

```

/*
 * drilling.c
 *
 * This file contains the function
 *
 *      Triangulation *drill_cusp(   Triangulation      *old_manifold,
 *                                  DualOneSkeletonCurve *curve_to_drill,
 *                                  char                  *new_name);
 *
 * which the kernel provides to the UI to drill out a simple closed curve
 * in a manifold's dual 1-skeleton. Please see dual_one_skeleton_curve.h
 * for a description of the DualOneSkeletonCurve. drill_cusp() accepts
 * as input the original n-cusp manifold and a DualOneSkeletonCurve to
 * be drilled. If the drilling curve is not boundary parallel
 * (i.e. if the curve is not parabolic) drill_cusp() returns a pointer
 * to the resulting (n+1)-cusp manifold. If the drilling curve is
 * boundary parallel, drill_cusp() returns NULL. (If the drilling curve
 * is boundary parallel, but in a nonobvious way, then drill_cusp() will
 * succeed and return a pointer to a nonhyperbolic manifold.) In
 * practice, I recommend that the UI not even offer the user the option
 * of drilling out parabolics.
 *
 * The original manifold is not altered.
 *
 * The meridian on the new cusp is chosen so that meridional Dehn
 * filling (i.e. (1,0) Dehn filling) restores the original manifold.
 *
 * We assume the Tetrahedra in old_manifold are numbered, e.g.
 * by the kernel function number_the_tetrahedra(), and that the
 * curve_to_drill conforms to this numbering.
 *
 * Note that an arbitrary curve in the dual 1-skeleton may or may not
 * be knotted. By definition (my definition, anyhow) a simple closed curve
 * in a hyperbolic 3-manifold is unknotted iff it is isotopic to the unique
 * geodesic in its homotopy class.
 *
 * The remainder of this comment describes drill_cusp()'s algorithm.
 *
 * Tetrahedra which don't intersect the drilling curve are left alone.
 *
 * Each tetrahedron which does intersect the drilling curve is subdivided
 * into four small Tetrahedra by coning to the center. (I recommend you
 * draw sketches as you read this.) The drilling curve intersects the
 * interior of exactly two of the four small Tetrahedra -- throw those two
 * away and keep the two which don't intersect it. Each of the two
 * remaining small Tetrahedra has an ideal vertex at the center of
 * the original large Tetrahedron -- visualize them as truncated ideal
 * vertices. (Time to revise your sketch!) It may be helpful to color
 * one truncated vertex red and the other blue. The red and blue
 * triangles (of the truncated vertices) together determine a tiny
 * tetrahedron at the center of the original Tetrahedron. Draw its
 * entire 1-skeleton in black. Now -- this is the crucial observation --
 * the tiny tetrahedron is a scaled down version of the big Tetrahedron.
 * So, thinking of the manifold globally now, glue together pairs
 * of exposed faces of small Tetrahedra in the natural way.
 * Just as the set of large Tetrahedra intersecting the drilling
 * curve forms a solid torus in the manifold (possibly with
 * self-intersections on its boundary, which don't concern us), the
 * tiny tetrahedra composed of truncated ideal vertices piece together
 * to form a tiny torus. The tiny torus is just a scaled down version
 * of the large one.
 *
 * I claim that the Triangulation we have just created will be
 * homeomorphic to the original Triangulation with the drilling
 * curve removed iff the drilling curve is not "obviously" parallel
 * to the boundary (in a moment the meaning of this statement will
 * be made more precise). Consider each large Tetrahedron intersecting
 * the drilling curve in the original Triangulation, and its subdivision
 * into four small Tetrahedra, two of which get thrown away. Each
 * vertex cross section of such a large Tetrahedron is a triangle,
 * which is subdivided into three smaller triangles by the small
 * Tetrahedra. At two of the large Tetrahedron's vertices, two
 * small triangles will be retained, and one will have been discarded

```

```

* (when we discarded two of the four small Tetrahedra),
* while at the the remaining two vertices, one small triangle will
* be retained and two will have been discarded. Now look at an entire
* torus or Klein bottle boundary component. If the drilling
* curve is blatantly parallel to this boundary component, then
* when you glue the faces of the small Tetrahedra as explained
* in the preceeding paragraph, the image of the drilling curve
* on the boundary gets pinched off (draw yourself a picture).
* This increases the Euler characteristic of the boundary, and
* the function check_Euler_characteristic_of_boundary() flags the error.
* If, on the other hand, the drilling curve follows this
* boundary component only along isolated intervals, then it's
* easy to see that when the faces of the small Tetrahedra are
* glued as defined above, the holes are filled in correctly (again,
* draw yourself a picture), and the topology of the manifold is preserved.
*/

#include "kernel.h"

/*
* If you are not familiar with SnapPea's "Extra" field in
* the Tetrahedron data structure, please see the explanation
* preceding the Extra typedef in kernel_typedefs.h.
*
* drill_cusp() attaches an Extra field to each old Tetrahedron
* to keep track of the new Tetrahedra associated with it.
*/

struct extra
{
    /*
    * Does the drilling curve pass through this Tetrahedron?
    */
    Boolean drilling_curve_intersects_tet;

    /*
    * Does the drilling curve pass through the given face?
    */
    Boolean drilling_curve_intersects_face[4];

    /*
    * If the drilling curve does not pass through this Tetrahedron,
    * the new Triangulation will contain a single Tetrahedron
    * corresponding to this one. extra->big_tet will point to it.
    */
    Tetrahedron *big_tet;

    /*
    * If the drilling curve does pass through this Tetrahedron, it
    * will cross exactly two faces. There will be a small Tetrahedron
    * associated with each face which does not intersect the
    * drilling curve. Two of the following pointers will point to
    * those Tetrahedra; the other two will be NULL.
    */
    Tetrahedron *small_tet[4];

    /*
    * index[] says which of the two small_tet's are actually in use.
    */
    FaceIndex index[2];
};

/*
* The functions which find the meridian and longitude on the new
* Cusp use a MeridionalAnnulus data structure.
*
* Recall from above that the each old Tetrahedron intersecting
* the drilling curve contributes two small new Tetrahedra to
* the Triangulation of the new manifold. These two small new
* Tetrahedra contribute a degenerate meridional annulus to the
* new boundary component. In terms of the above imagery, the
* degenerate meridional annulus consists of the small red triangle,
* the small blue triangle, and the black line segment connecting

