler_characteristic)
    {
        case 2:
            /*
             * We're cutting along a sphere, so treat the boundary
             * as a finite vertex (to automatically fill it in).
             */
            new_cusps[0] = NULL;
            new_cusps[1] = NULL;
            break;

        case 1:
            /*
             * We're cutting along a projective plane. If it's 1-sided
             * we'll get a spherical boundary component which should
             * be filled as in the spherical case immediately above.
             * If it's 2-sided, we're not prepared to handle it.
             */

```

```

    if (is_two_sided == FALSE)
    {
        new_cusps[0] = NULL;
        new_cusps[1] = NULL;
    }
    else
        uFatalError("subdivide_manifold", "normal_surface_splitting");
    break;

case 0:
    /*
     * We're cutting along a torus or Klein bottle.
     * Allocate one or two cusps as necessary.
     */

    new_cusps[0] = NEW_STRUCT(Cusp);
    initialize_cusp(new_cusps[0]);
    INSERT_BEFORE(new_cusps[0], &subdivision->cusp_list_end);
    new_cusps[0]->index = subdivision->num_cusps++;

    if (is_two_sided == TRUE)
    {
        new_cusps[1] = NEW_STRUCT(Cusp);
        initialize_cusp(new_cusps[1]);
        INSERT_BEFORE(new_cusps[1], &subdivision->cusp_list_end);
        new_cusps[1]->index = subdivision->num_cusps++;
    }
    else
        new_cusps[1] = NULL;
    break;

default:
    uFatalError("subdivide_manifold", "normal_surface_splitting");
}

/*
 * Allocate an array of SubdivisionData structures, one structure
 * for each old Tetrahedron in the original unsplit manifold.
 */
data = allocate_subdivision_data(manifold);

/*
 * Create the new Tetrahedra which subdivide each old Tetrahedron.
 * Set their internal neighbor and gluing fields, which specify
 * how they glue to each other. Set tet->cusp fields to NULL for
 * finite vertices, to the correct copy of an old cusp for ideal
 * vertices which are incident to an old cusp, and to new_cusps[0]
 * for ideal vertices which are incident to the normal surface.
 * (In the case of a 2-sided torus or Klein bottle, some references
 * to new_cusps[0] will be corrected to new_cusps[1] below.)
 */
for (old_tet = manifold->tet_list_begin.next;
     old_tet != &manifold->tet_list_end;
     old_tet = old_tet->next)

    subdivide_old_tetrahedron(old_tet, subdivision, &data[old_tet->index], new_cusps
[0]);

/*
 * Set the external neighbor and gluing fields, which connect
 * the (small, new) tetrahedra within one (large, old) tetrahedron
 * to those within another.
 */
glue_external_faces(manifold, data);

/*
 * For a 2-sided torus or Klein bottle, we have to change some
 * references from new_cusps[0] to new_cusps[1].
 * Change an arbitrary tet->cusp[v] from new_cusps[0] to new_cusps[1],
 * and then recursively change its neighbors.
 */
if (Euler_characteristic == 0 && is_two_sided == TRUE)
    distinguish_cusps(subdivision, new_cusps);

```

```

/*
 * Free the SubdivisionData array and its attached arrays,
 * but not the new Tetrahedra themselves, of course.
 */
free_subdivision_data(data, manifold->num_tetrahedra);

/*
 * Return the subdivision, which may contain one or two
 * connected components.
 */
return subdivision;
}

static SubdivisionData *allocate_subdivision_data(
    Triangulation *manifold)
{
    SubdivisionData *data;
    Tetrahedron *old_tet;
    FaceIndex f;
    VertexIndex v;
    int length;
    TetReferencePair *array;
    int i;

    data = NEW_ARRAY(manifold->num_tetrahedra, SubdivisionData);

    for (old_tet = manifold->tet_list_begin.next;
        old_tet != &manifold->tet_list_end;
        old_tet = old_tet->next)
    {
        /*
         * Set the central references to NULL.
         */
        for (f = 0; f < 4; f++)
            for (v = 0; v < 4; v++)
            {
                data[old_tet->index].central[f][v].tet = NULL;
                data[old_tet->index].central[f][v].gluing = 0;
            }

        /*
         * Initialize the number of bigons at each vertex of each face
         * to be the number of blue triangles.
         */
        for (f = 0; f < 4; f++)
            for (v = 0; v < 4; v++)
                if (v != f)
                    data[old_tet->index].side_array_length[f][v] = old_tet->num_triangles
[v];
                else
                    data[old_tet->index].side_array_length[f][v] = 0;

        /*
         * If there are any red squares, add in their contribution
         * to the number of bigons. (As usual, a picture makes
         * all this clear.)
         */
        if (old_tet->num_squares != 0)
        {
            data[old_tet->index].side_array_length
                [one_vertex_at_edge [ old_tet->parallel_edge ]]
                [other_vertex_at_edge[ old_tet->parallel_edge ]]
                += old_tet->num_squares;
            data[old_tet->index].side_array_length
                [other_vertex_at_edge[ old_tet->parallel_edge ]]
                [one_vertex_at_edge [ old_tet->parallel_edge ]]
                += old_tet->num_squares;
            data[old_tet->index].side_array_length
                [one_vertex_at_edge [5 - old_tet->parallel_edge]]
                [other_vertex_at_edge[5 - old_tet->parallel_edge]]
                += old_tet->num_squares;
            data[old_tet->index].side_array_length
                [other_vertex_at_edge[5 - old_tet->parallel_edge]]

```



```

        [one_vertex_at_edge [5 - old_tet->parallel_edge]]
        += old_tet->num_squares;
    }

