

```

/*
 * cusp_neighborhoods.c
 *
 * This file provides the following functions for creating and manipulating
 * horospherical cross sections of a manifold's cusps, and computing the
 * triangulation dual to the corresponding Ford complex. These functions
 * communicate with the UI by passing pointers to CuspNeighborhoods
 * data structures; but even though the UI may keep pointers to
 * CuspNeighborhoods, the structure's internal details are private
 * to this file.
 *
 *      CuspNeighborhoods *initialize_cusp_neighborhoods(
 *                                Triangulation      *manifold);
 *
 *      void free_cusp_neighborhoods(
 *                                CuspNeighborhoods  *cusp_neighborhoods);
 *
 * [Many other externally available functions are provided -- please see
 * SnapPea.h for details.]
 *
 * When the canonical cell decomposition dual to the Ford complex is not
 * a triangulation, it is arbitrarily subdivided into tetrahedra.
 *
 * Note: This file uses the fields "displacement" and "displacement_exp"
 * in the Cusp data structure to keep track of where the cusp cross
 * sections are (details appear below). Manifolds not under the care
 * of a CuspNeighborhoods structure should keep the "displacement" set
 * to 0 and the "displacement_exp" set to 1 at all times, so that
 * canonical cell decompositions will be computed relative to cusps
 * cross sections of equal volume.
 */

/*
 * Proposition 1. The area of a horospherical cusp cross section is
 * exactly twice the volume it contains.
 *
 * Proof. Do an integral in the upper half space model of hyperbolic
 * 3-space. Consider a unit square in the horosphere  $z = 1$ , and calculate
 * the volume lying above it as the integral of  $1/z^3 dz$  from  $z = 1$  to
 *  $z = \infty$ . QED
 *
 * Comment. This proposition relies on the hyperbolic manifold having
 * curvature -1. If the curvature had some other value, the proportionality
 * constant would be something other than 2.
 *
 * Comment. The proportionality constant has units of  $1 / \text{distance}$ .
 * Normally, though, one doesn't have to think about units in hyperbolic
 * geometry, because one uses the canonical ones. I just wanted to make
 * sure that nobody is bothered by the fact that we're specifying an area
 * as twice a volume.
 *
 * Comment. We can measure the size of a cusp cross section by area or
 * by volume. The two measures are the same modulo a factor of two.
 *
 * Proposition 2. If we choose a manifold's cusp cross sections to each
 * have area  $(3/8)\sqrt{3}$ , then their interiors cannot overlap themselves
 * or each other.
 *
 * Proof. By Lemma A below, we can choose a set of cusp cross sections
 * with nonoverlapping interiors. Advance each cusp cross section into
 * the fat part of the manifold, until it bumps into itself or another
 * cross section. Look at the horoball packing as seen from a given cusp.
 * Because the cusp is tangent to itself or some other cusp, there'll be
 * a maximally large horoball. If we draw the given cusp as the plane
 *  $z = 1$  in the upper half space model, the maximal horoball (and each of
 * its translates) will appear as a sphere of diameter 1. The view as
 * seen from the given cusp therefore includes a packing of disjoint circles
 * of diameter  $1/2$ . If it's a hexagonal packing the area of the cusp will
 * equal the area of a hexagon of outradius  $1/2$ , which works out to be
 *  $(3/8)\sqrt{3}$ . If it's not a hexagonal packing, the cusp's area will be
 * even greater. In the latter case, we retract the cusp cross section
 * until its area is exactly  $(3/8)\sqrt{3}$ . QED
 */

```

```

* Lemma A. We can choose a set of cusp cross sections with nonoverlapping
* interiors.
*
* Comment. One expects the proof of this lemma to be completely trivial,
* but I don't think it is.
*
* Proof #1. Lemma A follows directly from the Margulis Lemma. The
* required cusp cross sections are portions of the boundary of the thin
* part of the manifold. QED
*
* Proof #2. Start with any decomposition of the manifold into positively
* oriented 3-cells. For example, we could start with the canonical cell
* decomposition constructed in
*
* J. Weeks, Convex hulls and isometries of cusped hyperbolic
* 3-manifolds, Topology Appl. 52 (1993) 127-149.
*
* Choose arbitrary cusp cross sections. Retract each cusp as necessary
* so that its intersection with each 3-cell is "standard". (I don't want
* to spend a lot of time fussing over the wording of this -- the idea is
* that the cross section shouldn't be so far forward that it has
* unnecessary intersections with other faces of the 3-cell.) Then further
* retract each cusp cross section so that it doesn't intersect the other
* cross sections incident to the same 3-cell. This gives the nonoverlapping
* cross sections, as required. QED
*
* Definition. The "home position" of a cusp cross section is the one
* at which its area is  $(3/8)\sqrt{3}$  and its enclosed volume is  $(3/16)\sqrt{3}$ .
*
* By Proposition 2 above, when all the cusps are at their home positions,
* their interiors are disjoint.
*
* The displacement field in the Cusp data structure measures how far
* a cusp cross section is from its home position. The displacement is
* measured towards the fat part of the manifold, so a positive displacement
* means the cusp cross section is larger, and a negative displacement
* means it is smaller.
*
* If we visualize a cusp's home position as a plane at height  $z = 1$  in
* the upper half space model, then after a displacement  $d > 0$  it will be
* at some height  $h < 1$ . Set  $d$  equal to the integral of  $dz/z$  from  $z = h$ 
* to  $z = 1$  to obtain  $d = -\log h$ , or  $h = \exp(-d)$ . It follows that a cusp's
* linear dimensions vary as  $\exp(d)$ , while its area (and therefore its
* enclosed volume) vary as  $\exp(2d)$ . The Cusp data structure stores the
* quantity  $\exp(d)$  in its displacement_exp field, to avoid excessive
* recomputation.
*
* Definition. The "reach" of a cusp is the distance from the cross
* section's home position to the position at which it first bumps into
* itself.
*
* Note that the reach is half the distance from the cusp to itself,
* measured along the shortest homotopically nontrivial path.
* Proposition 2 implies that the reach of each cusp will be nonnegative.
*
* Definition. As a given cusp cross section moves forward into the
* fat part of the manifold, the first cusp cross section it bumps into
* is called its "stopper". The displacement (measured from the home
* position) at which the given cusp meets its stopper is called the
* "stopping displacement".
*
* Comment. Unlike the reach, the stopper and the stopping displacement
* depend on the current displacements of all the cusps in the triangulation.
* They vary dynamically as the user moves the cusp cross sections.
*
* Sometimes the user may wish to change two or more cusp displacements
* in unison. The Cusp's is_tied field supports this. The displacements
* of "tied" cusps always stay the same -- when one changes they all do.
* The tie_group_reach keeps track of the reach of the tied cusps:
* it tells the displacement at which some cusp in the group first
* bumps into itself or some other cusp in the group. Note that the