```

```

* the far vertices. The annulus is degenerate because the black
* segment is only a segment, but nevertheless it should be clear
* how these annuli piece together to form the new boundary component.
*
* The MeridionalAnnulus consists of a PositionedTet, plus a
* current_position field.
*
* I'm sorry to have to do this to you, but please imagine the
* PositionedTet with the bottom_face down ("on the table"),
* the near_face away from you, the left_face towards you and on
* the right, and the right_face towards you and on the left.
* I.e. rotate it a half turn about the vertical axis, relative
* to the way you usually imagine it.
*
* We make the convention that the drilling curve passes through
* the left_ and right_faces, while new small Tetrahedra are located
* at the near_ and bottom_faces. (The reason for the nonstandard
* positioning described in the previous paragraph is that it gives
* a good view of the truncated vertices of the new Cusp.)
* On the new Cusp, "northward" is from the vertex on the bottom_face
* towards the vertex on the near_face (i.e. "up"), while "eastward"
* is from the side near the right_face towards the side near
* left_face (i.e. to the right). The meridian on the new Cusp will
* run north, while the longitude runs east (this corresponds to the
* usual convention).
*
* The current_position field says where we are in the PositionedTet.
* If current_position is
*     0   we're on the degenerate edge,
*     1   we're on the truncated ideal vertex sitting over the bottom_face,
*     2   we're on the truncated ideal vertex sitting over the near_face.
*/

typedef struct
{
    PositionedTet    ptet;
    int              current_position;
} MeridionalAnnulus;

typedef int DirectionToTravel;
enum
{
    to_the_east,
    to_the_west
};

static void attach_extra(Triangulation *manifold);
static void free_extra(Triangulation *manifold);
static void mark_drilling_curve(Triangulation *old_manifold, DualOneSkeletonCurve *
    curve_to_drill);
static void set_up_new_triangulation(Triangulation *old_manifold, Triangulation **
    new_manifold, char *new_name);
static void allocate_new_tetrahedra(Triangulation *old_manifold, Triangulation *
    new_manifold);
static void set_neighbors_and_gluings(Triangulation *old_manifold);
static void set_big_tet_neighbors_and_gluings(Tetrahedron *old_tet);
static void set_small_tet_neighbors_and_gluings(Tetrahedron *old_tet);
static void set_cusps(Triangulation *old_manifold, Triangulation *new_manifold);
static void copy_old_peripheral_curves(Triangulation *old_manifold, Triangulation *
    new_manifold);
static void create_new_peripheral_curves(Triangulation *old_manifold, Triangulation *
    new_manifold);
static void create_new_meridian(PositionedTet ptet);
static void create_new_longitude(PositionedTet ptet, CuspTopology *cusp_topology);
static void move_sideways(MeridionalAnnulus *ma, DirectionToTravel direction);
static void transfer_CS(Triangulation *old_manifold, Triangulation *new_manifold);

Triangulation *drill_cusp(
    Triangulation      *old_manifold,
    DualOneSkeletonCurve *curve_to_drill,
    char                *new_name)

```

```

{
    Triangulation    *new_manifold;

    /*
     * Attach an Extra field to each old Tetrahedron to keep
     * track of the new Tetrahedra associated with it.
     */
    attach_extra(old_manifold);

    /*
     * Determine the exact path of the drilling curve in
     * the dual 1-skeleton of the old_manifold.
     */
    mark_drilling_curve(old_manifold, curve_to_drill);

    /*
     * Set up the global data for the new_manifold.
     */
    set_up_new_triangulation(old_manifold, &new_manifold, new_name);

    /*
     * Allocate space for the new Tetrahedra, and associate them
     * with the corresponding old Tetrahedra.
     */
    allocate_new_tetrahedra(old_manifold, new_manifold);

    /*
     * Set the neighbors and gluings.
     */
    set_neighbors_and_gluings(old_manifold);

    /*
     * Make copies of the old cusps, and create a new cusp.
     */
    set_cusps(old_manifold, new_manifold);

    /*
     * Add the bells and whistles.
     * Note that it isn't necessary to call orient(). The new_manifold
     * will automatically be oriented iff the old_manifold was oriented.
     */
    create_edge_classes(new_manifold);
    orient_edge_classes(new_manifold);

    /*
     * The algorithm will have failed
     * iff the Euler characteristic of the boundary is positive
     * iff the drilling curve was parallel to the (original) boundary.
     * Cf. the discussion at the top of this file.
     */
    if (check_Euler_characteristic_of_boundary(new_manifold) == func_failed)
    {
        free_triangulation(new_manifold);
        free_extra(old_manifold);
        return NULL;
    }

    /*
     * Copy the peripheral curves from the old_manifold for the
     * preexisting cusps, and make a new set of peripheral curves
     * for the brand new cusp. It's essential that we do
     * the peripheral curves AFTER checking the Euler characteristic
     * of the boundary.
     */
    copy_old_peripheral_curves(old_manifold, new_manifold);
    create_new_peripheral_curves(old_manifold, new_manifold);

    /*
     * Free the Extra fields.
     */
    free_extra(old_manifold);

    /*
     * Simplify the triangulation. (Usually it's pretty good

