

```

/*
 * peripheral_curves.c
 *
 *
 * This file provides the function
 *
 *     void peripheral_curves(Triangulation *manifold);
 *
 * which puts a meridian and longitude on each cusp.  If the
 * manifold is oriented, the meridian and longitude adhere to
 * the usual orientation convention; this is, if you place your
 * right hand on the torus with your fingers pointing in the
 * direction of the meridian and your thumb pointing in the
 * direction of the longitude, then your palm will face the
 * cusp while the back of your hand faces the fat part of the
 * manifold.  Note that this corresponds to the usual convention
 * for orienting meridians and longitudes on link complements.
 *
 * Even if the manifold isn't oriented, the peripheral curves
 * adhere to the standard orientation convention relative to
 * the orientation of each Cusp's orientation double cover (more
 * on this below).
 *
 * peripheral_curves() does not need to know the CuspTopology
 * ahead of time.  It figures it out for itself and records the
 * result in the field cusp->topology.
 *
 * The remainder of this documentation will
 *
 *     (1) Define the meridian and the longitude.
 *
 *     (2) Describe the data structure used to store
 *         meridians and longitudes.
 *
 *     (3) Explain the algorithm which peripheral_curves()
 *         uses to find meridians and longitudes.
 *
 * (1) Definition of the meridian and the longitude.
 *
 * The meridian and the longitude of an orientable cusp are
 * any pair of simple closed curves which intersect exactly
 * once.  If the manifold is orientable, they will adhere to
 * the orientation convention described above.  peripheral_curves()
 * finds meridians and longitudes which are reasonably short
 * in the sense that they pass through a small number of
 * triangles in the triangulation of the boundary torus.
 *
 * The meridian and the longitude of a nonorientable cusp are
 * defined more precisely.  There are exactly four nontrivial
 * simple closed curves on a Klein bottle, up to isotopy.  I'd
 * like to provide a picture of them, but, alas, there's no way
 * to include a picture in an ASCII file, so I must ask you to
 * draw your own picture as you read along.  Define the
 * Klein bottle as a cylinder with ends glued.  Parameterize
 * the cylinder by (x, theta), where  $-1 \leq x \leq +1$  and theta
 * is defined mod  $2\pi$  as for a circle.  To make the Klein
 * bottle, identify the cylinder's ends via the mapping
 *  $(-1, \theta) \rightarrow (+1, -\theta)$ .  With this notation, the four
 * simple closed curves on the Klein bottle are
 *
 *     A:  theta = 0
 *     B:  theta = pi
 *     C:  (theta = pi/2) U (theta = - pi/2)
 *     D:  x = 0
 *
 * The longitude will be either curve A or curve B.  Thus,
 * counting orientation, there are four possible longitudes.
 * (Note: as explained in section (2) immediately following,
 * the longitude of a Klein bottle is actually stored as its
 * preimage in the Klein bottle's double cover.)
 * The meridian will be curve D.  Curve D cannot be oriented,
 * because it's isotopic to its inverse.