    /*
     * Allocate the arrays of TetReferencePairs, and set them to NULL.
     */
    for (f = 0; f < 4; f++)
        for (v = 0; v < 4; v++)
        {
            length = data[old_tet->index].side_array_length[f][v];
            array = NEW_ARRAY(length, TetReferencePair);
            data[old_tet->index].side_array[f][v] = array;
            for (i = 0; i < length; i++)
            {
                array[i].outer.tet = NULL;
                array[i].outer.gluing = 0;
                array[i].inner.tet = NULL;
                array[i].inner.gluing = 0;
            }
        }
    }

    return data;
}

static void free_subdivision_data(
    SubdivisionData *data,
    int num_old_tetrahedra)
{
    int i;
    FaceIndex f;
    VertexIndex v;

    for (i = 0; i < num_old_tetrahedra; i++)
        for (f = 0; f < 4; f++)
            for (v = 0; v < 4; v++)
                my_free(data[i].side_array[f][v]);

    my_free(data);
}

static void copy_cusps(
    Triangulation *manifold,
    Triangulation *subdivision)
{
    Cusp *cusp;

    if (subdivision->num_cusps != 0
        || subdivision->cusp_list_begin.next != &subdivision->cusp_list_end)
        uFatalError("copy_cusps", "normal_surface_splitting");

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)
    {
        cusp->matching_cusp = NEW_STRUCT(Cusp);
        initialize_cusp(cusp->matching_cusp);

        if (cusp->is_complete != TRUE)
            uFatalError("copy_cusps", "normal_surface_splitting");

        cusp->matching_cusp->topology = cusp->topology;
        cusp->matching_cusp->is_complete = TRUE;
        cusp->matching_cusp->m = 0;
        cusp->matching_cusp->l = 0;
        cusp->matching_cusp->index = cusp->index;
        cusp->matching_cusp->is_finite = FALSE;

        INSERT_BEFORE(cusp->matching_cusp, &subdivision->cusp_list_end);
        subdivision->num_cusps++;
    }
}

```

```

    if (subdivision->num_cusps != manifold->num_cusps)
        uFatalError("copy_cusps", "normal_surface_splitting");
}

static void subdivide_old_tetrahedron(
    Tetrahedron    *old_tet,
    Triangulation   *subdivision,
    SubdivisionData *tet_data,
    Cusp            *cusp_at_split)
{
    VertexIndex v;
    int         i;

    /*
     * Subdivide the triangular prisms, if any.
     */
    for (v = 0; v < 4; v++)
        for (i = 0; i < old_tet->num_triangles[v]; i++)
            subdivide_triangular_prism(old_tet, v, i, subdivision, tet_data, cusp_at_split) ✓
    ;

    /*
     * Does this old_tet contain squares?
     */
    if (old_tet->num_squares == 0)
    {
        /*
         * There are no squares.
         * Subdivide the central tetrahedron.
         */
        subdivide_central_tetrahedron(old_tet, subdivision, tet_data, cusp_at_split);
    }
    else
    {
        /*
         * There are squares.
         * Subdivide the two pillows, as well as any square prisms.
         */
        subdivide_pillow(old_tet, old_tet->parallel_edge, subdivision, tet_data, cusp_at_split); ✓
        subdivide_pillow(old_tet, 5 - old_tet->parallel_edge, subdivision, tet_data, cusp_at_split); ✓
        for (i = 0; i < old_tet->num_squares - 1; i++)
            subdivide_square_prism(old_tet, i, subdivision, tet_data, cusp_at_split);
    }
}

```

```

static void subdivide_triangular_prism(
    Tetrahedron    *old_tet,
    VertexIndex     old_v,
    int             index,
    Triangulation   *subdivision,
    SubdivisionData *tet_data,
    Cusp            *cusp_at_split)
{
    Tetrahedron    *tet[2];
    int             i;
    PeripheralCurve c;
    Orientation     h;
    FaceIndex       f;
    VertexIndex     v;

    /*
     * Subdivide the triangular prism into two Tetrahedra, and add
     * the Tetrahedra to the subdivision.
     * tet[0] is closer to the cusp, while tet[1] is farther from it.
     *
     * Important note: tet[0] and tet[1] inherit VertexIndices from
     * old_tet in the natural way. tet[0] inherits the same orientation
     * as the old_tet, which, together with a similar convention in

```

```

    * subdivide_central_tetrahedron() and subdivide_pillow(),
    * ensures that the peripheral curves from the original cusps will
    * match correctly across the right_ and left_handed sheets.
    */

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

        /*
        * Is vertex old_v at a cusp of the original manifold?
        */
        if (index == 0 && i == 0)
        {
            tet[i]->cusp[old_v] = old_tet->cusp[old_v]->matching_cusp;

            for (c = 0; c < 2; c++)          /* M or L */
                for (h = 0; h < 2; h++)      /* right_handed or left_handed */
                    for (f = 0; f < 4; f++) /* which side of the triangle */
                        tet[i]->curve[c][h][old_v][f] = old_tet->curve[c][h][old_v][f];
        }
        else
            tet[i]->cusp[old_v] = cusp_at_split;

        for (v = 0; v < 4; v++)
            if (v != old_v)
                tet[i]->cusp[v] = NULL;

        INSERT_BEFORE(tet[i], &subdivision->tet_list_end);
        subdivision->num_tetrahedra++;
    }

    /*
    * Glue tet[0] and tet[1] to each other.
    */
    for (i = 0; i < 2; i++)
    {
        tet[i]->neighbor[old_v] = tet[!i];
        tet[i]->gluing[old_v] = IDENTITY_PERMUTATION;
    }

    /*
    * Fill in the appropriate fields of the SubdivisionData.
    */
    for (f = 0; f < 4; f++)
        if (f != old_v)
        {
            tet_data->side_array[f][old_v][index].outer.tet = tet[0];
            tet_data->side_array[f][old_v][index].inner.tet = tet[1];

            tet_data->side_array[f][old_v][index].outer.gluing
            = tet_data->side_array[f][old_v][index].inner.gluing
            = CREATE_PERMUTATION(
                old_v, old_v,
                f, f,
                remaining_face[old_v][f], remaining_face[f][old_v],
                remaining_face[f][old_v], remaining_face[old_v][f]);
        }