```

```

    * to begin with, but EdgeClasses of order 2 or 3 may
    * occasionally appear.)
    *
    * basic_simplification() will call tidy_peripheral_curves().
    * Otherwise we'd do it here.
    */
basic_simplification(new_manifold);

/*
 * If the old_manifold had a hyperbolic structure,
 * try to find one for the new_manifold as well.
 */
if (old_manifold->solution_type[complete] != not_attempted)
{
    find_complete_hyperbolic_structure(new_manifold);
    do_Dehn_filling(new_manifold);

    /*
     * If the old_manifold had a known Chern-Simons invariant,
     * try to transfer it to the new_manifold.
     */
    transfer_CS(old_manifold, new_manifold);
}

return new_manifold;
}

static void attach_extra(
    Triangulation *manifold)
{
    Tetrahedron *tet;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * Make sure no other routine is using the "extra"
         * field in the Tetrahedron data structure.
         */
        if (tet->extra != NULL)
            uFatalError("attach_extra", "drilling");

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

static void free_extra(
    Triangulation *manifold)
{
    Tetrahedron *tet;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * Free the struct extra.
         */
        my_free(tet->extra);

        /*
         * Set the extra pointer to NULL to let other
         * modules know we're done with it.
         */
        tet->extra = NULL;
    }
}
```

```

static void mark_drilling_curve(
    Triangulation    *old_manifold,
    DualOneSkeletonCurve *curve_to_drill)
{
    Tetrahedron *tet;
    int i;

    for (tet = old_manifold->tet_list_begin.next;
         tet != &old_manifold->tet_list_end;
         tet = tet->next)
    {
        tet->extra->drilling_curve_intersects_tet = FALSE;

        for (i = 0; i < 4; i++)
        {
            tet->extra->drilling_curve_intersects_face[i]
                = curve_to_drill->tet_intersection[tet->index][i];

            if (tet->extra->drilling_curve_intersects_face[i] == TRUE)
                tet->extra->drilling_curve_intersects_tet = TRUE;
        }
    }
}

static void set_up_new_triangulation(
    Triangulation    *old_manifold,
    Triangulation    **new_manifold,
    char             *new_name)
{
    /*
     * Allocate memory for the new_manifold.
     */
    *new_manifold = NEW_STRUCT(Triangulation);

    /*
     * Call the generic initialization routine.
     */
    initialize_triangulation(*new_manifold);

    /*
     * Copy in the name requested by the UI.
     */
    (*new_manifold)->name = NEW_ARRAY(strlen(new_name) + 1, char);
    strcpy((*new_manifold)->name, new_name);

    /*
     * The triangulation algorithm guarantees that the new_manifold
     * will be oriented iff the old one was.
     */
    (*new_manifold)->orientability = old_manifold->orientability;

    /*
     * For now we set the number of cusps equal to the number in the
     * old manifold. We'll increment the appropriate numbers once
     * we discover whether the new cusp is orientable or nonorientable.
     */
    (*new_manifold)->num_cusps      = old_manifold->num_cusps;
    (*new_manifold)->num_or_cusps   = old_manifold->num_or_cusps;
    (*new_manifold)->num_nonor_cusps = old_manifold->num_nonor_cusps;
}

static void allocate_new_tetrahedra(
    Triangulation    *old_manifold,
    Triangulation    *new_manifold)
{
    Tetrahedron *tet;
    int i,
        count;

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

```

```

    tet = tet->next)

    if (tet->extra->drilling_curve_intersects_tet == FALSE)
    {
        tet->extra->big_tet = NEW_STRUCT(Tetrahedron);
        initialize_tetrahedron(tet->extra->big_tet);
        INSERT_BEFORE(tet->extra->big_tet, &new_manifold->tet_list_end);
        new_manifold->num_tetrahedra++;

        for (i = 0; i < 4; i++)
            tet->extra->small_tet[i] = NULL;
    }
    else
    {
        tet->extra->big_tet = NULL;

        count = 0;
        for (i = 0; i < 4; i++)
            if (tet->extra->drilling_curve_intersects_face[i] == FALSE)
            {
                tet->extra->small_tet[i] = NEW_STRUCT(Tetrahedron);
                initialize_tetrahedron(tet->extra->small_tet[i]);
                INSERT_BEFORE(tet->extra->small_tet[i], &new_manifold->tet_list_end);
                new_manifold->num_tetrahedra++;
                tet->extra->index[count++] = i;
            }
            else
                tet->extra->small_tet[i] = NULL;
    }
}

static void set_neighbors_and_gluings(
    Triangulation *old_manifold)
{
    Tetrahedron *old_tet;

    /*
     * The VertexIndices of the new Tetrahedra are inherited from
     * those of the old Tetrahedra in the obvious, canonical way.
     * Thus, except for the "internal" gluings between two small
     * new Tetrahedra associated with the same old Tetrahedron,
     * all the new gluings are the same as the corresponding old
     * gluings.
     *
     * We don't explicitly set inverses -- they'll be taken care
     * of when the for(;;) loop comes around to them.
     */

    for (old_tet = old_manifold->tet_list_begin.next;
         old_tet != &old_manifold->tet_list_end;
         old_tet = old_tet->next)

        if (old_tet->extra->drilling_curve_intersects_tet == FALSE)
            set_big_tet_neighbors_and_gluings(old_tet);
        else
            set_small_tet_neighbors_and_gluings(old_tet);
}

static void set_big_tet_neighbors_and_gluings(
    Tetrahedron *old_tet)
{
    int i;

    /*
     * Set the neighbors and gluings for the four faces
     * of the new big Tetrahedron.
     */

    for (i = 0; i < 4; i++)
    {
        /*
         * Check whether the neighbor is a big tet or a small tet,