```

```
* tie_group_reach might be less than the stopping displacement of any
* of its constituent cusps; this is because when a cusp moves forward
* its (ordinary) stopper stays still, but members of its tie group
* move towards it.
*/

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

/*
 * Report all horoballs higher than the requested cutoff_height
 * minus CUTOFF_HEIGHT_EPSILON.  For example, if the user wants to see
 * all horoballs of height at least 0.25, we should report a horoball
 * of height 0.249999999963843.
 */
#define CUTOFF_HEIGHT_EPSILON      1e-6

/*
 * A horoball is considered to be "maximal" iff it's distance from a fixed
 * cusp is within INTERCUSP_EPSILON of being minimal.  (The idea is that
 * if there are several different maximal cusps, whose distances from the
 * fixed cusp differ only by roundoff error, we want to consider all them
 * to be maximal.)
 */
#define INTERCUSP_EPSILON          1e-6

/*
 * If a given cusp does not have a maximal horoball, all other cusp cross
 * sections are retracted in increments of DELTA_DISPLACEMENT until it does.
 * The value of DELTA_DISPLACEMENT should be large enough that the algorithm
 * has a fair shot at the getting a maximal horoball on the first try,
 * but not so large that the canonization algorithm has to do a lot of
 * thrashing around (in particular, we don't want it to have to randomize
 * very often).
 */
#define DELTA_DISPLACEMENT          0.5

/*
 * If the longitudinal translation has length zero,
 * something has gone very, very wrong.
 */
#define LONGITUDE_EPSILON          1e-2

/*
 * contains_north_pole() uses NORTH_POLE_EPSILON to decide when a face
 * of a tetrahedron stands vertically over a vertex.
 */
#define NORTH_POLE_EPSILON          1e-6

/*
 * A complex number of modulus greater than KEY_INFINITY is considered
 * to be infinite, at least for the purpose of computing key values.
 */
#define KEY_INFINITY                1e+6

/*
 * tiling_tet_on_tree() will compare two TilingTets iff their key values
 * are within KEY_EPSILON of each other.  KEY_EPSILON can be fairly large;
 * other than a loss of speed there is no harm in having the program make
 * some occasional unnecessary comparisons.
 */
#define KEY_EPSILON                 1e-4

/*
 * Two TilingTets are considered equivalent under the Z + Z action of
 * the cusp translations iff their corresponding (transformed) corners
 * lie within CORNER_EPSILON of each other.
 */
#define CORNER_EPSILON              1e-6

/*
 * cull_duplicate_horoballs() checks whether two horoballs are equivalent
 * iff their radii differ by less than DUPLICATE_RADIUS_EPSILON.
 */
```

```

* We should make DUPLICATE_RADIUS_EPSILON fairly large, to be sure we
* don't miss any horoballs even when their precision is low.
*/
#define DUPLICATE_RADIUS_EPSILON    1e-3

typedef int MinDistanceType;
enum
{
    dist_self_to_self,
    dist_self_to_any,
    dist_group_to_group,
    dist_group_to_any
};

typedef struct
{
    Tetrahedron      *tet;
    Orientation      h;
    VertexIndex      v;
} CuspTriangle;

typedef struct TilingHoroball
{
    CuspNbhdHoroball data;
    struct TilingHoroball *next;
} TilingHoroball;

typedef struct TilingTet
{
    /*
    * Which Tetrahedron in the original manifold lifts to this TilingTet?
    */
    Tetrahedron      *underlying_tet;

    /*
    * Does it appear with the left_handed or right_handed orientation?
    */
    Orientation      orientation;

    /*
    * Where are its four corners on the boundary of upper half space?
    */
    Complex           corner[4];

    /*
    * What is the Euclidean diameter of the horoball at each corner?
    */
    double            horoball_height[4];

    /*
    * If the neighboring TilingTet incident to face f has already been
    * found, neighbor_found[f] is set to TRUE so we won't waste time
    * finding it again. More importantly, we won't have to worry about
    * the special case of "finding" the initial TilingTets incident to
    * the "horoball of infinite Euclidean radius".
    */
    Boolean           neighbor_found[4];

    /*
    * Pointer for the NULL-terminated queue.
    */
    struct TilingTet *next;

    /*
    * Pointers for the tree.
    */

    /*
    * The left child and right child pointers implement the binary tree.
    */
    struct TilingTet *left,
    *right;

```

```

/*
 * The sort key is a continuous function of the TilingTet's corners,
 * and is well defined under the Z + Z action of the group of
 * covering transformations of the cusp.
 */
double          key;

/*
 * We don't want our tree handling functions to be recursive,
 * for fear of stack/heap collisions. So we implement them using
 * our own private stack, which is a NULL-terminated linked list
 * using the next_subtree pointer. Unlike the "left" and "right"
 * fields (which are maintained throughout the algorithm) the
 * "next_subtree" field is used only locally within a given tree
 * handling function.
 */
struct TilingTet  *next_subtree;

} TilingTet;

typedef struct
{
    TilingTet      *begin,
                  *end;
} TilingQueue;

static void          initialize_cusp_displacements(CuspNeighborhoods *
    cusp_neighborhoods);
static void          compute_cusp_reaches(CuspNeighborhoods *cusp_neighborhoods);
static void          compute_one_reach(CuspNeighborhoods *cusp_neighborhoods, Cusp *
    cusp);
static void          compute_tie_group_reach(CuspNeighborhoods *cusp_neighborhoods);
static Cusp          *some_tied_cusp(CuspNeighborhoods *cusp_neighborhoods);
static void          compute_cusp_stoppers(CuspNeighborhoods *cusp_neighborhoods);
static void          compute_intercusp_distances(Triangulation *manifold);
static void          compute_one_intercusp_distance(EdgeClass *edge);
static void          compute_min_dist(Triangulation *manifold, Cusp *cusp,
    MinDistanceType min_distance_type);
static void          initialize_cusp_ties(CuspNeighborhoods *cusp_neighborhoods);
static void          initialize_cusp_nbhd_positions(CuspNeighborhoods *
    cusp_neighborhoods);
static void          allocate_cusp_nbhd_positions(CuspNeighborhoods *
    cusp_neighborhoods);
static void          compute_cusp_nbhd_positions(CuspNeighborhoods *
    cusp_neighborhoods);
static Boolean       contains_meridian(Tetrahedron *tet, Orientation h, VertexIndex
    v);
static void          set_one_component(Tetrahedron *tet, Orientation h, VertexIndex
    v, int max_triangles);
static CuspNbhdHoroballList *get_quick_horoball_list(CuspNeighborhoods *cusp_neighborhoods,
    Cusp *cusp);
static void          get_quick_edge_horoballs(Triangulation *manifold, Cusp *cusp,
    CuspNbhdHoroball **next_horoball);
static void          get_quick_face_horoballs(Triangulation *manifold, Cusp *cusp,
    CuspNbhdHoroball **next_horoball);
static CuspNbhdHoroballList *get_full_horoball_list(CuspNeighborhoods *cusp_neighborhoods,
    Cusp *cusp, double cutoff_height);
static void          compute_exp_min_d(Triangulation *manifold);
static void          compute_parallelogram_to_square(Complex meridian, Complex
    longitude, double parallelogram_to_square[2][2]);
static void          read_initial_tetrahedra(Triangulation *manifold, Cusp *cusp,
    TilingQueue *tiling_queue, TilingTet **tiling_tree_root, TilingHoroball **
    horoball_linked_list, double cutoff_height);
static TilingTet     *get_tiling_tet_from_queue(TilingQueue *tiling_queue);
static void          add_tiling_tet_to_queue(TilingTet *tiling_tet, TilingQueue *
    tiling_queue);
static void          add_tiling_horoball_to_list(TilingTet *tiling_tet, VertexIndex
    v, TilingHoroball **horoball_linked_list);
static Boolean       face_contains_useful_edge(TilingTet *tiling_tet, FaceIndex f,
    double cutoff_height);
static TilingTet     *make_neighbor_tiling_tet(TilingTet *tiling_tet, FaceIndex f);
static void          prepare_sort_key(TilingTet *tiling_tet, double