    /*
    * tet[0] has the correct orientation, but tet[1] does not.
    */
    tet[0]->has_correct_orientation = TRUE;
    tet[1]->has_correct_orientation = FALSE;
}

static void subdivide_central_tetrahedron(
    Tetrahedron      *old_tet,
    Triangulation     *subdivision,
    SubdivisionData   *tet_data,
    Cusp              *cusp_at_split)
{

```

```

Tetrahedron      *vertex_tet[4],
                  *middle_tet[4],
                  *face_tet[4];
int              i;
PeripheralCurve   c;
Orientation       h;
FaceIndex        f;
VertexIndex      v;

/*
 * Think of the central tetrahedron as the union of an octahedron
 * (whose vertices are the finite vertices at the "midpoints" of
 * the central tetrahedron's sides) plus four tetrahedra, one at
 * each of the central tetrahedron's four ideal vertices.
 */

/*
 * There are two obvious ways to subdivide the octahedron into
 * tetrahedra. One could divide it into four tetrahedra meeting
 * along an axis, or one could divide it into eight tetrahedra by
 * coning to its center. The first approach uses less memory, but
 * the second approach is simpler to program. For now I have
 * adopted the second approach. If memory usage gets to be a problem,
 * this function could be rewritten using the first approach.
 * The required changes would be local to this function; no other
 * functions would be affected.
 */

/*
 * The 12 tetrahedra in the subdivision are grouped as follows:
 *
 *     vertex_tet[4]   records the four Tetrahedra incident to the
 *                      ideal vertices,
 *     middle_tet[4]   records the four Tetrahedra which share
 *                      faces with the vertex_tet[], and
 *     face_tet[4]     records the remaining four Tetrahedra.
 *
 * All Tetrahedra are numbered and indexed in the obvious natural way.
 *
 * Important note: Each of the vertex_tets inherits VertexIndices
 * from old_tet in the natural way. In particular, they inherit
 * old_tet's orientation. Together with a similar convention
 * in subdivide_triangular_prism() and subdivide_pillow(), this
 * ensures that the peripheral curves of the original cusps will
 * match correctly across the right_ and left_handed sheets.
 */

for (i = 0; i < 4; i++)
{
    vertex_tet[i]   = NEW_STRUCT(Tetrahedron);
    middle_tet[i]   = NEW_STRUCT(Tetrahedron);
    face_tet[i]     = NEW_STRUCT(Tetrahedron);

    initialize_tetrahedron(vertex_tet[i]);
    initialize_tetrahedron(middle_tet[i]);
    initialize_tetrahedron(face_tet[i]);

    for (v = 0; v < 4; v++)
    {
        if (v != i)
            vertex_tet[i]->cusp[v] = NULL;
        middle_tet[i]->cusp[v] = NULL;
        face_tet[i]->cusp[v]   = NULL;
    }

    /*
     * Is vertex i of vertex_tet[i] at a cusp of the original manifold?
     */
    if (old_tet->num_triangles[i] == 0)
    {
        vertex_tet[i]->cusp[i] = old_tet->cusp[i]->matching_cusp;

        for (c = 0; c < 2; c++)          /* M or L */
            for (h = 0; h < 2; h++)      /* right_handed or left_handed */

```

```

        for (f = 0; f < 4; f++) /* which side of the triangle */
            vertex_tet[i]->curve[c][h][i][f] = old_tet->curve[c][h][i][f];
    }
    else
        vertex_tet[i]->cusp[i] = cusp_at_split;

    INSERT_BEFORE(vertex_tet[i], &subdivision->tet_list_end);
    INSERT_BEFORE(middle_tet[i], &subdivision->tet_list_end);
    INSERT_BEFORE(face_tet[i], &subdivision->tet_list_end);
    subdivision->num_tetrahedra += 3;
}

/*
 * Glue the vertex_tets to the middle_tets.
 */
for (i = 0; i < 4; i++)
{
    vertex_tet[i]->neighbor[i] = middle_tet[i];
    middle_tet[i]->neighbor[i] = vertex_tet[i];

    vertex_tet[i]->gluing[i] = IDENTITY_PERMUTATION;
    middle_tet[i]->gluing[i] = IDENTITY_PERMUTATION;
}

/*
 * Glue the middle_tets to the face_tets.
 */
for (i = 0; i < 4; i++)
    for (f = 0; f < 4; f++)
        if (f != i)
        {
            middle_tet[i]->neighbor[f] = face_tet[f];
            face_tet[f]->neighbor[i] = middle_tet[i];

            middle_tet[i]->gluing[f]
            = face_tet[f]->gluing[i]
            = CREATE_PERMUTATION(
                i,f,
                f,i,
                remaining_face[i][f], remaining_face[f][i],
                remaining_face[f][i], remaining_face[i][f]);
        }

/*
 * Fill in the appropriate fields of the SubdivisionData.
 */
for (f = 0; f < 4; f++)
{
    tet_data->central[f][f].tet      = face_tet[f];
    tet_data->central[f][f].gluing  = IDENTITY_PERMUTATION;

    for (v = 0; v < 4; v++)
        if (v != f)
        {
            tet_data->central[f][v].tet      = vertex_tet[v];
            tet_data->central[f][v].gluing  = CREATE_PERMUTATION(
                v,v,
                f,f,
                remaining_face[v][f], remaining_face[f][v],
                remaining_face[f][v], remaining_face[v][f]);
        }
}

/*
 * The vertex_tets and face_tets inherit the correct orientation,
 * but the middle_tets do not.
 */
for (i = 0; i < 4; i++)
{
    vertex_tet[i]->has_correct_orientation = TRUE;
    middle_tet[i]->has_correct_orientation = FALSE;
    face_tet[i]->has_correct_orientation = TRUE;
}
}

```

```

static void subdivide_pillow(
    Tetrahedron    *old_tet,
    EdgeIndex      defining_edge,
    Triangulation   *subdivision,
    SubdivisionData *tet_data,
    Cusp            *cusp_at_split)
{
    Tetrahedron    *vertex_tet[2],
                  *octa_tet[4];
    VertexIndex     v[4];
    int             i;
    PeripheralCurve c;
    Orientation      h;
    FaceIndex        f;
    VertexIndex      vv;
    int             ind[2];