```

```

        * and set the neighbor field accordingly.
    */
    old_tet->extra->big_tet->neighbor[i] =
        (old_tet->neighbor[i]->extra->drilling_curve_intersects_tet == FALSE) ?
        old_tet->neighbor[i]->extra->big_tet :
        old_tet->neighbor[i]->extra->small_tet[EVALUATE(old_tet->gluing[i], i)];

    /*
    * The gluing is independent of whether the neighbor is a
    * big tet or a small tet.
    */
    old_tet->extra->big_tet->gluing[i] = old_tet->gluing[i];
}

static void set_small_tet_neighbors_and_gluings(
    Tetrahedron *old_tet)
{
    int            i,
                  j;
    FaceIndex      f0,
                  f1,
                  f2,
                  f3;
    PositionedTet  ptet;

    /*
    * Set the neighbors and gluings for the two new small Tetrahedra
    * associated with old_tet.
    */

    /*
    * First take care of the faces of the new Tetrahedra which coincide
    * with faces of old_tet.
    */

    for (i = 0; i < 2; i++)
    {
        /*
        * Let f0 be the actual index of the small Tetrahedron
        * under consideration.
        */
        f0 = old_tet->extra->index[i];

        /*
        * Check whether the neighbor is a big tet or a small tet,
        * and set the neighbor field accordingly.
        */
        old_tet->extra->small_tet[f0]->neighbor[f0] =
            (old_tet->neighbor[f0]->extra->drilling_curve_intersects_tet == FALSE) ?
            old_tet->neighbor[f0]->extra->big_tet :
            old_tet->neighbor[f0]->extra->small_tet[EVALUATE(old_tet->gluing[f0], f0)];

        /*
        * The gluing is independent of whether the neighbor is a
        * big tet or a small tet.
        */
        old_tet->extra->small_tet[f0]->gluing[f0] = old_tet->gluing[f0];
    }

    /*
    * Glue the two small Tetrahedra to each other.
    *
    * Let f0 and f1 be the indices of the two small Tetrahedra under
    * consideration, and f2 and f3 be the unused indices.
    */

    f0 = old_tet->extra->index[0];
    f1 = old_tet->extra->index[1];
    f2 = remaining_face[f0][f1];
    f3 = remaining_face[f1][f0];

    old_tet->extra->small_tet[f0]->neighbor[f1]

```



```

    = old_tet->extra->small_tet[f1];
old_tet->extra->small_tet[f1]->neighbor[f0]
    = old_tet->extra->small_tet[f0];

old_tet->extra->small_tet[f0]->gluing[f1]
    = old_tet->extra->small_tet[f1]->gluing[f0]
    = CREATE_PERMUTATION(f0, f1, f1, f0, f2, f2, f3, f3);

/*
 * Now set the neighbors and gluings for the two remaining
 * faces of each of the two small Tetrahedra by swinging around
 * the appropriate edge of the old_tet until a noncollapsed
 * small Tetrahedron is found. (To see why this is correct,
 * think of the extra small Tetrahedra collapsing to triangles,
 * as described in the documentation at the top of this file.)
 */

for (i = 0; i < 2; i++)
{
    /*
     * Let f0 be the index of the small Tetrahedron under
     * consideration, and f1 be the index of the other small
     * Tetrahedron.
     */

    f0 = old_tet->extra->index[i];
    f1 = old_tet->extra->index[!i];

    for (j = 0; j < 2; j++)
    {
        /*
         * Let f2 be the face whose neighbor and gluing we'll
         * determine, and f3 be the face left over.
         */

        f2 = (j ? remaining_face[f0][f1] : remaining_face[f1][f0]);
        f3 = (j ? remaining_face[f1][f0] : remaining_face[f0][f1]);

        /*
         * Set up a PositionedTet which we'll rotate around until
         * we find a match for the face under consideration.
         */

        ptet.tet          = old_tet;
        ptet.near_face     = f0;
        ptet.left_face     = f2;
        ptet.right_face    = f1;
        ptet.bottom_face   = f3;
        ptet.orientation   = (f2 == remaining_face[f0][f1]) ?
                               right_handed :
                               left_handed;

        /*
         * Veer_left() as long as necessary until we find a small
         * Tetrahedron to glue to.
         */

        do
            veer_left(&ptet);
        while (ptet.tet->extra->drilling_curve_intersects_face[ptet.left_face] == TRUE) ✓

    }

    /*
     * Set the neighbor and gluing fields.
     */

    old_tet->extra->small_tet[f0]->neighbor[f2]
        = ptet.tet->extra->small_tet[ptet.left_face];
    old_tet->extra->small_tet[f0]->gluing[f2]
        = CREATE_PERMUTATION(
            f0, ptet.left_face,
            f1, ptet.right_face,
            f2, ptet.near_face,
            f3, ptet.bottom_face);