```

```
* The holonomies of the longitude and meridian of a Klein
* bottle have some very special properties, which are
* discussed in the comment at the top of holonomy.c.
*
*
* (2) How meridians and longitudes are stored.
*
* The meridian and longitude are stored not on the boundary
* component itself, but on its orientation double cover.
* The double cover of a Klein bottle is a torus. The double
* cover of a torus is the union of two tori, only one of
* which is actually used (the right_handed one, if the
* manifold is oriented). The reason we want to always
* store curves on tori and never on Klein bottles directly
* is that the fundamental group of the torus is abelian:
* isotopy classes of curves can be recovered from homological
* information. The reason only the right_handed sheet
* of the cover is used when the manifold is oriented is
* that the holonomy of a Dehn filling curve on the left_handed
* sheet is not a complex analytic function of the tetrahedron
* shapes, but rather the complex conjugate of such a function;
* we need complex analytic functions to compute the
* derivative matrix in the complex version of Newton's
* method used to find hyperbolic structures for oriented
* manifolds.
*
* Each torus is the union of triangular cross sections of
* ideal vertices. Each ideal vertex of each Tetrahedron
* contributes two triangles, one with the right_handed
* orientation and one with the left_handed orientation.
* Visualize the right-handed triangle as containing a
* counter-clockwise oriented circle, and the left-handed
* triangle as containing a clockwise-oriented circle, as viewed
* from infinity relative to the right_handed orientation of the
* Tetrahedron. The triangles piece together so that orientations
* of neighboring circles agree: if two neighboring Tetrahedra
* have opposite orientations, then the right_handed triangles
* of one connect with the left_handed triangles of the other;
* if the two tetrahedra have the same orientation, then the
* right_handed triangles of one match to the right_handed triangles
* of the other, and left_handed to left_handed. Note that
* the components of the orientation double cover of the Cusps
* are all oriented, not just orientable (use the rule which
* says to view the circles so that all appear counterclockwise
* as seen from infinity).
*
* The meridian and longitude are specified by their
* intersection numbers with the triangular cross sections.
* tet->curve[M][right_handed][v][f] is the net number of
* times the meridian crosses side f of the right_handed
* triangle at vertex v, and similarly for the
* longitude and the left_handed triangle.
* A positive intersection number means the curve is
* entering the triangle. The sides of the triangle are
* numbered according to the faces of the tetrahedron
* containing them. E.g., the sides of the triangle at
* vertex 2 will be numbered 0, 1 and 3. The array vt_side[][]
* in tables.c lets you refer to the sides of a triangle by
* the integers 0, 1, 2, if this suits your purposes.
* Note that these intersection numbers are sufficient to
* reconstruct a simple closed curve up to isotopy.
*
* The longitude of the Klein bottle is a special case in
* that its preimage in the double cover is connected. It
* is stored as the complete preimage. All other curves are
* stored as one component of their two-component preimages.
* By the way, the two candidates for the longitude (curves
* A and B in the discussion above) lift to the same curve in
* the double cover.
*
*
* (3) How peripheral_curves() finds the meridian and the
* longitude.
```

```

*   There are three main steps:
*
*       Compute a fundamental domain for the boundary component.
*
*       Identify it as a torus or Klein bottle.
*
*       Find the longitude and meridian.
*
*   The plan for finding a fundamental domain is
*   conceptually simple: initialize the domain as a single
*   triangular vertex cross section, and then expand it outward
*   by adding neighboring triangles in a breadth-first
*   fashion, until the domain fills the entire cusp.
*
*   This plan is implemented with the PerimeterPiece data
*   structure. At each step, a circular linked list of
*   PerimeterPieces defines the boundary of the domain.
*   (Each PerimeterPiece corresponds to one edge of one
*   triangle on the perimeter of the domain.) The algorithm
*   is to keep going around the perimeter, adding new
*   PerimeterPieces as the domain expands across triangles
*   which were not previously included.
*
*   While this process goes on, a second data structure is
*   being created. An array of 4 Extra fields attached to
*   each tetrahedron (one Extra field for each vertex) records
*
*       (a) whether the vertex has been included in the domain,
*   and, if so,
*
*       (b) which other vertex (the "parent vertex") was
*           responsible for including it.
*
*   The result is a tree structure, with the root at the
*   original triangle and the leaves at the perimeter. In
*   a moment we'll see how this tree is used to create the
*   longitudes and meridians.
*
*   Once the domain fills the entire cusp, we check whether
*   there are any vertices of order one on the perimeter, and
*   if so we remove them by cancelling the adjacent
*   PerimeterPieces.
*
*   Conceptually (but NOT in the code) we imagine removing
*   vertices of order two, thereby fusing the adjacent edges
*   into one. This must yield a fundamental domain which is
*   either a square or a hexagon. Here's the proof. Assume
*   a  $2n$ -gon has pairs of sides glued so as to form a torus
*   or a Klein bottle, and assume all vertices have order
*   three or greater. Since the  $2n$ -gon itself has  $2n$  vertices,
*   there can be at most  $2n/3$  vertices in the resulting
*   cell-decomposition of the torus or Klein bottle.
*   Compute the Euler characteristic:
*
*       0 = Euler characteristic
*         = vertices - edges + faces
*       <=  $2n/3 - n + 1$ 
*         =  $1 - n/3$ 
*
*       =>  $n \leq 3$ 
*       => the  $2n$ -gon is a bigon, a square or a hexagon
*
*   A case-by-case analysis reveals that the only possible
*   gluings are

```

	square	hexagon
torus	abAB	abcABC
Klein bottle	abAb aabb	abcAcb aabccB

```

* where the notation is what you would expect. I wish I could
* provide an illustration of each gluing, but in platform-
* independent text file this just isn't possible. So I ask
* that you make your own illustration of each gluing.
*
* It is now straightforward to apply the definitions of the
* longitude and meridian to each gluing. The details are
* spelled out in the documentation contained in the function
* find_meridian_and_longitude() and, especially, the functions
* it calls.
*/

#include "kernel.h"

typedef struct PerimeterPiece PerimeterPiece;

struct extra
{
    /*
     * Has this vertex been included in the fundamental domain?
     */
    Boolean visited;

    /*
     * Which vertex of which tetrahedron is its parent in the
     * tree structure?
     * (parent_tet == NULL at the root.)
     */
    Tetrahedron *parent_tet;
    VertexIndex parent_vertex;

    /*
     * Which side of this vertex faces the parent vertex?
     * Which side of the parent vertex faces this vertex?
     */
    FaceIndex this_faces_parent,
              parent_faces_this;

    /*
     * What is the orientation of this vertex in the
     * fundamental domain?
     */
    Orientation orientation;

    /*
     * Which PerimeterPiece, if any, is associated with
     * a given edge of the triangle at this vertex?
     * (As you might expect, its_perimeter_piece[i] refers
     * to the edge of the triangle contained in face i of
     * the Tetrahedron.)
     */
    PerimeterPiece *its_perimeter_piece[4];

    /*
     * When computing intersection numbers in
     * adjust_Klein_cusp_orientations() we want to allow for
     * the possibility that the Triangulation's scratch_curves
     * are already in use, so we copy them to scratch_curve_backup,
     * and restore them when we're done.
     */
    int scratch_curve_backup[2][2][2][4][4];
};

struct PerimeterPiece
{
    Tetrahedron *tet;
    VertexIndex vertex;
    FaceIndex face;
    Orientation orientation; /* How the PerimeterPiece sees the tetrahedron */
    Boolean checked;
    PerimeterPiece *mate; /* the PerimeterPiece this one is glued to . . . */
    GluingParity gluing_parity; /* . . . and how they match up */
    PerimeterPiece *next; /* the neighbor in the counterclockwise direction */
};