```

```

    parallelogram_to_square[2][2]);
static Boolean      tiling_tet_on_tree(TilingTet *tiling_tet, TilingTet *
    tiling_tree_root, Complex meridian, Complex longitude);
static Boolean      same_corners(TilingTet *tiling_tet1, TilingTet *tiling_tet2,
    Complex meridian, Complex longitude);
static void         add_tiling_tet_to_tree(TilingTet *tiling_tet, TilingTet **
    tiling_tree_root);
static void         add_horoball_if_necessary(TilingTet *tiling_tet, TilingHoroball **
    horoball_linked_list, double cutoff_height);
static Boolean      contains_north_pole(TilingTet *tiling_tet, VertexIndex v);
static void         free_tiling_tet_tree(TilingTet *tiling_tree_root);
static CuspNbhdHoroballList *transfer_horoballs(TilingHoroball **horoball_linked_list);
static int CDECL    compare_horoballs(const void *horoball0, const void *horoball1);
;
static void         cull_duplicate_horoballs(Cusp *cusp, CuspNbhdHoroballList *
    aHoroballList);

/*
 * Conceptually, the CuspNeighborhoods structure stores cross sections
 * of a manifold's cusps, and also keeps a Triangulation dual to the
 * corresponding Ford complex. In the present implementation, the
 * information about the cross sections is stored entirely within the
 * copy of the triangulation (specifically, in the Cusp's displacement,
 * displacement_exp and reach fields, the EdgeClass's inter_cusp_distance
 * field, and the Triangulation's max_reach field).
 *
 * SnapPea.h (the only header file common to the user interface and the
 * computational kernel) contains the opaque typedef
 *
 *     typedef struct CuspNeighborhoods      CuspNeighborhoods;
 *
 * This opaque typedef allows the user interface to declare and pass
 * a pointer to a CuspNeighborhoods structure, without being able to
 * access a CuspNeighborhoods structure's fields directly. Here is
 * the actual definition, which is private to this file.
 */

struct CuspNeighborhoods
{
    /*
     * We'll keep our own private copy of the Triangulation, to avoid
     * messing up the original one.
     */
    Triangulation    *its_triangulation;
};

/*
 * Technical musings.
 *
 * There are different approaches to maintaining a canonical
 * triangulation as the cusp displacements change.
 *
 * Low-level approach.
 *     Handle the 2-3 and 3-2 moves explicitly. Calculate which
 *     move will be required next as the given cusp moves towards
 *     the requested displacement.
 *
 * High-level approach.
 *     Set the requested cusp displacement directly, and call the
 *     standard proto_canonize() function to compute the corresponding
 *     canonical triangulation.
 *
 * The low-level approach would be much more efficient at run time.
 * The overhead of setting up the cusp cross sections at the beginning,
 * and polishing the hyperbolic structure at the end, would be done
 * only once. It would also be efficient in that it tracks the convex
 * hull (i.e. the canonical triangulation) precisely as the cusp moves
 * toward the requested displacement. (At each step it finds the next
 * 2-3 or 3-2 move which would be required as the cusp cross section
 * moves continuously towards the requested displacement.)
 *
 * The drawback of the low-level approach is that it would require
 * a lot of low-level programming, which is time consuming, tends to

```

```

* make a mess, and can be error prone. The high-level approach keeps
* the code cleaner, even though it's less efficient at run time.
*
* For now I have implemented the high-level approach. If it turns
* out that it is too slow, I can consider replacing it with the
* low-level approach. An even better approach might be to make
* some simple changes to speed up the high-level approach. For example,
* I was concerned that for large manifolds proto_canonize()'s bottleneck
* might be polishing the hyperbolic structure at the end. I modified
* proto_canonize() to polish the hyperbolic structure iff the
* triangulation has been changed.
*/

CuspNeighborhoods *initialize_cusp_neighborhoods(
    Triangulation *manifold)
{
    Triangulation *simplified_manifold;
    CuspNeighborhoods *cusp_neighborhoods;

    /*
     * If the space isn't a manifold, return NULL.
     */
    if (all_Dehn_coefficients_are_relatively_prime_integers(manifold) == FALSE)
        return NULL;

    /*
     * Get rid of "unnecessary" cusps.
     * If we encounter topological obstructions, return NULL.
     */
    simplified_manifold = fill_reasonable_cusps(manifold);
    if (simplified_manifold == NULL)
        return NULL;

    /*
     * If the manifold is closed, free it and return NULL.
     */
    if (all_cusps_are_filled(simplified_manifold) == TRUE)
    {
        free_triangulation(simplified_manifold);
        return NULL;
    }

    /*
     * Attempt to canonize the manifold.
     */
    if (proto_canonize(simplified_manifold) == func_failed)
    {
        free_triangulation(simplified_manifold);
        return NULL;
    }

    /*
     * Our manifold has passed all its tests,
     * so set up a CuspNeighborhoods structure.
     */
    cusp_neighborhoods = NEW_STRUCT(CuspNeighborhoods);

    /*
     * Install our private copy of the triangulation.
     */
    cusp_neighborhoods->its_triangulation = simplified_manifold;
    simplified_manifold = NULL;

    /*
     * Most likely the displacements will be zero already,
     * but we set them anyhow, just to be safe.
     */
    initialize_cusp_displacements(cusp_neighborhoods);

    /*
     * Compute all cusp reaches.
     */
    compute_cusp_reaches(cusp_neighborhoods);
}