    /*
     * Think of the pillow as the union of an octahedron (whose vertices
     * are the five finite vertices at the "midpoints" of the pillow's
     * sides) plus two tetrahedra, one at each of the pillow's triangular
     * ideal vertices. The octahedron gets further subdivided into
     * four tetrahedra, meeting along the obvious axis of symmetry.
     *
     * The six tetrahedra in the subdivision are grouped as follows:
     *
     *     vertex_tet[2]    records the two Tetrahedra incident to the
     *                       ideal vertices,
     *     octa_tet[2]      records the four Tetrahedra which comprise
     *                       the octahedron.
     *
     * It will be helpful to draw yourself a picture as you read through
     * this documentation. Draw the triangular pillow, representing the
     * ideal vertices in the usual way (cf. the top of this file) as two
     * blue triangles and one red square. Then, with dotted lines, draw
     * the remaining portion of the original large tetrahedra, i.e. the
     * part that lies beyond the red square. Having the rest of the
     * large tetrahedron visible will make it easier to keep track of
     * the VertexIndices. The defining_edge is the edge of the pillow
     * parallel to the red square. Call its two endpoints (the blue
     * triangles) v[0] and v[1]. The faces opposite v[0] and v[1] are
     * f[0] and f[1], respectively. They are the bigonal faces of the
     * pillow, although when extended to the original large tetrahedron
     * they are triangles. Relative to the large tetrahedron's right_handed
     * Orientation, we may apply remaining_face[][] to locate faces
     * f[2] and f[3]. The ideal vertices opposite faces f[2] and f[3]
     * are v[2] and v[3], respectively. They lie at the far side of the
     * original large tetrahedron, and are NOT contained within the pillow
     * itself. (Error check: the vertices v[0],v[1],v[3] should go
     * counterclockwise around face f[2] in your drawing, and the vertices
     * v[1],v[0],v[2] should go counterclockwise around face f[3].)
     *
     * vertex_tet[0] is the one incident to vertex v[0], and similarly
     * vertex_tet[1] is the one incident to vertex v[1].
     */

    v[0] = one_vertex_at_edge[defining_edge];
    v[1] = other_vertex_at_edge[defining_edge];
    v[2] = remaining_face[v[0]][v[1]];
    v[3] = remaining_face[v[1]][v[0]];

    /*
     * Important note: Each of the vertex_tets inherits VertexIndices
     * from old_tet in the natural way. Together with a similar convention
     * in subdivide_triangular_prism() and subdivide_central_tetrahedron(),
     * this ensures that the peripheral curves of the original cusps will
     * match correctly across the right_ and left_handed sheets.
     */

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

```

```

    initialize_tetrahedron(vertex_tet[i]);

    for (vv = 0; vv < 4; vv++)
        if (vv != v[i])
            vertex_tet[i]->cusp[vv] = NULL;

    /*
     * Is vertex v[i] of vertex_tet[i] at a cusp
     * of the original manifold?
     */
    if (old_tet->num_triangles[v[i]] == 0)
    {
        vertex_tet[i]->cusp[v[i]] = old_tet->cusp[v[i]]->matching_cusp;

        for (c = 0; c < 2; c++)          /* M or L */
            for (h = 0; h < 2; h++)      /* right_handed or left_handed */
                for (f = 0; f < 4; f++) /* which side of the triangle */
                    vertex_tet[i]->curve[c][h][v[i]][f] = old_tet->curve[c][h][v[i]][f]
    }
    else
        vertex_tet[i]->cusp[v[i]] = cusp_at_split;

    INSERT_BEFORE(vertex_tet[i], &subdivision->tet_list_end);
    subdivision->num_tetrahedra++;
}

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

    /*
     * As explained in the paragraph immediately below,
     * the ideal vertex at the red square has VertexIndex 1.
     * The other vertices of the octa_tets are finite vertices.
     */
    for (vv = 0; vv < 4; vv++)
    {
        if (vv == 1)
            octa_tet[i]->cusp[vv] = cusp_at_split;
        else
            octa_tet[i]->cusp[vv] = NULL;
    }

    INSERT_BEFORE(octa_tet[i], &subdivision->tet_list_end);
    subdivision->num_tetrahedra++;
}

/*
 * Glue the four octa_tets to each other. They are numbered
 * in "west-to-east" order, with
 *
 *     octa_tet[0] at face f[1] of the original large tetrahedron,
 *     octa_tet[1] at face f[3] of the original large tetrahedron,
 *     octa_tet[2] at face f[0] of the original large tetrahedron,
 *     octa_tet[3] at face f[2] of the original large tetrahedron.
 *
 * Error check: Your drawing should show that vertex_tet[0] borders
 * octa_tet[0], and vertex_tet[1] borders octa_tet[3].
 *
 * Assign VertexIndices to each octa_tet so that vertex 0 is at the
 * "north pole" (i.e. on the defining_edge), vertex 1 is at the
 * "south pole" (i.e. at the red square), vertex 2 is to the "west"
 * and vertex 3 is to the "east".
 */
for (i = 0; i < 4; i++)
{
    octa_tet[i]->neighbor[2] = octa_tet[(i+1)%4];
    octa_tet[i]->gluing[2] = CREATE_PERMUTATION( 0,0, 1,1, 2,3, 3,2 );

    octa_tet[i]->neighbor[3] = octa_tet[(i+3)%4];
    octa_tet[i]->gluing[3] = CREATE_PERMUTATION( 0,0, 1,1, 2,3, 3,2 );
}

```