```

```

    PerimeterPiece *prev; /* the neighbor in the clockwise direction */
};

/*
 * The following enum lists the six possible gluing
 * patterns for a torus or Klein bottle.
 */
typedef int GluingPattern;
enum
{
    abAB, /* square torus */
    abcABC, /* hexagonal torus */
    abAb, /* standard square Klein bottle */
    aabb, /* P^2 # P^2 square Klein bottle */
    abcAcb, /* standard hexagonal Klein bottle */
    aabccB /* P^2 # P^2 hexagonal Klein bottle */
};

static void zero_peripheral_curves(Triangulation *manifold);
static void attach_extra(Triangulation *manifold);
static void free_extra(Triangulation *manifold);
static void initialize_flags(Triangulation *manifold);
static Boolean cusp_has_curves(Triangulation *manifold, Cusp *cusp);
static void do_one_cusp(Triangulation *manifold, Cusp *cusp);
static void pick_base_tet(Triangulation *manifold, Cusp *cusp, Tetrahedron **
    base_tet, VertexIndex *base_vertex);
static void set_up_perimeter(Tetrahedron *base_tet, VertexIndex base_vertex,
    PerimeterPiece **perimeter_anchor);
static void expand_perimeter(PerimeterPiece *perimeter_anchor);
static void find_mates(PerimeterPiece *perimeter_anchor);
static void simplify_perimeter(PerimeterPiece **perimeter_anchor);
static void find_meridian_and_longitude(PerimeterPiece *perimeter_anchor,
    CuspTopology *cusp_topology);
static void advance_to_next_side(PerimeterPiece **pp);
static GluingPattern determine_gluing_pattern(PerimeterPiece *side[6], int num_sides);
static void do_torus(PerimeterPiece *side[6]);
static void do_standard_Klein_bottle(PerimeterPiece *side[6], int num_sides);
static void do_P2P2_Klein_bottle(PerimeterPiece *side[6], int num_sides);
static void trace_curve(PerimeterPiece *start, PeripheralCurve
    trace_which_curve, TraceDirection trace_direction, Boolean use_opposite_orientation);
static void free_perimeter(PerimeterPiece *perimeter_anchor);
static void adjust_Klein_cusp_orientations(Triangulation *manifold);
static void reverse_meridians_where_necessary(Triangulation *manifold);
static void backup_scratch_curves(Triangulation *manifold);
static void restore_scratch_curves(Triangulation *manifold);

void peripheral_curves(
    Triangulation *manifold)
{
    Cusp *cusp;

    zero_peripheral_curves(manifold);
    attach_extra(manifold);
    initialize_flags(manifold);

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

        if (cusp->is_finite == FALSE) /* 97/2/4 Added to accomodate finite vertices. */

            do_one_cusp(manifold, cusp);

    adjust_Klein_cusp_orientations(manifold);

    free_extra(manifold);
}

void peripheral_curves_as_needed(
    Triangulation *manifold)
{

```

```

/*
 * Add peripheral curves only to cusps for which all the
 * tet->curve[i][j][k][l] fields are zero.
 */

Cusp      *cusp;

attach_extra(manifold);
initialize_flags(manifold);

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

    if (cusp->is_finite == FALSE
        && cusp_has_curves(manifold, cusp) == FALSE)

        do_one_cusp(manifold, cusp);

adjust_Klein_cusp_orientations(manifold);

free_extra(manifold);
}

static void zero_peripheral_curves(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int          i,
                j,
                k,
                l;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
        for (i = 0; i < 2; i++)
            for (j = 0; j < 2; j++)
                for (k = 0; k < 4; k++)
                    for (l = 0; l < 4; l++)
                        tet->curve[i][j][k][l] = 0;
}

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", "peripheral_curves");

        /*
         * Attach the locally defined struct extra.
         */
        tet->extra = NEW_ARRAY(4, 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 initialize_flags(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    VertexIndex v;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
        for (v = 0; v < 4; v++)
            tet->extra[v].visited = FALSE;
}

static Boolean cusp_has_curves(
    Triangulation *manifold,
    Cusp *cusp)
{
    Tetrahedron *tet;
    VertexIndex v;
    FaceIndex f;
    PeripheralCurve c;
    Orientation h;

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

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

            if (tet->cusp[v] == cusp)

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

                    if (f != v)

                        for (c = 0; c < 2; c++) /* c = M, L */

                            for (h = 0; h < 2; h++) /* h = right_handed, left_handed */

                                if (tet->curve[c][h][v][f] != 0)

                                    return TRUE;

    return FALSE;
}

static void do_one_cusp(
    Triangulation *manifold,
    Cusp *cusp)
{
    Tetrahedron *base_tet;
    VertexIndex base_vertex;
    PerimeterPiece *perimeter_anchor;