```

```

/*
 * Find the stoppers.
 */
compute_cusp_stoppers(cusp_neighborhoods);

/*
 * Initially no cusps are tied.
 */
initialize_cusp_ties(cusp_neighborhoods);

/*
 * Set up an implicit coordinate system on each cusp cross section
 * so that we can report the position of horoballs etc. consistently,
 * even as the canonical triangulation changes.
 */
initialize_cusp_nbhd_positions(cusp_neighborhoods);

/*
 * Record the volume so we don't have to recompute it
 * over and over in real time.
 */
cusp_neighborhoods->its_triangulation->volume = volume(cusp_neighborhoods->
its_triangulation, NULL);

/*
 * Done.
 */
return cusp_neighborhoods;
}

void free_cusp_neighborhoods(
    CuspNeighborhoods *cusp_neighborhoods)
{
    if (cusp_neighborhoods != NULL)
    {
        free_triangulation(cusp_neighborhoods->its_triangulation);
        my_free(cusp_neighborhoods);
    }
}

static void initialize_cusp_displacements(
    CuspNeighborhoods *cusp_neighborhoods)
{
    Cusp *cusp;

    for (cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
         cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
         cusp = cusp->next)
    {
        cusp->displacement = 0.0;
        cusp->displacement_exp = 1.0;
    }
}

static void compute_cusp_reaches(
    CuspNeighborhoods *cusp_neighborhoods)
{
    Cusp *cusp;

    cusp_neighborhoods->its_triangulation->max_reach = 0.0;

    for (cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
         cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
         cusp = cusp->next)
    {
        compute_one_reach(cusp_neighborhoods, cusp);

        if (cusp->reach > cusp_neighborhoods->its_triangulation->max_reach)
            cusp_neighborhoods->its_triangulation->max_reach = cusp->reach;
    }
}

```



```

}

static void compute_one_reach(
    CuspNeighborhoods *cusp_neighborhoods,
    Cusp *cusp)
{
    /*
     * The key observation is the following. Think of a horoball
     * packing corresponding to the cusp cross sections in their home
     * positions, with the given cusp lifting to the plane  $z = 1$  in
     * the upper half space model. The vertical line passing through
     * the top of a maximally (Euclidean-)large round horoball is
     * guaranteed to be an edge in the canonical triangulation.
     * (Proof: As the horoballs expand equivariantly, the largest round
     * horoball(s) is(are) the first one(s) to touch the  $z = 1$  horoball.)
     * So by measuring the distance between cusp cross sections along the
     * edges of the canonical triangulation, we can deduce the distance
     * from the given cusp to the largest round horoball(s). If a largest
     * round horoball corresponds to the given cusp, then we know the
     * cusp's reach and we're done. If the largest horoballs all belong
     * to other cusps, then we retract the other cusps a bit (i.e. give
     * them a negative displacement) and try again. Eventually a horoball
     * corresponding to the given cusp will be maximal.
     */

    Triangulation *triangulation_copy;
    Cusp *cusp_copy,
        *other_cusp;
    double dist_any,
        dist_self;

    /*
     * Make a copy of the triangulation, so we don't disturb the original.
     */
    copy_triangulation(cusp_neighborhoods->its_triangulation, &triangulation_copy);
    cusp_copy = find_cusp(triangulation_copy, cusp->index);

    /*
     * Carry out the algorithm described above.
     */
    while (TRUE)
    {
        /*
         * Compute the distances between cusp cross sections along each
         * edge of the (already canonical) triangulation, and store the
         * results in the EdgeClass's intercusp_distance field.
         *
         * Technical note: There is a small inefficiency here in that
         * proto_canonize() creates and discards the cusp cross sections,
         * and here we create and discard them again. If this turns out
         * to be a problem we could have proto_canonize() compute the
         * intercusp distances when it does the canonization, but for
         * now I'll put up with the inefficiency to keep the code clean.
         */
        compute_intercusp_distances(triangulation_copy);

        /*
         * Does a maximally large round horoball belong to the given cusp?
         * If so, we know the reach and we're done.
         */
        dist_self = compute_min_dist(triangulation_copy, cusp_copy, dist_self_to_self);
        dist_any = compute_min_dist(triangulation_copy, cusp_copy, dist_self_to_any);
        if (dist_self < dist_any + INTERCUSP_EPSILON)
        {
            cusp->reach = 0.5 * dist_self;
            break;
        }

        /*
         * Otherwise, retract all cross sections except the given one,
         * recanonize, and continue with the loop.
         *
         * Note: initialize_cusp_neighborhoods() has already checked

```

```

    * that the manifold is hyperbolic, so proto_canonize() should
    * not fail.
    */

    for (other_cusp = triangulation_copy->cusp_list_begin.next;
         other_cusp != &triangulation_copy->cusp_list_end;
         other_cusp = other_cusp->next)

        if (other_cusp != cusp_copy)
        {
            other_cusp->displacement -= DELTA_DISPLACEMENT;
            other_cusp->displacement_exp = exp(other_cusp->displacement);
        }

    if (proto_canonize(triangulation_copy) != func_OK)
        uFatalError("compute_one_reach", "cusp_neighborhoods.c");
}

/*
 * Free the copy of the triangulation.
 */
free_triangulation(triangulation_copy);
}

static void compute_tie_group_reach(
    CuspNeighborhoods *cusp_neighborhoods)
{
    /*
     * This function is similar to compute_one_reach(), but instead of
     * computing the reach of a single cusp, it computes the reach of
     * a group of tied cusps (that is a group of cusp neighborhoods which
     * move forward and backward in unison). Please see compute_one_reach()
     * above for detailed documentation.
     */

    Triangulation *triangulation_copy;
    double dist_any,
            dist_self;
    Cusp *cusp;

    /*
     * If no cusps are tied, there is nothing to be done.
     */
    if (some_tied_cusp(cusp_neighborhoods) == NULL)
    {
        cusp_neighborhoods->its_triangulation->tie_group_reach = 0.0;
        return;
    }

    /*
     * Make a copy of the triangulation, so we don't disturb the original.
     * copy_triangulation() copies the is_tied field, even though it is
     * in some sense private to this file.
     */
    copy_triangulation(cusp_neighborhoods->its_triangulation, &triangulation_copy);

    /*
     * Carry out the algorithm described in compute_one_reach().
     */

    while (TRUE)
    {
        compute_intercusp_distances(triangulation_copy);

        dist_self = compute_min_dist(triangulation_copy, NULL, dist_group_to_group);
        dist_any = compute_min_dist(triangulation_copy, NULL, dist_group_to_any);

        if (dist_self < dist_any + INTERCUSP_EPSILON)
        {
            cusp_neighborhoods->its_triangulation->tie_group_reach
                = some_tied_cusp(cusp_neighborhoods)->displacement
                  + 0.5 * dist_self;
            break;
        }
    }
}