```

```

    }
}

static void set_cusps(
    Triangulation *old_manifold,
    Triangulation *new_manifold)
{
    int i,
        j;
    FaceIndex f;
    Cusp **new_cusp_addresses,
        *old_cusp,
        *new_cusp,
        *brand_new_cusp;
    Tetrahedron *old_tet;

    /*
     * Make copies of the old Cusps.
     * Record the addresses of the new Cusps in an array for
     * later convenience.
     */

    new_cusp_addresses = NEW_ARRAY(old_manifold->num_cusps, Cusp *);

    for (old_cusp = old_manifold->cusp_list_begin.next;
        old_cusp != &old_manifold->cusp_list_end;
        old_cusp = old_cusp->next)
    {
        new_cusp = NEW_STRUCT(Cusp);
        *new_cusp = *old_cusp;
        new_cusp_addresses[new_cusp->index] = new_cusp;
        INSERT_BEFORE(new_cusp, &new_manifold->cusp_list_end);
    }

    /*
     * Create a brand new Cusp for the drilling curve.
     */

    brand_new_cusp = NEW_STRUCT(Cusp);
    initialize_cusp(brand_new_cusp);
    INSERT_BEFORE(brand_new_cusp, &new_manifold->cusp_list_end);
    brand_new_cusp->index = old_manifold->num_cusps;

    /*
     * Set the new_tet->cusp[] fields.
     */

    for (old_tet = old_manifold->tet_list_begin.next;
        old_tet != &old_manifold->tet_list_end;
        old_tet = old_tet->next)

        if (old_tet->extra->drilling_curve_intersects_tet == FALSE)
            for (i = 0; i < 4; i++)
                old_tet->extra->big_tet->cusp[i]
                    = new_cusp_addresses[old_tet->cusp[i]->index];
        else
            for (i = 0; i < 2; i++)
            {
                f = old_tet->extra->index[i];
                for (j = 0; j < 4; j++)
                    old_tet->extra->small_tet[f]->cusp[j]
                        = (j == f) ?
                            brand_new_cusp :
                            new_cusp_addresses[old_tet->cusp[j]->index];
            }

    /*
     * Free the array used to hold the new Cusp addresses.
     */

    my_free(new_cusp_addresses);
}

```

```

static void copy_old_peripheral_curves(
    Triangulation *old_manifold,
    Triangulation *new_manifold)
{
    Tetrahedron *old_tet,
                *new_tet;
    int i,
        ii,
        j,
        k,
        l;
    FaceIndex f;
    EdgeClass *new_edge;
    VertexIndex v0,
                v1;
    PositionedTet ptet0,
                 ptet;
    int in_hand[2][2];

    /*
     * First copy the peripheral curves onto the edges of the
     * new boundary triangulation which correspond exactly with
     * edges of the old boundary triangulation.
     */

    for (old_tet = old_manifold->tet_list_begin.next;
         old_tet != &old_manifold->tet_list_end;
         old_tet = old_tet->next)

        if (old_tet->extra->drilling_curve_intersects_tet == FALSE)
            for (i = 0; i < 2; i++)
                for (j = 0; j < 2; j++)
                    for (k = 0; k < 4; k++)
                        for (l = 0; l < 4; l++)
                            old_tet->extra->big_tet->curve[i][j][k][l]
                                = old_tet->curve[i][j][k][l];
        else
            for (i = 0; i < 2; i++)
            {
                f = old_tet->extra->index[i];
                for (j = 0; j < 4; j++)
                    if (j != f)
                        for (k = 0; k < 2; k++)
                            for (l = 0; l < 2; l++)
                                old_tet->extra->small_tet[f]->curve[k][l][j][f]
                                    = old_tet->curve[k][l][j][f];
            }

    /*
     * At this point it's helpful to draw the old boundary triangulation
     * at a given cusp, with the new boundary triangulation superimposed
     * in green. At ideal vertices of old tetrahedra not intersecting
     * the drilling curve, the new green triangle will coincide with the
     * old plain triangle. At ideal vertices of old tetrahedra which do
     * intersect the drilling curve, either a single small green triangle
     * will occupy a third of the old plain triangle, or two small green
     * triangles will occupy two-thirds of the old plain triangle, depending
     * on which vertex you're at. Consider the gaps where the new green
     * triangles don't cover the old plain ones. If the drilling curve
     * had been boundary parallel, the gaps would form a topological
     * annulus, but by the time this function is called the program will
     * have already checked the Euler characteristic of the boundary, so
     * we know this can't occur. Instead, the gaps form topological disks
     * on the boundary. When the new small Tetrahedra are glued to each
     * other, the small green triangles on the boundary come together to
     * form a disk, thereby closing the gaps. In my illustration, this
     * disk looks like a green pizza. The peripheral curves will be
     * correct around the circumferences of such pizzas. The purpose of
     * the remainder of this function is to adjust them in the interior
     * of the pizza, i.e. between the slices. We'll walk around
     * the circumference of each pizza, hooking up incoming strands
     * on one part of the circumference to outgoing strands on another.
    */

```

```

    */

/*
 * Check each new EdgeClass which connects an old cusp
 * to the brand new cusp.
 */

for (new_edge = new_manifold->edge_list_begin.next;
     new_edge != &new_manifold->edge_list_end;
     new_edge = new_edge->next)
{
    /*
     * Does new_edge have one endpoint on an old cusp and one
     * on the brand new cusp? If not, skip this EdgeClass.
     */
    new_tet = new_edge->incident_tet;
    v0 = one_vertex_at_edge[new_edge->incident_edge_index];
    v1 = other_vertex_at_edge[new_edge->incident_edge_index];
    if ((new_tet->cusp[v0]->index < old_manifold->num_cusps)
        == (new_tet->cusp[v1]->index < old_manifold->num_cusps))
        continue;

    /*
     * Set up a PositionedTet, which we'll rotate about the
     * center of the green pizza described above.
     */

    ptet0.tet = new_tet;
    ptet0.right_face = (new_tet->cusp[v0]->index < old_manifold->num_cusps) ? v1 : v0;
    ptet0.bottom_face = (new_tet->cusp[v0]->index < old_manifold->num_cusps) ? v0 : v1;
    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;

    /*
     * In_hand will record how many strands of each curve
     * (meridian, longitude) on each sheet (right_handed,
     * left_handed) we'll be carrying with us "in hand"
     * as we progress to the next slice of pizza. Initialize
     * it to zero.
     */

    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            in_hand[i][j] = 0;

    /*
     * Circumnavigate the pizza, correctly setting the peripheral
     * curves between slices.
     */

    ptet = ptet0;
    do
    {
        /*
         * Update the value of in_hand to account for the strands which
         * just entered or left through the circumference of the pizza.
         */
        for (i = 0; i < 2; i++)
        {
            ii = (ptet.orientation == ptet0.orientation) ? i : !i;
            for (j = 0; j < 2; j++)
                in_hand[j][ii] += ptet.tet->curve[j][ii][ptet.bottom_face][ptet.
right_face];
        }

        /*
         * Adjust the leading edge of this slice.
         */
        for (i = 0; i < 2; i++)
        {
            ii = (ptet.orientation == ptet0.orientation) ? i : !i;