```

/*
 * Glue the vertex_tets to the octa_tets.
 */

vertex_tet[0]->neighbor[v[0]] = octa_tet[0];
vertex_tet[0]->gluing[v[0]] = CREATE_PERMUTATION( v[0],1, v[1],0, v[2],3, v[3],2 );
octa_tet[0]->neighbor[1] = vertex_tet[0];
octa_tet[0]->gluing[1] = CREATE_PERMUTATION( 0,v[1], 1,v[0], 2,v[3], 3,v[2] );

vertex_tet[1]->neighbor[v[1]] = octa_tet[2];
vertex_tet[1]->gluing[v[1]] = CREATE_PERMUTATION( v[1],1, v[0],0, v[3],3, v[2],2 );
octa_tet[2]->neighbor[1] = vertex_tet[1];
octa_tet[2]->gluing[1] = CREATE_PERMUTATION( 0,v[0], 1,v[1], 2,v[2], 3,v[3] );

/*
 * Fill in the appropriate fields of the SubdivisionData.
 */

tet_data->central[v[2]][v[2]].tet = octa_tet[3];
tet_data->central[v[2]][v[2]].gluing = CREATE_PERMUTATION( v[3],0, v[0],2, v[1],3, v[2],1 );

tet_data->central[v[2]][v[0]].tet = vertex_tet[0];
tet_data->central[v[2]][v[0]].gluing = CREATE_PERMUTATION( v[0],v[0], v[1],v[3], v[2],v[2], v[3],v[1] );

tet_data->central[v[2]][v[1]].tet = vertex_tet[1];
tet_data->central[v[2]][v[1]].gluing = CREATE_PERMUTATION( v[0],v[3], v[1],v[1], v[2],v[2], v[3],v[0] );

tet_data->central[v[2]][v[3]].tet = octa_tet[3];
tet_data->central[v[2]][v[3]].gluing = CREATE_PERMUTATION( v[0],2, v[1],3, v[2],0, v[3],1 );

tet_data->central[v[3]][v[3]].tet = octa_tet[1];
tet_data->central[v[3]][v[3]].gluing = CREATE_PERMUTATION( v[2],0, v[1],2, v[0],3, v[3],1 );

tet_data->central[v[3]][v[1]].tet = vertex_tet[1];
tet_data->central[v[3]][v[1]].gluing = CREATE_PERMUTATION( v[1],v[1], v[0],v[2], v[3],v[3], v[2],v[0] );

tet_data->central[v[3]][v[0]].tet = vertex_tet[0];
tet_data->central[v[3]][v[0]].gluing = CREATE_PERMUTATION( v[1],v[2], v[0],v[0], v[3],v[3], v[2],v[1] );

tet_data->central[v[3]][v[2]].tet = octa_tet[1];
tet_data->central[v[3]][v[2]].gluing = CREATE_PERMUTATION( v[1],2, v[0],3, v[3],0, v[2],1 );

ind[0] = old_tet->num_triangles[v[0]];
ind[1] = old_tet->num_triangles[v[1]];

tet_data->side_array[v[1]][v[0]][ind[0]].outer.tet = vertex_tet[0];
tet_data->side_array[v[1]][v[0]][ind[0]].outer.gluing =
    CREATE_PERMUTATION( v[0],v[0], v[2],v[3], v[3],v[2], v[1],v[1] );

tet_data->side_array[v[1]][v[0]][ind[0]].inner.tet = octa_tet[0];
tet_data->side_array[v[1]][v[0]][ind[0]].inner.gluing =
    CREATE_PERMUTATION( v[0],1, v[2],2, v[3],3, v[1],0 );

tet_data->side_array[v[0]][v[1]][ind[1]].outer.tet = vertex_tet[1];
tet_data->side_array[v[0]][v[1]][ind[1]].outer.gluing =
    CREATE_PERMUTATION( v[1],v[1], v[3],v[2], v[2],v[3], v[0],v[0] );

tet_data->side_array[v[0]][v[1]][ind[1]].inner.tet = octa_tet[2];
tet_data->side_array[v[0]][v[1]][ind[1]].inner.gluing =
    CREATE_PERMUTATION( v[1],1, v[3],2, v[2],3, v[0],0 );

/*
 * All tetrahedra inherit the correct orientation.
 */
for (i = 0; i < 2; i++)

```



```

    vertex_tet[i]->has_correct_orientation = TRUE;
for (i = 0; i < 4; i++)
    octa_tet[i] ->has_correct_orientation = TRUE;
}

static void subdivide_square_prism(
    Tetrahedron    *old_tet,
    int            index,
    Triangulation  *subdivision,
    SubdivisionData *tet_data,
    Cusp           *cusp_at_split)
{
    Tetrahedron    *tet[4];
    int            i;
    FaceIndex       f[4];
    VertexIndex     v[4];
    int            ind[4];

    /*
     * Subdivide the square prism (which is combinatorially an octahedron,
     * after taking into account the subdivision of its boundary as defined
     * at the top of this file) into four Tetrahedra arranged symmetrically
     * about a common "vertical" axis.  Assign VertexIndices to each
     * Tetrahedron so that the ideal vertex at the octahedron's
     * "north pole" has index 0, the ideal vertex at its "south pole"
     * has index 1, the "western" finite vertex on the "equator" has
     * index 2, and the "eastern" finite vertex on the "equator" has
     * index 3.
     */

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

        tet[i]->cusp[0] = cusp_at_split;
        tet[i]->cusp[1] = cusp_at_split;
        tet[i]->cusp[2] = NULL;
        tet[i]->cusp[3] = NULL;

        INSERT_BEFORE(tet[i], &subdivision->tet_list_end);
        subdivision->num_tetrahedra++;
    }

    /*
     * Glue the four Tetrahedra to each other.
     * The array tet[] lists the Tetrahedra in west-to-east order.
     */
    for (i = 0; i < 4; i++)
    {
        tet[i]->neighbor[2] = tet[(i+1)%4];
        tet[i]->gluing[2]   = CREATE_PERMUTATION( 0,0, 1,1, 2,3, 3,2);

        tet[i]->neighbor[3] = tet[(i+3)%4];
        tet[i]->gluing[3]   = CREATE_PERMUTATION( 0,0, 1,1, 2,3, 3,2);
    }