```

```

    pick_base_tet(manifold, cusp, &base_tet, &base_vertex);
    set_up_perimeter(base_tet, base_vertex, &perimeter_anchor);
    expand_perimeter(perimeter_anchor);
    find_mates(perimeter_anchor);
    simplify_perimeter(&perimeter_anchor);
    find_meridian_and_longitude(perimeter_anchor, &cusp->topology);
    free_perimeter(perimeter_anchor);
}

```

```

static void pick_base_tet(
    Triangulation *manifold,
    Cusp *cusp,
    Tetrahedron **base_tet,
    VertexIndex *base_vertex)
{
    Tetrahedron *tet;
    VertexIndex v;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
        for (v = 0; v < 4; v++)
            if (tet->cusp[v] == cusp)
            {
                *base_tet = tet;
                *base_vertex = v;
                return;
            }

    /*
     * If pick_base_tet() didn't find any vertex belonging
     * to the specified cusp, we're in big trouble.
     */
    uFatalError("pick_base_tet", "peripheral_curves");
}

```

```

static void set_up_perimeter(
    Tetrahedron *base_tet,
    VertexIndex base_vertex,
    PerimeterPiece **perimeter_anchor)
{
    int i;
    PerimeterPiece *pp[3];

    base_tet->extra[base_vertex].visited = TRUE;
    base_tet->extra[base_vertex].parent_tet = NULL;
    base_tet->extra[base_vertex].orientation = right_handed;

    for (i = 0; i < 3; i++)
        pp[i] = NEW_STRUCT(PerimeterPiece);

    for (i = 0; i < 3; i++)
    {
        pp[i]->tet = base_tet;
        pp[i]->vertex = base_vertex;
        pp[i]->face = vt_side[base_vertex][i];
        pp[i]->orientation = right_handed;
        pp[i]->checked = FALSE;
        pp[i]->next = pp[(i+1)%3];
        pp[i]->prev = pp[(i+2)%3];
    }

    *perimeter_anchor = pp[0];
}

```

```

/*
 * expand_perimeter() starts with the initial triangular
 * perimeter found by set_up_perimeter() and expands it in
 * breadth-first fashion. It keeps going around and around
 * the perimeter, pushing it outwards wherever possible.
 * To know when it's done, it keeps track of the number

```



```

* of PerimeterPieces which have not yet been checked.
* When this number is zero, it's done.
*/

static void expand_perimeter(
    PerimeterPiece *perimeter_anchor)
{
    int num_unchecked_pieces;
    PerimeterPiece *pp,
                  *new_piece;
    Permutation gluing;
    Tetrahedron *nbr_tet;
    VertexIndex nbr_vertex;
    FaceIndex nbr_back_face,
              nbr_left_face,
              nbr_right_face;
    Orientation nbr_orientation;

    for (num_unchecked_pieces = 3, pp = perimeter_anchor;
         num_unchecked_pieces;
         pp = pp->next)

        if (pp->checked == FALSE)
        {
            gluing = pp->tet->gluing[pp->face];
            nbr_tet = pp->tet->neighbor[pp->face];
            nbr_vertex = EVALUATE(gluing, pp->vertex);
            if (nbr_tet->extra[nbr_vertex].visited)
            {
                pp->checked = TRUE;
                num_unchecked_pieces--;
            }
            else
            {
                /*
                 * Extend the tree to the neighboring vertex.
                 */

                nbr_back_face = EVALUATE(gluing, pp->face);

                if (parity[gluing] == orientation_preserving)
                    nbr_orientation = pp->orientation;
                else
                    nbr_orientation = ! pp->orientation;

                if (nbr_orientation == right_handed)
                {
                    nbr_left_face = remaining_face[nbr_vertex][nbr_back_face];
                    nbr_right_face = remaining_face[nbr_back_face][nbr_vertex];
                }
                else
                {
                    nbr_left_face = remaining_face[nbr_back_face][nbr_vertex];
                    nbr_right_face = remaining_face[nbr_vertex][nbr_back_face];
                }

                nbr_tet->extra[nbr_vertex].visited = TRUE;
                nbr_tet->extra[nbr_vertex].parent_tet = pp->tet;
                nbr_tet->extra[nbr_vertex].parent_vertex = pp->vertex;
                nbr_tet->extra[nbr_vertex].this_faces_parent = nbr_back_face;
                nbr_tet->extra[nbr_vertex].parent_faces_this = pp->face;
                nbr_tet->extra[nbr_vertex].orientation = nbr_orientation;

                /*
                 * Extend the perimeter across the neighboring
                 * vertex. The new PerimeterPiece is added on
                 * the right side of the old one, so that the
                 * pp = pp->next step in the loop moves us past
                 * both the old and new perimeter pieces. This
                 * causes the perimeter to expand uniformly in
                 * all directions.
                 */

                new_piece = NEW_STRUCT(PerimeterPiece);