```

```

        for (j = 0; j < 2; j++)
            ptet.tet->curve[j][ii][ptet.bottom_face][ptet.left_face] = - in_hand[j]
[i];
    }

    /*
     * Move on to the next slice.
     */
    veer_left(&ptet);

    /*
     * Adjust the trailing edge of this slice.
     */
    for (i = 0; i < 2; i++)
    {
        ii = (ptet.orientation == ptet0.orientation) ? i : !i;
        for (j = 0; j < 2; j++)
            ptet.tet->curve[j][ii][ptet.bottom_face][ptet.near_face] = in_hand[j]
[i];
    }

    /*
     * Quit if we're back to the slice we started on.
     * Otherwise, continue with the loop.
     */

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

    /*
     * Check that all incoming and outgoing strands
     * did in fact cancel out.
     */
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            if (in_hand[i][j] != 0)
                uFatalError("copy_old_peripheral_curves", "drilling");
    }
}

static void create_new_peripheral_curves(
    Triangulation *old_manifold,
    Triangulation *new_manifold)
{
    Tetrahedron *old_tet;
    PositionedTet ptet;
    CuspTopology cusp_topology;

    /*
     * We want to be sure to be sure to get the relative
     * orientation of the meridian and longitude correct
     * (cf. the orientation convention at the top of
     * peripheral_curves.c, which coincides with the usual
     * orientation convention for meridians and longitudes
     * on knot complements). Here we find an old Tetrahedron
     * which intersects the drilling curve, orient it, and
     * pass it to functions which actually find the meridian
     * and the longitude. We use the PositionedTet structure
     * as a bookkeeping device.
     */

    for (old_tet = old_manifold->tet_list_begin.next;
         old_tet != &old_manifold->tet_list_end;
         old_tet = old_tet->next)

        if (old_tet->extra->drilling_curve_intersects_tet == TRUE)
        {
            /*
             * Set up the PositionedTet.
             */

            ptet.tet = old_tet;
            ptet.near_face = old_tet->extra->index[0];
            ptet.bottom_face = old_tet->extra->index[1];

```

```

    ptet.left_face      = remaining_face[ptet.bottom_face][ptet.near_face];
    ptet.right_face     = remaining_face[ptet.near_face][ptet.bottom_face];
    ptet.orientation    = right_handed;

    /*
     * Compute the new peripheral curves.
     */

    create_new_meridian (ptet);
    create_new_longitude(ptet, &cuspid_topology);

    /*
     * Record the topology of the new cusp.
     */

    if (cuspid_topology == torus_cuspid)
    {
        old_tet->extra->small_tet[old_tet->extra->index[0]]
            ->cuspid[old_tet->extra->index[0]]->topology = torus_cuspid;
        new_manifold->num_or_cusps++;
    }
    else
    {
        old_tet->extra->small_tet[old_tet->extra->index[0]]
            ->cuspid[old_tet->extra->index[0]]->topology = Klein_cuspid;
        new_manifold->num_nonor_cusps++;
    }
    new_manifold->num_cusps++;

    return;
}

/*
 * We should have returned from within the above loop.
 */
uFatalError("create_new_peripheral_curves", "drilling");
}

static void create_new_meridian(
    PositionedTet ptet)
{
    MeridionalAnnulus ma0,
    ma;
    int steps_northward,
    steps_eastward;

    /*
     * Note that the meridian is set using "+=" rather than just "=".
     * This is because the algorithm proceeds in the universal cover,
     * and the curve might pass over itself in the manifold itself.
     * (Such precautions aren't necessary for the longitude.)
     */

    ma0.ptet = ptet;
    ma0.current_position = 1;

    ma = ma0;

    steps_northward = 0;
    steps_eastward = 0;

    /*
     * Move three steps northward, moving east as necessary.
     */

    while (steps_northward < 3)
    {
        /*
         * Move east until current_position == 1.
         * (If this were not possible, the manifold would have
         * already failed check_Euler_characteristic_of_boundary(),
         * and we wouldn't be at this point.)
         */