    /*
     * Fill in the appropriate fields of the SubdivisionData.
     *
     * Before reading this code, make yourself a sketch of a tetrahedron
     * containing two red squares at its center.  Label the "parallel edge",
     * and then label the various faces and vertices as you read the code.
     */

    f[0] = one_face_at_edge[old_tet->parallel_edge];
    f[1] = other_face_at_edge[old_tet->parallel_edge];
    f[2] = remaining_face[f[0]][f[1]];
    f[3] = remaining_face[f[1]][f[0]];

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

```

```

    v[3] = f[2];

    ind[0] = (old_tet->num_squares - 1) - index + old_tet->num_triangles[v[0]];
    ind[1] = (old_tet->num_squares - 1) - index + old_tet->num_triangles[v[1]];
    ind[2] = 1 + index + old_tet->num_triangles[v[2]];
    ind[3] = 1 + index + old_tet->num_triangles[v[3]];

    tet_data->side_array[f[0]][v[0]][ind[0]].outer.tet
= tet_data->side_array[f[0]][v[0]][ind[0]].inner.tet
= tet[0];

    tet_data->side_array[f[3]][v[3]][ind[3]].outer.tet
= tet_data->side_array[f[3]][v[3]][ind[3]].inner.tet
= tet[1];

    tet_data->side_array[f[1]][v[1]][ind[1]].outer.tet
= tet_data->side_array[f[1]][v[1]][ind[1]].inner.tet
= tet[2];

    tet_data->side_array[f[2]][v[2]][ind[2]].outer.tet
= tet_data->side_array[f[2]][v[2]][ind[2]].inner.tet
= tet[3];

    tet_data->side_array[f[0]][v[0]][ind[0]].outer.gluing
= CREATE_PERMUTATION( v[0],1, v[3],2, v[2],3, v[1],0 );
    tet_data->side_array[f[0]][v[0]][ind[0]].inner.gluing
= CREATE_PERMUTATION( v[0],0, v[3],2, v[2],3, v[1],1 );

    tet_data->side_array[f[3]][v[3]][ind[3]].outer.gluing
= CREATE_PERMUTATION( v[3],0, v[1],2, v[0],3, v[2],1 );
    tet_data->side_array[f[3]][v[3]][ind[3]].inner.gluing
= CREATE_PERMUTATION( v[3],1, v[1],2, v[0],3, v[2],0 );

    tet_data->side_array[f[1]][v[1]][ind[1]].outer.gluing
= CREATE_PERMUTATION( v[1],1, v[2],2, v[3],3, v[0],0 );
    tet_data->side_array[f[1]][v[1]][ind[1]].inner.gluing
= CREATE_PERMUTATION( v[1],0, v[2],2, v[3],3, v[0],1 );

    tet_data->side_array[f[2]][v[2]][ind[2]].outer.gluing
= CREATE_PERMUTATION( v[2],0, v[0],2, v[1],3, v[3],1 );
    tet_data->side_array[f[2]][v[2]][ind[2]].inner.gluing
= CREATE_PERMUTATION( v[2],1, v[0],2, v[1],3, v[3],0 );

    /*
     * The tetrahedra inherit the correct orientation.
     */
    for (i = 0; i < 4; i++)
        tet[i]->has_correct_orientation = TRUE;
}

static void glue_external_faces(
    Triangulation *manifold,
    SubdivisionData *data)
{
    Tetrahedron *old_tet,
                *old_nbr,
                *tet,
                *nbr;
    Permutation gluing,
                tet_perm,
                nbr_perm;
    FaceIndex f,
              ff;
    VertexIndex v,
                vv;
    int i;

    /*
     * For each original large tetrahedron...
     */
    for (old_tet = manifold->tet_list_begin.next;
         old_tet != &manifold->tet_list_end;
         old_tet = old_tet->next)

```

```

/*
 * ...consider the subdivision of each face.
 */
for (f = 0; f < 4; f++)
{
    /*
     * Find our neighboring original large tetrahedron.
     */
    old_nbr = old_tet->neighbor[f];
    gluing = old_tet->gluing[f];
    ff = EVALUATE(gluing,f);

    /*
     * The gluing mapping each small tetrahedron at this
     * face to its mate at the neighboring face will be the
     * composition of
     *
     * (1) the mapping taking the actual VertexIndices of the
     *     small tetrahedron to the standard VertexIndices on
     *     the face (cf. the top of this file),
     * (2) old_tet->gluing[f], and
     * (3) the mapping taking the standard VertexIndices on the
     *     neighboring face to actual VertexIndices of the
     *     neighboring small tetrahedron.
     *
     * The function compute_external_gluing() computes
     * this composition.
     */

    /*
     * There will always be a central piece.
     */
    for (v = 0; v < 4; v++)
    {
        vv = EVALUATE(gluing,v);

        tet = data[old_tet->index].central[f][v].tet;
        tet_perm = data[old_tet->index].central[f][v].gluing;
        nbr = data[old_nbr->index].central[ff][vv].tet;
        nbr_perm = data[old_nbr->index].central[ff][vv].gluing;

        tet->neighbor[EVALUATE(tet_perm,f)] = nbr;
        tet->gluing[EVALUATE(tet_perm,f)] = compute_external_gluing(tet_perm,
gluing, nbr_perm);
    }