```

```

        new_piece->tet          = nbr_tet;
        new_piece->vertex       = nbr_vertex;
        new_piece->face         = nbr_right_face;
        new_piece->orientation  = nbr_orientation;
        new_piece->checked      = FALSE;
        new_piece->next         = pp;
        new_piece->prev        = pp->prev;

        pp->prev->next = new_piece;

        pp->tet          = nbr_tet;
        pp->vertex       = nbr_vertex;
        pp->face         = nbr_left_face;
        pp->orientation  = nbr_orientation;
        pp->checked      = FALSE; /* unchanged */
        pp->next         = pp->next; /* unchanged */
        pp->prev        = new_piece;

        /*
         * Increment the count of unchecked pieces.
         */
        num_unchecked_pieces++;
    }
}

static void find_mates(
    PerimeterPiece *perimeter_anchor)
{
    PerimeterPiece *pp;
    Tetrahedron *nbr_tet;
    Permutation gluing;
    VertexIndex nbr_vertex;
    FaceIndex nbr_face;

    /*
     * First tell the tetrahedra about the PerimeterPieces.
     */
    pp = perimeter_anchor;
    do
    {
        pp->tet->extra[pp->vertex].its_perimeter_piece[pp->face] = pp;
        pp = pp->next;
    }
    while (pp != perimeter_anchor);

    /*
     * Now let each PerimeterPiece figure out who its mate is.
     */
    pp = perimeter_anchor;
    do
    {
        nbr_tet      = pp->tet->neighbor[pp->face];
        gluing        = pp->tet->gluing[pp->face];
        nbr_vertex    = EVALUATE(gluing, pp->vertex);
        nbr_face      = EVALUATE(gluing, pp->face);

        pp->mate = nbr_tet->extra[nbr_vertex].its_perimeter_piece[nbr_face];
        pp->gluing_parity =
            (pp->orientation == pp->mate->orientation) ==
            (parity[gluing] == orientation_preserving) ?
            orientation_preserving :
            orientation_reversing;

        pp = pp->next;
    }
    while (pp != perimeter_anchor);
}

static void simplify_perimeter(

```

```

    PerimeterPiece  **perimeter_anchor)
{
    PerimeterPiece  *pp,
                    *stop,
                    *dead0,
                    *dead1;

    /*
     * The plan here is to cancel adjacent edges of the form
     *
     *          --o--->---o---<---o--
     *
     * Travelling around the perimeter looking for such
     * edges is straightforward.  For each PerimeterPiece (pp),
     * we check whether it will cancel with its lefthand
     * neighbor (pp->next).  If it doesn't cancel, we advance
     * one step to the left (pp = pp->next).  If it does cancel,
     * we move back one step to the right, to allow further
     * cancellation in case the previously cancelled edges were
     * part of a sequence
     *
     *      . . . --o--->---o--->---o---<---o---<---o-- . . .
     *
     * One could no doubt devise a clever and efficient
     * way of deciding when to stop (readers are invited
     * to submit solutions), but to save wear and tear on
     * the programmer's brain, the present algorithm simply
     * keeps going until it has made a complete trip around
     * the perimeter without doing any cancellation.  The
     * variable "stop" records the first noncancelling
     * PerimeterPiece which was encountered after the most
     * recent cancellation.  Here's the loop in skeleton form:
     *
     *      pp = perimeter_anchor;
     *      stop = NULL;
     *
     *      while (pp != stop)
     *      {
     *          if (pp cancels with its neighbor)
     *          {
     *              pp = pp->prev;
     *              stop = NULL;
     *          }
     *          else          (pp doesn't cancel with its neighbor)
     *          {
     *              if (stop == NULL)
     *                  stop = pp;
     *              pp = pp->next;
     *          }
     *      }
     */

    pp  = *perimeter_anchor;
    stop = NULL;

    while (pp != stop)
    {
        /*
         * Check whether pp and the PerimeterPiece to
         * its left will cancel each other.
         */
        if (pp->next == pp->mate
            && pp->gluing_parity == orientation_preserving)
        {
            /*
             * Note the addresses of the PerimeterPieces
             * which cancel . . .
             */
            dead0 = pp;
            dead1 = pp->next;

            /*
             * . . . then remove them from the perimeter.
             */