```

```

while (ma.current_position != 1)
{
    if (ma.current_position == 2)
        ma.ptet.tet->extra->small_tet[ma.ptet.near_face]->curve[M]
            [ma.ptet.orientation][ma.ptet.near_face][ma.ptet.left_face]
            += -1;

    move_sideways(&ma, to_the_east);

    if (ma.current_position == 1)
        ma.ptet.tet->extra->small_tet[ma.ptet.bottom_face]->curve[M]
            [ma.ptet.orientation][ma.ptet.bottom_face][ma.ptet.right_face]
            += 1;
    if (ma.current_position == 2)
        ma.ptet.tet->extra->small_tet[ma.ptet.near_face]->curve[M]
            [ma.ptet.orientation][ma.ptet.near_face][ma.ptet.right_face]
            += 1;

    steps_eastward++;
}

/*
 * Move north one step.
 */

ma.ptet.tet->extra->small_tet[ma.ptet.bottom_face]->curve[M]
    [ma.ptet.orientation][ma.ptet.bottom_face][ma.ptet.near_face]
    += -1;
ma.ptet.tet->extra->small_tet[ma.ptet.near_face]->curve[M]
    [ma.ptet.orientation][ma.ptet.near_face][ma.ptet.bottom_face]
    += 1;

ma.current_position = 2;

steps_northward++;
}

/*
 * Take as many steps back to the west as we took to the east.
 */

while (--steps_eastward >= 0)
{
    if (ma.current_position == 1)
        ma.ptet.tet->extra->small_tet[ma.ptet.bottom_face]->curve[M]
            [ma.ptet.orientation][ma.ptet.bottom_face][ma.ptet.right_face]
            += -1;
    if (ma.current_position == 2)
        ma.ptet.tet->extra->small_tet[ma.ptet.near_face]->curve[M]
            [ma.ptet.orientation][ma.ptet.near_face][ma.ptet.right_face]
            += -1;

    move_sideways(&ma, to_the_west);

    if (ma.current_position == 1)
        ma.ptet.tet->extra->small_tet[ma.ptet.bottom_face]->curve[M]
            [ma.ptet.orientation][ma.ptet.bottom_face][ma.ptet.left_face]
            += 1;
    if (ma.current_position == 2)
        ma.ptet.tet->extra->small_tet[ma.ptet.near_face]->curve[M]
            [ma.ptet.orientation][ma.ptet.near_face][ma.ptet.left_face]
            += 1;
}

/*
 * Just in case . . .
 */

if ( ! same_positioned_tet(&ma.ptet, &ma0.ptet)
    || ma.current_position != ma0.current_position)
    uFatalError("create_new_meridian", "drilling");
}

```

```

static void create_new_longitude(
    PositionedTet    ptet,
    CuspTopology     *cusp_topology)
{
    MeridionalAnnulus    ma0,
                        ma;
    Boolean               enters_north,
                        leaves_north;

    /*
     * We make the following convention in passing the longitude
     * from one MeridionalAnnulus to the next.  If the Meridional
     * Annuli are not aligned, then the longitude passes across
     * the unique edge which is degenerate for neither of them.
     * Otherwise it passes across the more northerly of the two
     * nondegenerate edges.
     *
     * Technical note: create_new_longitude() doesn't actually
     * use the degenerate_index field of the MeridionalAnnulus.
     * That field is included for the convenience of create_new_meridian(),
     * with which create_new_longitude() shares the move_sideways() function.
     */

    /*
     * We assume the Cusp is orientable unless we discover otherwise.
     */
    *cusp_topology = torus_cusp;

    ma0.ptet      = ptet;
    ma0.current_position = 0;    /* will be ignored */

    ma = ma0;

do
{
    /*
     * See where the longitude enters on the west.
     */

    if (ma.ptet.tet->neighbor[ma.ptet.right_face]
        ->extra->drilling_curve_intersects_face
        [EVALUATE(    ma.ptet.tet->gluing[ma.ptet.right_face],
                     ma.ptet.near_face)
        ] == FALSE)
    {
        enters_north = TRUE;
        ma.ptet.tet->extra->small_tet[ma.ptet.near_face]->curve[L]
            [ma.ptet.orientation][ma.ptet.near_face][ma.ptet.right_face]
            = 1;
    }
    else
    {
        enters_north = FALSE;
        ma.ptet.tet->extra->small_tet[ma.ptet.bottom_face]->curve[L]
            [ma.ptet.orientation][ma.ptet.bottom_face][ma.ptet.right_face]
            = 1;
    }

    /*
     * See where the longitude leaves on the east.
     */

    if (ma.ptet.tet->neighbor[ma.ptet.left_face]
        ->extra->drilling_curve_intersects_face
        [EVALUATE(    ma.ptet.tet->gluing[ma.ptet.left_face],
                     ma.ptet.near_face)
        ] == FALSE)
    {
        leaves_north = TRUE;
        ma.ptet.tet->extra->small_tet[ma.ptet.near_face]->curve[L]
            [ma.ptet.orientation][ma.ptet.near_face][ma.ptet.left_face]
            = -1;
    }
}

```



```

else
{
    leaves_north = FALSE;
    ma.ptet.tet->extra->small_tet[ma.ptet.bottom_face]->curve[L]
        [ma.ptet.orientation][ma.ptet.bottom_face][ma.ptet.left_face]
        = -1;
}

/*
 * Do we cross the edge between the northern and southern triangles?
 */

if (enters_north != leaves_north)
{
    ma.ptet.tet->extra->small_tet[ma.ptet.near_face]->curve[L]
        [ma.ptet.orientation][ma.ptet.near_face][ma.ptet.bottom_face]
        = (enters_north == TRUE) ? -1 : 1;
    ma.ptet.tet->extra->small_tet[ma.ptet.bottom_face]->curve[L]
        [ma.ptet.orientation][ma.ptet.bottom_face][ma.ptet.near_face]
        = (enters_north == TRUE) ? 1 : -1;
}

/*
 * Move on to the next MeridionalAnnulus.
 */

move_sideways(&ma, to_the_east);

/*
 * If we've come around to the original Tetrahedron, but
 * with the opposite orientation, then we know the cusp is
 * nonorientable.
 */

if (
    ma.ptet.tet == ma0.ptet.tet
    && ma.ptet.orientation != ma0.ptet.orientation)
    *cusp_topology = Klein_cusp;
} while ( ! same_positioned_tet(&ma.ptet, &ma0.ptet));
}

static void move_sideways(
    MeridionalAnnulus *ma,
    DirectionToTravel direction)
{
    MeridionalAnnulus new_ma;
    Permutation        gluing;
    FaceIndex          old_leading_face,
                      old_trailing_face,
                      *new_leading_face,
                      *new_trailing_face;

/*
 * Create references to the leading and trailing faces,
 * according to which direction we're going.
 */

if (direction == to_the_east)
{
    old_leading_face = ma->ptet.left_face;
    old_trailing_face = ma->ptet.right_face;
    new_leading_face = &new_ma.ptet.left_face;
    new_trailing_face = &new_ma.ptet.right_face;
}
else
{
    old_leading_face = ma->ptet.right_face;
    old_trailing_face = ma->ptet.left_face;
    new_leading_face = &new_ma.ptet.right_face;
    new_trailing_face = &new_ma.ptet.left_face;
}

/*