    /*
     * There may also be (subdivided) bigons at the incident vertices.
     */
    for (v = 0; v < 4; v++)
        if (v != f)
            for (i = 0; i < data[old_tet->index].side_array_length[f][v]; i++)
            {
                vv = EVALUATE(gluing,v);

                tet = data[old_tet->index].side_array[f][v][i].outer.tet;
                tet_perm = data[old_tet->index].side_array[f][v][i].outer.
gluing;
                nbr = data[old_nbr->index].side_array[ff][vv][i].outer.tet;
                nbr_perm = data[old_nbr->index].side_array[ff][vv][i].outer.
gluing;

                tet->neighbor[EVALUATE(tet_perm,f)] = nbr;
                tet->gluing[EVALUATE(tet_perm,f)] = compute_external_gluing
(tet_perm, gluing, nbr_perm);

                tet = data[old_tet->index].side_array[f][v][i].inner.tet;
                tet_perm = data[old_tet->index].side_array[f][v][i].inner.
gluing;
                nbr = data[old_nbr->index].side_array[ff][vv][i].inner.tet;
                nbr_perm = data[old_nbr->index].side_array[ff][vv][i].inner.
gluing;

                tet->neighbor[EVALUATE(tet_perm,f)] = nbr;

```

```

        tet->gluing [EVALUATE(tet_perm,f)] = compute_external_gluing
        (tet_perm, gluing, nbr_perm);
    }
}

static Permutation compute_external_gluing(
    Permutation perm0_inverse,
    Permutation perm1,
    Permutation perm2)
{
    Permutation result;

    result = inverse_permutation[perm0_inverse];
    result = compose_permutations(perm1, result); /* right-to-left evaluation */
    result = compose_permutations(perm2, result); /* right-to-left evaluation */

    return result;
}

static void distinguish_cusps(
    Triangulation *subdivision,
    Cusp *new_cusps[2])
{
    IdealVertexReference *ivr_stack;
    int ivr_stack_size;
    Tetrahedron *tet,
    *nbr;
    VertexIndex v,
    nbr_v;
    FaceIndex f;

    /*
     * The calling program has split along a 2-sided torus or Klein bottle.
     * By default it has set all the pointers tet->cusps[v] to point to
     * new_cusps[0]. The purpose of the present function is to leave
     * those belonging to one cusp set to new_cusps[0], while changing
     * those belong to the other cusp to new_cusps[1].
     */

    /*
     * Allocate space for a stack of IdealVertexReferences for which
     * tet->cusps[v] has been changed from new_cusps[0] to new_cusps[1],
     * but whose neighbors have not yet been examined.
     */
    ivr_stack = NEW_ARRAY(4 * subdivision->num_tetrahedra, IdealVertexReference);
    ivr_stack_size = 0;

    /*
     * Find a reference to new_cusps[0], change it to new_cusps[1],
     * and put it on the ivr_stack.
     */
    for (tet = subdivision->tet_list_begin.next;
        tet != &subdivision->tet_list_end && ivr_stack_size == 0;
        tet = tet->next)
        for (v = 0; v < 4; v++)
            if (tet->cusps[v] == new_cusps[0])
            {
                tet->cusps[v] = new_cusps[1];
                ivr_stack[0].tet = tet;
                ivr_stack[0].v = v;
                ivr_stack_size = 1;
                break;
            }
    if (ivr_stack_size == 0)
        uFatalError("distinguish_cusps", "normal_surface_splitting");

    /*
     * While the stack isn't empty...
     */
    while (ivr_stack_size > 0)
    {

```

```

    /*
     * Pull an IdealVertexReference off the top of the stack.
     */
    --ivr_stack_size;
    tet = ivr_stack[ivr_stack_size].tet;
    v = ivr_stack[ivr_stack_size].v;

    /*
     * Look at each of its neighbors.
     */
    for (f = 0; f < 4; f++)
        if (f != v)
        {
            nbr = tet->neighbor[f];
            nbr_v = EVALUATE(tet->gluing[f], v);

            /*
             * If the neighbor hasn't already been switched
             * to new_cusps[1], switch it and put it on the ivr_stack.
             */
            if (nbr->cusp[nbr_v] != new_cusps[1])
            {
                nbr->cusp[nbr_v] = new_cusps[1];
                ivr_stack[ivr_stack_size].tet = nbr;
                ivr_stack[ivr_stack_size].v = nbr_v;
                ivr_stack_size++;
            }
        }
    }

    /*
     * Free the stack.
     */
    my_free(ivr_stack);
}

static void separate_connected_components(
    Triangulation *subdivision,
    Triangulation *pieces[2])
{
    Tetrahedron *tet,
                **tet_stack,
                *nbr;
    int tet_stack_size,
        *which_piece,
        cusp_index,
        *old_to_new_index,
        i;
    FaceIndex f;
    VertexIndex v;
    Cusp *cusp;

    /*
     * Initialize pieces[0].
     */
    pieces[0] = NEW_STRUCT(Triangulation);
    initialize_triangulation(pieces[0]);

    /*
     * Move an arbitrary Tetrahedron from the subdivision to pieces[0],
     * and then recursively move its neighbors.
     */

    /*
     * Use the flag to record which Tetrahedra have been moved.
     */
    for (tet = subdivision->tet_list_begin.next;
         tet != &subdivision->tet_list_end;
         tet = tet->next)
        tet->flag = FALSE;

    /*
     * Initialize a stack. The stack will contain pointers to

```

```

    * Tetrahedra which have been moved, but whose neighbors have
    * not yet been examined.
    */
tet_stack = NEW_ARRAY(subdivision->num_tetrahedra, Tetrahedron *);
tet_stack_size = 0;

/*
 * Transfer an arbitrary Tetrahedron, and
 * put a pointer to it on the stack.
 */
if (subdivision->tet_list_begin.next == &subdivision->tet_list_end)
    uFatalError("separate_connected_components", "normal_surface_splitting");
tet_stack[0] = subdivision->tet_list_begin.next;
tet_stack_size = 1;
tet_stack[0]->flag = TRUE;
REMOVE_NODE(tet_stack[0]);
INSERT_BEFORE(tet_stack[0], &pieces[0]->tet_list_end);
subdivision->num_tetrahedra--;
pieces[0]->num_tetrahedra++;

/*
 * While the stack is nonempty...
 */
while (tet_stack_size > 0)
{
    /*
     * Pull a Tetrahedron pointer off the stack.
     */
    tet = tet_stack[--tet_stack_size];

    /*
     * Examine its neighbors.
     */
    for (f = 0; f < 4; f++)
    {
        nbr = tet->neighbor[f];