```

```

    dead0->prev->next = dead1->next;
    dead1->next->prev = dead0->prev;

    /*
     * Move pp back to the previous PerimeterPiece to
     * allow further cancellation.
     */
    pp = dead0->prev;

    /*
     * Deallocate the cancelled PerimeterPieces.
     */
    my_free(dead0);
    my_free(dead1);

    /*
     * We don't want to leave *perimeter_anchor
     * pointing to a dead PerimeterPiece, so set
     * it equal to a piece we know is still alive.
     */
    *perimeter_anchor = pp;

    /*
     * We just did a cancellation, so set the
     * variable stop to NULL.
     */
    stop = NULL;
}
else
{
    /*
     * If this is the first noncancelling PerimeterPiece
     * after a sequence of one or more cancellations,
     * record its address in the variable stop.
     */
    if (stop == NULL)
        stop = pp;

    /*
     * Advance to the next PerimeterPiece.
     */
    pp = pp->next;
}
}

static void find_meridian_and_longitude(
    PerimeterPiece *perimeter_anchor,
    CuspTopology *cusp_topology)
{
    PerimeterPiece *pp,
    *side[6];
    int i,
    num_sides;

    /*
     * As explained in the documentation at the top of
     * this file, the fundamental domain for the cusp
     * will be either a square or a hexagon. Go around
     * the perimeter, recording the first PerimeterPiece
     * on each side of the square or hexagon.
     */
    pp = perimeter_anchor;
    for (i = 0; i < 6; i++)
    {
        advance_to_next_side(&pp);
        side[i] = pp;
    }

    /*
     * Is it a square or a hexagon?
     */
    if (side[0] == side[4])

```

```

    num_sides = 4;
else
    num_sides = 6;

/*
 * Split into cases, according to how the square or
 * hexagon's edges are glued. The six types of gluings
 * are explained in the documentation at the top of
 * this file.
 */
switch (determine_gluings_pattern(side, num_sides))
{
    case abAB:
    case abcABC:
        do_torus(side);
        *cusp_topology = torus_cusp;
        break;

    case abAb:
    case abcAcb:
        do_standard_Klein_bottle(side, num_sides);
        *cusp_topology = Klein_cusp;
        break;

    case aabb:
    case aabccB:
        do_P2P2_Klein_bottle(side, num_sides);
        *cusp_topology = Klein_cusp;
        break;
}
}

/*
 * advance_to_next_side() advances the pointer *pp to
 * point to the first PerimeterPiece on the next side
 * of the square or hexagon, travelling counterclockwise.
 */

static void advance_to_next_side(
    PerimeterPiece **pp)
{
    PerimeterPiece *p0,
                  *p1;

    /*
     * Let p0 and p1 point to the given PerimeterPiece
     * and its mate.
     */
    p0 = *pp;
    p1 = (*pp)->mate;

    /*
     * Move along the perimeter until p0 and p1 part company,
     * or until their relative orientation changes.
     */
    if (p0->gluing_parity == orientation_preserving)
    {
        do
        {
            p0 = p0->next;
            p1 = p1->prev;
        }
        while ( p0->mate == p1
            && p0->gluing_parity == orientation_preserving);
    }
    else /* (*pp)->gluing_parity == orientation_reversing */
    {
        do
        {
            p0 = p0->next;
            p1 = p1->next;
        }
        while ( p0->mate == p1

```

```

        && p0->gluing_parity == orientation_reversing);

/*
 * p0 now points to the first PerimeterPiece in the next
 * edge of the square or hexagon. Write its value into *pp.
 */
*pp = p0;
}

static GluingPattern determine_gluing_pattern(
    PerimeterPiece *side[6],
    int num_sides)
{
    int i;

/*
 * Please draw pictures of the six possible gluings
 * shown in the table in the documentation at the
 * top of this file. They will show that the logic
 * of this function, as summarized in the following
 * skeleton code, is correct.
 *
 * if (there is an orientation reversing side)
 *     if (there are two adjacent, matching, orientation reversing sides)
 *         if (num_sides == 4)
 *             return(P^2 # P^2 square Klein bottle)
 *         else (num_sides == 6)
 *             return(P^2 # P^2 hexagonal Klein bottle)
 *     else
 *         if (num_sides == 4)
 *             return(standard square Klein bottle)
 *         else (num_sides == 6)
 *             return(standard hexagonal Klein bottle)
 * else (all sides are orientation preserving)
 *     if (num_sides == 4)
 *         return(square torus)
 *     else (num_sides == 6)
 *         return(hexagonal torus)
 */

/*
 * Look for an orientation reversing side.
 * If one is found, check whether it's adjacent to its mate.
 */

    for (i = 0; i < num_sides; i++)
        if (side[i]->gluing_parity == orientation_reversing)
        {
            if (side[i]->mate == side[(i+1)%num_sides]
                || side[i]->mate == side[(i-1+num_sides)%num_sides])
            {
                if (num_sides == 4)
                    return aabb; /* P^2 # P^2 square Klein bottle */
                else
                    return aabccB; /* P^2 # P^2 hexagonal Klein bottle */
            }
            else
            {
                if (num_sides == 4)
                    return abAb; /* standard square Klein bottle */
                else
                    return abcAcb; /* standard hexagonal Klein bottle */
            }
        }

/*
 * No orientation reversing side was found.
 * The surface is a torus.
 */

    if (num_sides == 4)
        return abAB; /* square torus */