```

```

    }

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

        if (cusp->is_tied == FALSE)
        {
            cusp->displacement -= DELTA_DISPLACEMENT;
            cusp->displacement_exp = exp(cusp->displacement);
        }

    if (proto_canonize(triangulation_copy) != func_OK)
        uFatalError("compute_tie_group_reach", "cusp_neighborhoods.c");
}

free_triangulation(triangulation_copy);
}

static Cusp *some_tied_cusp(
    CuspNeighborhoods *cusp_neighborhoods)
{
    Cusp *cusp;

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

        if (cusp->is_tied)

            return cusp;

    return NULL;
}

static void compute_cusp_stoppers(
    CuspNeighborhoods *cusp_neighborhoods)
{
    /*
     * Think of a horoball packing corresponding to the cusp cross sections
     * in their current positions, with a given cusp lifting to the plane
     * z == 1 in the upper half space model. The vertical line passing
     * through the top of a maximally (Euclidean-)large round horoball is
     * guaranteed to be an edge in the canonical cell decomposition.
     * (Proof: As the horoballs expand equivariantly, the largest
     * round horoballs will be the first to touch the z == 1 horoball.)
     *
     * Case 1. The maximal horoball belongs to the given cusp.
     *
     * In this case, the given cusp is its own stopper, and the
     * stopping displacement is its reach.
     *
     * Case 2. The maximal horoball belongs to some other cusp.
     *
     * The displacement at which the given cusp meets the other cusp
     * may or may not be less than the given cusp's reach.
     * (A less-than-maximal horoball belonging to the given cusp may
     * overtake a formerly maximal cusp, because horoballs belonging
     * to the given cusp grow as the given cusp moves forward, while
     * other horoballs do not.) If the stopping displacement is
     * less than the given cusp's reach, then we've found a stopper
     * cusp and stopping displacement (the stopping displacement is
     * unique, even though the stopper cusp may not be). If the
     * stopping is greater than or equal to the given cusp's reach,
     * then the cusp is its own stopper, as in case 1.
     */

    Cusp *cusp,
          *c[2];
    EdgeClass *edge;
    int i;
    double possible_stopping_displacement;

```

```

/*
 * Initialize each stopper to be the cusp itself, and the stopping
 * displacement to be its reach.
 */

for (cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
     cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
     cusp = cusp->next)
{
    cusp->stopper_cusp = cusp;
    cusp->stopping_displacement = cusp->reach;
}

/*
 * Now look at each edge of the canonical triangulation, to see
 * whether some other cusp cross section is closer.
 *
 * cusp_neighborhoods->its_triangulation is always the canonical
 * triangulation (or an arbitrary subdivision of the canonical
 * cell decomposition).
 */

compute_intercusp_distances(cusp_neighborhoods->its_triangulation);

for (edge = cusp_neighborhoods->its_triangulation->edge_list_begin.next;
     edge != &cusp_neighborhoods->its_triangulation->edge_list_end;
     edge = edge->next)
{
    c[0] = edge->incident_tet->cusp[ one_vertex_at_edge[edge->incident_edge_index]];
    c[1] = edge->incident_tet->cusp[other_vertex_at_edge[edge->incident_edge_index]];

    for (i = 0; i < 2; i++)
    {
        possible_stopping_displacement =
            c[i]->displacement + edge->intercusp_distance;

        if (possible_stopping_displacement < c[i]->stopping_displacement)
        {
            c[i]->stopping_displacement = possible_stopping_displacement;
            c[i]->stopper_cusp = c[!i];
        }
    }
}

static void compute_intercusp_distances(
    Triangulation *manifold)
{
    /*
     * In the present context we may assume the triangulation is
     * canonical (although all we really need to know is that it
     * has a geometric_solution).
     */

    EdgeClass *edge;

    /*
     * Set up the cusp cross sections.
     */
    allocate_cross_sections(manifold);
    compute_cross_sections(manifold);

    /*
     * Compute the intercusp_distances.
     */

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

        compute_one_intercusp_distance(edge);
}

```

```

/*
 * Release the cusp cross sections.
 */
free_cross_sections(manifold);
}

static void compute_one_intercusp_distance(
    EdgeClass *edge)
{
    int i,
        j;
    Tetrahedron *tet;
    EdgeIndex e;
    VertexIndex v[2];
    FaceIndex f[2];
    double length[2][2],
        product;

/*
 * Find an arbitrary Tetrahedron incident to the given EdgeClass.
 */
tet = edge->incident_tet;
e = edge->incident_edge_index;

/*
 * Note which vertices and faces are incident to the EdgeClass.
 */
v[0] = one_vertex_at_edge[e];
v[1] = other_vertex_at_edge[e];
f[0] = one_face_at_edge[e];
f[1] = other_face_at_edge[e];

/*
 * The vertex cross section at each vertex v[i] is a triangle.
 * Note the lengths of the triangle's edges incident to the EdgeClass.
 */
for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
        length[i][j] = tet->cusp[v[i]]->displacement_exp
            * tet->cross_section->edge_length[v[i]][f[j]];

/*
 * Our task is to compute the distance between the vertex cross sections
 * as a function of the length[]'s. Fortunately this is easier than
 * you might expect. I recommend you make sketches for yourself as
 * you read through the following. (It's much simpler in pictures
 * than it is in words.)
 *
 * Proposition. There is a unique common perpendicular to a pair of
 * opposite edges of an ideal tetrahedron.
 *
 * Proof. Consider the line segment which minimizes the distance
 * between the two opposite edges. If it weren't perpendicular to
 * each edge, then a shorter line segment could be found. QED
 *
 * Definition. The "midpoint" of an edge of an ideal tetrahedron is
 * the point where the edge intersects the unique common perpendicular
 * to the opposite edge.
 *
 * Proposition. A half turn about the aforementioned unique common
 * perpendicular is a symmetry of the ideal tetrahedron.
 *
 * Proof. It preserves (setwise) a pair of opposite edges. Therefore
 * it preserves (setwise) the tetrahedron's four ideal vertices, and
 * therefore the whole tetrahedron. QED
 *
 * Proposition. Consider a vertex cross section which passes through
 * the midpoint of an edge. The two sides of the vertex cross section
 * which are incident to the given edge of the tetrahedron have lengths
 * which are reciprocals of one another.
 *
 * Proof. Position the tetrahedron in the upper half space so
 * that the given edge is vertical and its midpoint is at height one.