        /*
         * If the neighbor hasn't been transferred,
         * transfer it and put a pointer to it on the stack.
         */
        if (nbr->flag == FALSE)
        {
            tet_stack[tet_stack_size++] = nbr;
            nbr->flag = TRUE;
            REMOVE_NODE(nbr);
            INSERT_BEFORE(nbr, &pieces[0]->tet_list_end);
            subdivision->num_tetrahedra--;
            pieces[0]->num_tetrahedra++;
        }
    }
}

/*
 * Free the stack.
 */
my_free(tet_stack);

/*
 * A quick error check.
 */
if ((subdivision->num_tetrahedra == 0) != (subdivision->tet_list_begin.next == &
subdivision->tet_list_end))
    uFatalError("separate_connected_components", "normal_surface_splitting");

/*
 * If any Tetrahedra remain in subdivision, transfer them to pieces[1].
 */
if (subdivision->num_tetrahedra > 0)
{
    pieces[1] = NEW_STRUCT(Triangulation);
    initialize_triangulation(pieces[1]);

    pieces[1]->tet_list_begin.next          = subdivision->tet_list_begin.next;

```

```

    pieces[1]->tet_list_begin.next->prev      = &pieces[1]->tet_list_begin;
    pieces[1]->tet_list_end.prev              = subdivision->tet_list_end.prev;
    pieces[1]->tet_list_end.prev->next        = &pieces[1]->tet_list_end;
    subdivision->tet_list_begin.next          = &subdivision->tet_list_end;
    subdivision->tet_list_end.prev            = &subdivision->tet_list_begin;

    pieces[1]->num_tetrahedra = subdivision->num_tetrahedra;
    subdivision->num_tetrahedra = 0;
}
else
    pieces[1] = NULL;

/*
 * Another quick error check.
 */
if (subdivision->cuspl_list_begin.next == &subdivision->cuspl_list_end)
    uFatalError("separate_connected_components", "normal_surface_splitting");

/*
 * If there's only one piece, transfer all the Cusps to it.
 * If there are two pieces we have to be a little more careful,
 * and transfer each Cusp to the correct piece.
 */
if (pieces[1] == NULL)
{
    pieces[0]->cuspl_list_begin.next          = subdivision->cuspl_list_begin.next;
    pieces[0]->cuspl_list_begin.next->prev    = &pieces[0]->cuspl_list_begin;
    pieces[0]->cuspl_list_end.prev            = subdivision->cuspl_list_end.prev;
    pieces[0]->cuspl_list_end.prev->next      = &pieces[0]->cuspl_list_end;
    subdivision->cuspl_list_begin.next        = &subdivision->cuspl_list_end;
    subdivision->cuspl_list_end.prev          = &subdivision->cuspl_list_begin;

    pieces[0]->num_cusps = subdivision->num_cusps;
    subdivision->num_cusps = 0;
}
else
{
    /*
     * We need to figure out which Cusp goes with which piece.
     * Set up an array to record this information:  which_piece[i]
     * will equal 0 (resp. 1) when the Cusp of index i belongs to
     * pieces[0] (resp. pieces[1]).
     */
    which_piece = NEW_ARRAY(subdivision->num_cusps, int);

    /*
     * We don't really need to initialize which_piece[],
     * but it allows error checking.
     */
    for (i = 0; i < subdivision->num_cusps; i++)
        which_piece[i] = -1;

    /*
     * Use the tet->cuspl[] fields to deduce which Cusp goes where.
     */
    for (i = 0; i < 2; i++)
        for (tet = pieces[i]->tet_list_begin.next;
             tet != &pieces[i]->tet_list_end;
             tet = tet->next)
            for (v = 0; v < 4; v++)
            {
                if (tet->cuspl[v] == NULL) /* skip finite vertices */
                    continue;

                cuspl_index = tet->cuspl[v]->index;

                if (cuspl_index < 0 || cuspl_index >= subdivision->num_cusps)
                    uFatalError("separate_connected_components",
"normal_surface_splitting");

                switch (which_piece[cuspl_index])
                {
                    case -1:
                        which_piece[cuspl_index] = i;

```

```

        break;
    case 0:
        if (i != 0)
            uFatalError("separate_connected_components",
"normal_surface_splitting");
        break;
    case 1:
        if (i != 1)
            uFatalError("separate_connected_components",
"normal_surface_splitting");
        break;
    }
}

for (i = 0; i < subdivision->num_cusps; i++)
    if (which_piece[i] == -1)
        uFatalError("separate_connected_components", "normal_surface_splitting");

/*
 * The Cusps should be numbered consecutively within each piece,
 * yet retain the same relative order as they had in subdivision.
 * The array old_to_new_index[] translates an old Cusp index
 * (in subdivision) to a new Cusp index (in pieces[which_piece[i]]).
 * As a side effect, this code sets pieces[]->num_cusps.
 */
old_to_new_index = NEW_ARRAY(subdivision->num_cusps, int);
pieces[0]->num_cusps = 0;
pieces[1]->num_cusps = 0;
for (i = 0; i < subdivision->num_cusps; i++)
    old_to_new_index[i] = pieces[which_piece[i]]->num_cusps++;

/*
 * Transfer the Cusps from subdivision to pieces[0] and pieces[1].
 */
while (subdivision->cusp_list_begin.next != &subdivision->cusp_list_end)
{
    cusp = subdivision->cusp_list_begin.next;
    REMOVE_NODE(cusp);
    INSERT_BEFORE(cusp, &pieces[which_piece[cusp->index]]->cusp_list_end);
    cusp->index = old_to_new_index[cusp->index];
}
subdivision->num_cusps = 0;

my_free(which_piece);
my_free(old_to_new_index);
}

static Tetrahedron *find_correctly_oriented_tet(
    Triangulation *manifold)
{
    Tetrahedron *tet;

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

        if (tet->has_correct_orientation == TRUE)

            return tet;

    uFatalError("find_correctly_oriented_tet", "normal_surface_splitting");
    return NULL; /* provide a return value to keep the compiler happy */
}

```