```

```

    else
        return abcABC; /* hexagonal torus */
}

static void do_torus(
    PerimeterPiece *side[6]
/* int          num_sides */)
{
    /*
     * The following calls to trace_curve() will always produce
     * a meridian and longitude which intersect exactly once.
     * If the manifold is orientable, they will adhere to the
     * orientation convention described at the top of this file.
     * (The proof of this relies on the fact that
     * set_up_perimeter() views the base vertex with the
     * right_handed orientation. Thus in an oriented manifold
     * all vertices are viewed with the right_handed orientation.)
     */

    trace_curve(side[0],          L, trace_backwards, FALSE);
    trace_curve(side[0]->mate, L, trace_forwards,  FALSE);
    trace_curve(side[1],          M, trace_backwards, FALSE);
    trace_curve(side[1]->mate, M, trace_forwards,  FALSE);
}

static void do_standard_Klein_bottle(
    PerimeterPiece *side[6],
    int          num_sides)
{
    int i;

    /*
     * Let the meridian connect the unique pair of
     * orientation_preserving sides.
     */
    for (i = 0; i < num_sides; i++)
        if (side[i]->gluing_parity == orientation_preserving)
        {
            trace_curve(side[i],          M, trace_backwards, FALSE);
            trace_curve(side[i]->mate, M, trace_forwards,  FALSE);
            break;
        }

    /*
     * Let the longitude connect a pair of orientation_reversing
     * sides. Store it as its complete preimage in the double
     * cover, as explained in the documentation at the top of
     * this file.
     */
    for (i = 0; i < num_sides; i++)
        if (side[i]->gluing_parity == orientation_reversing)
        {
            trace_curve(side[i],          L, trace_backwards, FALSE);
            trace_curve(side[i]->mate, L, trace_forwards,  FALSE);
            trace_curve(side[i],          L, trace_backwards, TRUE);
            trace_curve(side[i]->mate, L, trace_forwards,  TRUE);
            break;
        }
}

static void do_P2P2_Klein_bottle(
    PerimeterPiece *side[6],
    int          num_sides)
{
    int i;
    PerimeterPiece *side0a,
                  *side0b,
                  *side1a,
                  *side1b;

    /*

```

```

    * Let the longitude connect either pair of
    * orientation_reversing sides, and store it as its
    * complete preimage in the double cover, as in
    * do_standard_Klein_bottle() above.
    */
for (i = 0; i < num_sides; i++)
    if (side[i]->gluing_parity == orientation_reversing)
    {
        trace_curve(side[i], L, trace_backwards, FALSE);
        trace_curve(side[i]->mate, L, trace_forwards, FALSE);
        trace_curve(side[i], L, trace_backwards, TRUE);
        trace_curve(side[i]->mate, L, trace_forwards, TRUE);
        break;
    }

/*
 * The meridian is trickier. If you refer to pictures of
 * the aabb and aabccB Klein bottles and the definition
 * of the meridian, you will see the meridian is obtained
 * by gluing each orientation reversing side to the mate
 * of the opposite side of the square or hexagon. For
 * global consistency, the two pieces of the meridian
 * must be drawn on different preimages of the fundamental
 * domain (in the torus double cover).
 */

/*
 * Look for a side followed by its mate.
 */
for (i = 0; i < num_sides; i++)
    if (side[i]->mate == side[(i+1)%num_sides])
    {
        /*
         * Name the four relevant sides.
         */
        side0a = side[i];
        side0b = side[(i + 1)%num_sides];
        side1a = side[(i + num_sides/2)%num_sides];
        side1b = side[(i + 1 + num_sides/2)%num_sides];

        /*
         * Trace out the meridian.
         */
        trace_curve(side0a, M, trace_backwards, FALSE);
        trace_curve(side1b, M, trace_forwards, FALSE);
        trace_curve(side1a, M, trace_backwards, TRUE);
        trace_curve(side0b, M, trace_forwards, TRUE);

        break;
    }
}

/*
 * trace_curve() traces out a curve on a cusp, beginning
 * at start and following the tree structure in Extra
 * back to the base vertex. The result is written directly
 * into the Tetrahedra's meridian or longitude fields,
 * according to whether trace_which_curve is M
 * or L. The curve is directed toward the
 * perimeter if trace_direction is trace_backwards, and
 * toward the base vertex if trace_direction is trace_forwards.
 * The orientation specified by start is used iff
 * use_opposite_orientation is FALSE.
 *
 * To trace a curve from one point on the perimeter to another,
 * you make two calls to trace_curve(), each of which traces
 * from the perimeter to the center. Note that some cancellation
 * is possible, so the final curve need not pass through the
 * center. The final curve will be the unique shortest path
 * in the tree structure.
 */

static void trace_curve(