```

```

*
* Let P1 be the unique plane which contains the aforementioned unique
* common perpendicular and also contains the edge itself.
*
* Let P2 be the unique plane which contains the aforementioned unique
* common perpendicular and is orthogonal to the edge itself.
*
* Let S be the symmetry defined by a reflection in P1 followed by a
* reflection in P2.
*
* S is equivalent to a half turn about the unique common perpendicular
* (proof: P1 and P2 are orthogonal to each other, and both contain
* the unique common perpendicular). Therefore S is a symmetry of the
* ideal tetrahedron, by the preceding proposition.
*
* Let L1 and L2 be the lengths of the two sides of the vertex cross
* section which are incident to the given edge. Because the vertex
* cross section is at height one in the upper half space model,
* L1 and L2 also represent the Euclidean lengths of two sides of the
* triangle obtained by projecting the ideal tetrahedron onto the
* plane  $z = 0$  in the upper half space model. Reflection in the
* plane P1 does not change the lengths of those two sides of the
* triangle, while reflection in the plane P2 (which, in Euclidean
* terms, is inversion in a hemisphere of radius one) sends each
* length to its inverse. Since the composition S of the two
* reflections preserves the triangle, it follows that L1 and L2
* must be inverses of one another. QED
*
* If a vertex cross section passes through the midpoint of an edge,
* then the product of the lengths L1 and L2 (using the notation of
* the preceding proof) is  $L1 L2 = 1$ . Now consider a vertex cross
* section which is a distance d away from the midpoint (towards
* the fat part of the manifold if d is positive, towards the cusp
* if d is negative). According to the documentation at the top of
* this file, a cusp cross section's linear dimensions vary as  $\exp(d)$ ,
* so the lengths of the corresponding sides of the new vertex cross
* section will be  $\exp(d)L1$  and  $\exp(d)L2$ . Their product is
*  $\exp(d)L1 \exp(d)L2 = \exp(2d) L1 L2 = \exp(2d)$ .
*
* If the lengths of the sides of the vertex cross section at the
* other end of the given edge are  $\exp(d')L1$  and  $\exp(d')L2$ , then
* their product is  $\exp(2d')$ . The product of all four lengths is
*
*  $\exp(d)L1 \exp(d)L2 \exp(d')L1 \exp(d')L2 = \exp(2(d + d'))$ .
*
* This is exactly what we need to know:  $d + d'$  is the negative
* of the intercusp distance. (Note that the midpoint has dropped
* out of the picture!)
*/

product = 1.0;
for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
        product *= length[i][j];

edge->intercusp_distance = -0.5 * log(product);
}

static double compute_min_dist(
    Triangulation *manifold,
    Cusp *cusp, /* ignored for tie group distances */
    MinDistanceType min_distance_type)
{
    /*
     * This function assumes the intercusp_distances
     * have already been computed.
     */

    double min_dist;
    EdgeClass *edge;
    Cusp *cusp1,
        *cusp2;

```

```

    min_dist = DBL_MAX;

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)
    {
        cusp1 = edge->incident_tet->cusp[ one_vertex_at_edge[edge->incident_edge_index]];
        cusp2 = edge->incident_tet->cusp[other_vertex_at_edge[edge->incident_edge_index]];

        if (edge->intercusp_distance < min_dist)
        {
            switch (min_distance_type)
            {
                case dist_self_to_self:
                    if (cusp == cusp1 && cusp == cusp2)
                        min_dist = edge->intercusp_distance;
                    break;

                case dist_self_to_any:
                    if (cusp == cusp1 || cusp == cusp2)
                        min_dist = edge->intercusp_distance;
                    break;

                case dist_group_to_group:
                    if (cusp1->is_tied && cusp2->is_tied)
                        min_dist = edge->intercusp_distance;
                    break;

                case dist_group_to_any:
                    if (cusp1->is_tied || cusp2->is_tied)
                        min_dist = edge->intercusp_distance;
                    break;
            }
        }

        return min_dist;
    }
}

int get_num_cusp_neighborhoods(
    CuspNeighborhoods *cusp_neighborhoods)
{
    if (cusp_neighborhoods == NULL)
        return 0;
    else
        return get_num_cusps(cusp_neighborhoods->its_triangulation);
}

CuspTopology get_cusp_neighborhood_topology(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index)
{
    return find_cusp(cusp_neighborhoods->its_triangulation, cusp_index)->topology;
}

double get_cusp_neighborhood_displacement(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index)
{
    return find_cusp(cusp_neighborhoods->its_triangulation, cusp_index)->displacement;
}

Boolean get_cusp_neighborhood_tie(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index)
{
    return find_cusp(cusp_neighborhoods->its_triangulation, cusp_index)->is_tied;
}

double get_cusp_neighborhood_cusp_volume(

```

```

    CuspNeighborhoods    *cusp_neighborhoods,
    int                  cusp_index)
{
    /*
     * As explained in the documentation at the top of this file,
     * the volume will be the volume enclosed by the cusp in its
     * home position, multiplied by exp(2 * displacement).
     */

    return 0.1875 * ROOT_3 * exp(2 * find_cusp(cusp_neighborhoods->its_triangulation,
    cusp_index)->displacement);
}

double get_cusp_neighborhood_manifold_volume(
    CuspNeighborhoods    *cusp_neighborhoods)
{
    return cusp_neighborhoods->its_triangulation->volume;
}

Triangulation *get_cusp_neighborhood_manifold(
    CuspNeighborhoods    *cusp_neighborhoods)
{
    Triangulation    *manifold_copy;
    Cusp              *cusp;

    /*
     * Make a copy of its_triangulation.
     */
    copy_triangulation(cusp_neighborhoods->its_triangulation, &manifold_copy);

    /*
     * Reset the cusp displacements to zero, so if a canonical triangulation
     * is needed later it will be computed relative to cusp cross sections
     * of equal volume.
     */
    for (cusp = manifold_copy->cusp_list_begin.next;
         cusp != &manifold_copy->cusp_list_end;
         cusp = cusp->next)
    {
        cusp->displacement      = 0.0;
        cusp->displacement_exp  = 1.0;
    }

    return manifold_copy;
}

double get_cusp_neighborhood_reach(
    CuspNeighborhoods    *cusp_neighborhoods,
    int                  cusp_index)
{
    return find_cusp(cusp_neighborhoods->its_triangulation, cusp_index)->reach;
}

double get_cusp_neighborhood_max_reach(
    CuspNeighborhoods    *cusp_neighborhoods)
{
    return cusp_neighborhoods->its_triangulation->max_reach;
}

double get_cusp_neighborhood_stopping_displacement(
    CuspNeighborhoods    *cusp_neighborhoods,
    int                  cusp_index)
{
    return find_cusp(cusp_neighborhoods->its_triangulation, cusp_index)->
    stopping_displacement;
}

int get_cusp_neighborhood_stopper_cusp_index(

```



```

    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index)
{
    return find_cusp(cusp_neighborhoods->its_triangulation, cusp_index)->stopper_cusp->
    index;
}

void set_cusp_neighborhood_displacement(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index,
    double new_displacement)
{
    Cusp *cusp,
        *other_cusp;

    /*
     * Get a pointer to the cusp whose displacement is being changed.
     */

    cusp = find_cusp(cusp_neighborhoods->its_triangulation, cusp_index);

    /*
     * Clip the displacement to the feasible range.
     */

    if (new_displacement < 0.0)
        new_displacement = 0.0;

    if (cusp->is_tied == FALSE)
    {
        /*
         * The stopping_displacement has already been set to be less than or
         * equal to the reach, so by clipping to the stopping_displacement
         * we know the cusp neighborhood won't overlap itself or any
         * other cusp neighborhood.
         */

        if (new_displacement > cusp->stopping_displacement)
            new_displacement = cusp->stopping_displacement;
    }
    else /* cusp->is_tied == TRUE */
    {
        /*
         * Make sure the new_displacement doesn't exceed the tie_group_reach.
         * Other cusps in the tie group will be coming at us as we move
         * toward them, so collisions might not be detected by the
         * stopping_displacement alone. (The latter assumes the other
         * cusp is stationary.)
         */

        if (new_displacement > cusp_neighborhoods->its_triangulation->tie_group_reach)
            new_displacement = cusp_neighborhoods->its_triangulation->tie_group_reach;

        /*
         * Don't overlap untied stoppers either.
         */

        for (other_cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
             other_cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
             other_cusp = other_cusp->next)

            if (other_cusp->is_tied
                && new_displacement > other_cusp->stopping_displacement)

                new_displacement = other_cusp->stopping_displacement;
    }

    /*
     * Set the new displacement.
     */

    if (cusp->is_tied == FALSE)