```

```

    * Find the new Tetrahedron.
    */
    new_ma.ptet.tet = ma->ptet.tet->neighbor[old_leading_face];

    /*
    * For convenience, record the pertinent gluing.
    */
    gluing = ma->ptet.tet->gluing[old_leading_face];

    /*
    * Find the new_trailing_face.
    */
    *new_trailing_face = EVALUATE(gluing, old_leading_face);

    /*
    * The values of the remaining _faces will depend on which is degenerate.
    */
    if (new_ma.ptet.tet->extra->drilling_curve_intersects_face
        [EVALUATE(gluing, old_trailing_face)] == TRUE)
    {
        *new_leading_face = EVALUATE(gluing, old_trailing_face);
        new_ma.ptet.bottom_face = EVALUATE(gluing, ma->ptet.bottom_face);
        new_ma.ptet.near_face = EVALUATE(gluing, ma->ptet.near_face);

        new_ma.current_position = ma->current_position;
    }
    if (new_ma.ptet.tet->extra->drilling_curve_intersects_face
        [EVALUATE(gluing, ma->ptet.bottom_face)] == TRUE)
    {
        *new_leading_face = EVALUATE(gluing, ma->ptet.bottom_face);
        new_ma.ptet.bottom_face = EVALUATE(gluing, ma->ptet.near_face);
        new_ma.ptet.near_face = EVALUATE(gluing, old_trailing_face);

        new_ma.current_position = (ma->current_position + 2) % 3;
    }
    if (new_ma.ptet.tet->extra->drilling_curve_intersects_face
        [EVALUATE(gluing, ma->ptet.near_face)] == TRUE)
    {
        *new_leading_face = EVALUATE(gluing, ma->ptet.near_face);
        new_ma.ptet.bottom_face = EVALUATE(gluing, old_trailing_face);
        new_ma.ptet.near_face = EVALUATE(gluing, ma->ptet.bottom_face);

        new_ma.current_position = (ma->current_position + 1) % 3;
    }

    /*
    * Set the orientation in the ptet.
    */
    new_ma.ptet.orientation = (parity[gluing] == orientation_preserving) ?
                             ma->ptet.orientation :
                             ! ma->ptet.orientation;

    /*
    * Copy the new data to the original data structure.
    */
    *ma = new_ma;
}

static void transfer_CS(
    Triangulation *old_manifold,
    Triangulation *new_manifold)
{
    Triangulation *old_copy,
                  *new_copy;

    /*
    * If the old_manifold doesn't have a known CS value,
    * then we certainly can't find one for the new_manifold.
    */
    if (old_manifold->CS_fudge_is_known == FALSE)
        return;

    /*

```

```

    * To minimize the potential trouble with negatively
    * oriented Tetrahedra, we transfer the CS_value from
    * the complete structure on the old_manifold to the
    * (,)(,)...(,)(1,0) Dehn filling on the new_manifold.
    */

/*
    * First make copies of the old_manifold and the new_manifold,
    * so we can feel free to mess 'em up.
    */
copy_triangulation(old_manifold, &old_copy);
copy_triangulation(new_manifold, &new_copy);

/*
    * Restore the complete solution on old_copy,
    * and complete all the cusps.
    */
copy_solution(old_copy, complete, filled);
complete_all_cusps(old_copy);

/*
    * Attempt to compute the CS_value for old_copy
    * based on its CS_fudge.
    */
compute_CS_value_from_fudge(old_copy);

/*
    * If no CS_value has materialized (e.g. due to
    * negatively oriented tetrahedra), we're out of luck.
    */
if (old_copy->CS_value_is_known == FALSE)
{
    free_triangulation(old_copy);
    free_triangulation(new_copy);
    return;
}

/*
    * Restore the complete solution on new_copy,
    * complete all the cusps, and then do a (1,0)
    * Dehn filling on the recently drilled cusp
    * to obtain a manifold isometric to old_copy.
    */
copy_solution(new_copy, complete, filled);
complete_all_cusps(new_copy);
set_cusp_info(new_copy, new_copy->num_cusps - 1, FALSE, 1.0, 0.0);
do_DeHN_filling(new_copy);

/*
    * Transfer the CS_value from old_copy to new_copy.
    */
new_copy->CS_value_is_known = TRUE;
new_copy->CS_value[ultimate] = old_copy->CS_value[ultimate];
new_copy->CS_value[penultimate] = old_copy->CS_value[penultimate];

/*
    * With luck, we can convert the CS_value into a CS_fudge.
    * (Without luck, CS_fudge_is_known will be set to FALSE,
    * and subsequent operations will be vacuous.)
    */
compute_CS_fudge_from_value(new_copy);

/*
    * Transfer the CS_fudge from new_copy to new_manifold.
    */
new_manifold->CS_fudge_is_known = new_copy->CS_fudge_is_known;
new_manifold->CS_fudge[ultimate] = new_copy->CS_fudge[ultimate];
new_manifold->CS_fudge[penultimate] = new_copy->CS_fudge[penultimate];

/*
    * If everything is still hanging together, we can
    * use the CS_fudge to compute the CS_value.
    */
compute_CS_value_from_fudge(new_manifold);

```

```
/*
 * Free the copies.
 */
free_triangulation(old_copy);
free_triangulation(new_copy);
}
```