```



```

    PerimeterPiece *start,
    PeripheralCurve trace_which_curve,
    TraceDirection trace_direction,
    Boolean        use_opposite_orientation)
{
    int            out_sign,
                  in_sign,
                  (*curve)[4][4];
    Tetrahedron *tet,
                  *next_tet;
    VertexIndex vertex,
                  next_vertex;
    Extra        *tet_extra,
                  *next_extra;

    /*
     * Based on the direction of the curve, decide which
     * sign (+1 or -1) is required where the curve leaves
     * a triangle going towards the perimeter, and which
     * is required where it is going towards the center.
     */
    if (trace_direction == trace_backwards)
    {
        out_sign = -1;
        in_sign  = +1;
    }
    else
    {
        out_sign = +1;
        in_sign  = -1;
    }

    /*
     * Record where the curve hits the perimeter.
     */
    curve = start->tet->curve[trace_which_curve];

    curve[use_opposite_orientation ^ start->orientation]
        [start->vertex]
        [start->face]
        += out_sign;

    /*
     * Now trace back to the root.
     */

    tet      = start->tet;
    vertex   = start->vertex;
    tet_extra = &tet->extra[vertex];

    while (tet_extra->parent_tet != NULL)
    {
        /*
         * Note where the curve leaves the present vertex . . .
         */
        curve = tet->curve[trace_which_curve];
        curve[use_opposite_orientation ^ tet_extra->orientation]
            [vertex]
            [tet_extra->this_faces_parent]
            += in_sign;

        /*
         * . . . and where it enters the parent vertex.
         */

        next_tet      = tet_extra->parent_tet;
        next_vertex   = tet_extra->parent_vertex;
        next_extra    = &next_tet->extra[next_vertex];

        curve = next_tet->curve[trace_which_curve];
        curve[use_opposite_orientation ^ next_extra->orientation]
            [next_vertex]
            [tet_extra->parent_faces_this]
            += out_sign;
    }
}

```

```

    /*
     * Move on to the parent vertex.
     */
    tet      = next_tet;
    vertex    = next_vertex;
    tet_extra = next_extra;
}
}

static void free_perimeter(
    PerimeterPiece *perimeter_anchor)
{
    PerimeterPiece *pp,
                  *dead;

    pp = perimeter_anchor;
    do
    {
        dead = pp;
        pp = pp->next;
        my_free(dead);
    }
    while (pp != perimeter_anchor);
}

static void adjust_Klein_cusp_orientations(
    Triangulation *manifold)
{
    /*
     * As explained at the top of this file, a Cusp's peripheral curves
     * live in the its orientation double cover. When I first wrote this
     * file, I didn't worry about the orientation of peripheral curves
     * in nonorientable manifolds. Subsequently it became clear that
     * they should have the standard orientation relative to the Cusp's
     * (oriented, not just orientable) orientation double cover. In
     * the case of a torus Cusp, the peripheral curves live in one
     * (arbitrarily chosen) component of the orientation double cover;
     * in the case of a Klein bottle Cusp, the orientation double cover
     * is connected.
     *
     * Fortunately, it's very easy to check whether the peripheral curves
     * have the standard orientation, and to correct them if necessary.
     * The definition of the standard orientation for peripheral curves on
     * a torus is that when the fingers of your right hand point in the
     * direction of the meridian and your thumb points in the direction
     * of the longitude, the palm of your hand should face the cusp and
     * the back of your hand should face the fat part of the manifold.
     * Combining this with the definition of the intersection number
     * found at the top of intersection_numbers.c reveals that the
     * intersection number of the longitude and the meridian (in that
     * order) should be +1. If it happens to be -1, we must reverse
     * the meridian.
     */

    /*
     * If the manifold is oriented, then the peripheral curves will
     * already have the correct orientation. In fact, they will lie
     * on the right handed sheet of the Cusp's orientation double
     * cover, relative to the orientation of the manifold.
     */
    if (manifold->orientability == oriented_manifold)
        return;

    /*
     * The scratch curves might already be in use, so let's make
     * a copy of whatever's there.
     */
    backup_scratch_curves(manifold);

    /*

```

```

    * Copy the peripheral curves to both sets of scratch_curve fields.
    */
    copy_curves_to_scratch(manifold, 0, FALSE);
    copy_curves_to_scratch(manifold, 1, FALSE);

    /*
    * Compute their intersection numbers.
    */
    compute_intersection_numbers(manifold);

    /*
    * Restore whatever used to be in the scratch_curves.
    */
    restore_scratch_curves(manifold);

    /*
    * On Cusps where the intersection number of the longitude and
    * meridian is -1, reverse the meridian.
    */
    reverse_meridians_where_necessary(manifold);
}

static void reverse_meridians_where_necessary(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i,
        j,
        k;

    /* which Tetrahedron */
    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)

        /* which ideal vertex */
        for (i = 0; i < 4; i++)

            if (tet->cuspid[i]->intersection_number[L][M] == -1)

                /* which side of the vertex */
                for (j = 0; j < 4; j++)

                    if (i != j)

                        /* which sheet (right_handed or left_handed) */
                        for (k = 0; k < 2; k++)

                            tet->curve[M][k][i][j] = - tet->curve[M][k][i][j];
}

static void backup_scratch_curves(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int g,
        h,
        i,
        j,
        k;

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

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

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

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

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

```

```
        for (k = 0; k < 4; k++)

            tet->extra->scratch_curve_backup[g][h][i][j][k]
            = tet->scratch_curve[g][h][i][j][k];
    }

static void restore_scratch_curves(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int          g,
                h,
                i,
                j,
                k;

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

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

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

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

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

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

                            tet->scratch_curve[g][h][i][j][k]
                            = tet->extra->scratch_curve_backup[g][h][i][j][k];
}
```