```

```

{
    cusp->displacement      = new_displacement;
    cusp->displacement_exp  = exp(new_displacement);
}
else /* cusp->is_tied == TRUE */
{
    for (other_cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
         other_cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
         other_cusp = other_cusp->next)

        if (other_cusp->is_tied)
        {
            other_cusp->displacement      = new_displacement;
            other_cusp->displacement_exp  = exp(new_displacement);
        }
}

/*
 * Compute the canonical cell decomposition
 * relative to the new displacement.
 */

if (proto_canonize(cusp_neighborhoods->its_triangulation) != func_OK)
    uFatalError("set_cusp_neighborhood_displacement", "cusp_neighborhoods");

/*
 * The cusp reaches won't have changed, but the stoppers might have.
 */

compute_cusp_stoppers(cusp_neighborhoods);
}

void set_cusp_neighborhood_tie(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index,
    Boolean new_tie)
{
    Cusp *cusp,
          *other_cusp;
    double min_displacement;

    /*
     * Get a pointer to the cusp which is being tied or untied.
     */
    cusp = find_cusp(cusp_neighborhoods->its_triangulation, cusp_index);

    /*
     * Tie or untie the cusp.
     */
    cusp->is_tied = new_tie;

    /*
     * If the cusp is being tied, bring it and its mates into line.
     */

    if (cusp->is_tied == TRUE)
    {
        /*
         * Find the minimum displacement for a tied cusp . . .
         */

        min_displacement = DBL_MAX;

        for (other_cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
             other_cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
             other_cusp = other_cusp->next)

            if (other_cusp->is_tied && other_cusp->displacement < min_displacement)

                min_displacement = other_cusp->displacement;

        /*
         * . . . and set all tied cusps to that minimum value.
         */
    }
}

```

```

    */

    for (other_cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
         other_cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
         other_cusp = other_cusp->next)

        if (other_cusp->is_tied)
        {
            other_cusp->displacement      = min_displacement;
            other_cusp->displacement_exp  = exp(min_displacement);
        }

    /*
     * Compute the canonical cell decomposition
     * relative to the minimum displacement.
     */
    if (proto_canonize(cusp_neighborhoods->its_triangulation) != func_OK)
        uFatalError("set_cusp_neighborhood_tie", "cusp_neighborhoods");

    /*
     * The cusp reaches won't have changed,
     * but the stoppers might have.
     */
    compute_cusp_stoppers(cusp_neighborhoods);
}

/*
 * How far can the group of tied cusps go before bumping into itself?
 */
compute_tie_group_reach(cusp_neighborhoods);
}

static void initialize_cusp_ties(
    CuspNeighborhoods *cusp_neighborhoods)
{
    Cusp *cusp;

    /*
     * Initially no cusps are tied . . .
     */
    for (cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
         cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
         cusp = cusp->next)

        cusp->is_tied = FALSE;

    /*
     * . . . and the tie_group_reach is undefined.
     */
    cusp_neighborhoods->its_triangulation->tie_group_reach = 0.0;
}

static void initialize_cusp_nbhd_positions(
    CuspNeighborhoods *cusp_neighborhoods)
{
    /*
     * Install VertexCrossSections so that we know the size of each
     * vertex cross section in the cusp's home position.
     */
    allocate_cross_sections(cusp_neighborhoods->its_triangulation);
    compute_cross_sections(cusp_neighborhoods->its_triangulation);

    /*
     * Allocate storage for the CuspNbhdPositions . . .
     */
    allocate_cusp_nbhd_positions(cusp_neighborhoods);

    /*
     * . . . and then compute them.
     */
    compute_cusp_nbhd_positions(cusp_neighborhoods);
}

```

```

/*
 * Free the VertexCrossSections now that we're done with them.
 * (proto_canonize()) will of course need them again, but it likes
 * to allocate them for itself -- this keeps its interaction with
 * the rest of the kernel cleaner.)
 */
free_cross_sections(cusp_neighborhoods->its_triangulation);
}

static void allocate_cusp_nbhd_positions(
    CuspNeighborhoods *cusp_neighborhoods)
{
    Tetrahedron *tet;

    for (tet = cusp_neighborhoods->its_triangulation->tet_list_begin.next;
         tet != &cusp_neighborhoods->its_triangulation->tet_list_end;
         tet = tet->next)
    {
        /*
         * Just for good measure, make sure no CuspNbhdPositions
         * are already allocated.
         */
        if (tet->cusp_nbhd_position != NULL)
            uFatalError("allocate_cusp_nbhd_positions", "cusp_neighborhoods");

        /*
         * Allocate a CuspNbhdPosition structure.
         */
        tet->cusp_nbhd_position = NEW_STRUCT(CuspNbhdPosition);
    }
}

static void compute_cusp_nbhd_positions(
    CuspNeighborhoods *cusp_neighborhoods)
{
    Tetrahedron *tet;
    Orientation h;
    VertexIndex v;
    int max_triangles;
    Cusp *cusp;
    PeripheralCurve c;
    Complex (*x)[4][4],
            *translation;
    Boolean (*in_use)[4];
    FaceIndex f,
              f0,
              f1,
              f2;

    int strands1,
        strands2,
        flow;

    double length;
    Complex factor;

    /*
     * Initialize all the tet->in_use[][] fields to FALSE,
     * and all tet->x[][][] to Zero.
     */

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

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

            for (v = 0; v < 4; v++)
            {
                for (f = 0; f < 4; f++)
                    tet->cusp_nbhd_position->x[h][v][f] = Zero;

                tet->cusp_nbhd_position->in_use[h][v] = FALSE;
            }
}

```

```

/*
 * For each vertex cross section which has not yet been set, set the
 * positions of its three vertices, and then recursively set the
 * positions of neighboring vertex cross sections. The positions
 * are relative to each cusp cross section's home position.
 * (Recall that initialize_cusp_nbhd_positions() has already called
 * compute_cross_sections() for us.) For torus cusps, do only the
 * sheet of the double cover which contains the peripheral curves
 * (this will be the right_handed sheet if the manifold is orientable).
 */

max_triangles = 2 * 4 * cusp_neighborhoods->its_triangulation->num_tetrahedra;

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

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

        if (tet->cusp_nbhd_position->in_use[right_handed][v] == FALSE
            && tet->cusp_nbhd_position->in_use[ left_handed][v] == FALSE)
        {
            /*
             * Use the sheet which contains the peripheral curves.
             * If neither does, do nothing for now. They'll show
             * up eventually.
             */

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

                if (contains_meridian(tet, h, v) == TRUE)
                {
                    set_one_component(tet, h, v, max_triangles);
                    break;
                }
        }

/*
 * Compute the meridional and longitudinal translation on each
 * cusp cross section. For Klein bottle cusps, the longitude
 * will actually be that of the double cover. The translations
 * are stored in the Cusp data structure as translation[M] and
 * translation[L].
 */

/*
 * The Algorithm
 *
 * The calls to set_one_component() have assigned coordinates to all
 * the triangles in the induced triangulation of the cusp cross section.
 * The problem is that these coordinates are well defined only up
 * to translations in the covering transformation group (or the
 * orientation preserving subgroup, in the case of a Klein bottle cusp).
 * So we want an algorithm which uses only the local coordinates within
 * each triangle, without requiring global consistency.
 *
 * Imagine following a peripheral curve around the cusp cross section,
 * and look at the sides of the triangles it passes through. As we
 * go along, we can keep track of the coordinates of the left and
 * right hand edges. When we "veer left" the left hand endpoint stays
 * constant, while the right hand endpoint moves forward, and vice
 * versa when we "veer right". By adding up all the displacements to
 * each endpoint, by the time we get back to our starting point we will
 * have computed the total translation along the curve. Actually,
 * it suffices to compute the total displacement for only one endpoint
 * (left or right) since both will give the same answer.
 *
 * Finally, note that it doesn't matter in what order we sum the
 * displacements. We can just iterate through all tetrahedra in the
 * triangulation without explicitly tracing curves.
 */

```

```

/* Initialize all translations to (0.0, 0.0), and then . . .
*/

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

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

        cusp->translation[c] = Zero;

/*
 * . . . add in the contribution of each piece of each curve.
 */

for (tet = cusp_neighborhoods->its_triangulation->tet_list_begin.next;
     tet != &cusp_neighborhoods->its_triangulation->tet_list_end;
     tet = tet->next)
{
    x      = tet->cusp_nbhd_position->x;
    in_use = tet->cusp_nbhd_position->in_use;

    for (v = 0; v < 4; v++)
    {
        cusp = tet->cusp[v];

        for (c = 0; c < 2; c++)
        {
            translation = &cusp->translation[c];

            for (f0 = 0; f0 < 4; f0++)
            {
                if (f0 == v)
                    continue;

                /*
                 * Relative to the right_handed Orientation, the faces
                 * f0, f1 and f2 are arranged around the ideal vertex v
                 * like this
                 *
                 *          /\
                 *         /  \ f0
                 *        /    \
                 *       /____\
                 *      /      \
                 *     f1      f2
                 *
                 * The triangles corners inherit the indices of the
                 * opposite sides.
                 */
                f1 = remaining_face[f0][v];
                f2 = remaining_face[v][f0];

                for (h = 0; h < 2; h++) /* h = right_handed, left_handed */
                {
                    if (in_use[h][v] == FALSE)
                        continue;

                    strands1 = tet->curve[c][h][v][f1];
                    strands2 = tet->curve[c][h][v][f2];

                    flow = FLOW(strands2, strands1);

                    /*
                     * We're interested only in displacements of the
                     * left hand endpoint (cf. above), which occur when
                     * the flow is negative (if h == right_handed) or
                     * the flow is positive (if h == left_handed).
                     */
                    if ((h == right_handed) ? (flow < 0) : (flow > 0))
                        *translation = complex_plus(
                            *translation,
                            complex_real_mult(
                                flow,
                                complex_minus(x[h][v][f2], x[h][v][f1])));
                }
            }
        }
    }
}

```

```

    }
}

/*
 * Rotate the coordinates so that the longitudes point in the
 * direction of the positive x-axis.
 */

/*
 * Find the rotation needed for each cusp,
 * and use it to rotate the meridian and longitude.
 */

for (cusp = cusp_neighborhoods->its_triangulation->cusp_list_begin.next;
     cusp != &cusp_neighborhoods->its_triangulation->cusp_list_end;
     cusp = cusp->next)
{
    cusp->scratch = cusp->translation[L];
    length = complex_modulus(cusp->scratch);
    if (length < LONGITUDE_EPSILON)
        uFatalError("compute_cusp_nbhd_positions", "cusp_neighborhoods");
    cusp->scratch = complex_real_mult(1.0/length, cusp->scratch);
    cusp->scratch = complex_div(One, cusp->scratch);

    cusp->translation[M] = complex_mult(cusp->scratch, cusp->translation[M]);
    cusp->translation[L] = complex_mult(cusp->scratch, cusp->translation[L]);

    cusp->translation[L].imag = 0.0;    /* kill the roundoff error */
}

/*
 * Use the same rotation (stored in cusp->scratch) to rotate
 * the coordinates in the triangulation of the cusp.
 */

for (tet = cusp_neighborhoods->its_triangulation->tet_list_begin.next;
     tet != &cusp_neighborhoods->its_triangulation->tet_list_end;
     tet = tet->next)
{
    x      = tet->cusp_nbhd_position->x;
    in_use = tet->cusp_nbhd_position->in_use;

    for (h = 0; h < 2; h++)    /* h = right_handed, left_handed */
    {
        for (v = 0; v < 4; v++)
        {
            if (in_use[h][v] == FALSE)
                continue;

            factor = tet->cusp[v]->scratch;

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

                x[h][v][f] = complex_mult(factor, x[h][v][f]);
            }
        }
    }
}

static Boolean contains_meridian(
    Tetrahedron    *tet,
    Orientation     h,
    VertexIndex     v)
{
    /*
     * It suffices to check any two sides, because the meridian
     * can't possibly intersect only one side of a triangle.
     * (These are signed intersection numbers.)
     */
}

```

```

    */

    VertexIndex w0,
                w1;

    w0 = ! v;
    w1 = remaining_face[v][w0];

    return (tet->curve[M][h][v][w0] != 0
           || tet->curve[M][h][v][w1] != 0);
}

static void set_one_component(
    Tetrahedron      *tet,
    Orientation       h,
    VertexIndex       v,
    int               max_triangles)
{
    /*
     *      FaceIndices are the natural way to index the corners
     *      of a vertex cross section.
     *
     *      The VertexIndex v tells which vertex cross section we're at.
     *      The vertex cross section is (a triangular component of) the
     *      intersection of a cusp cross section with the ideal tetrahedron.
     *      Each side of the triangle is the intersection of the cusp cross
     *      section with some face of the ideal tetrahedron, so FaceIndices
     *      may naturally be used to index them.  Each corner of the triangle
     *      then inherits the FaceIndex of the opposite side.
     */

    FaceIndex         f[3],
                      ff,
                      nbr_f[3];
    int               i;
    CuspTriangle       *queue,
                      tri,
                      nbr;

    int               queue_begin,
                      queue_end;
    Permutation        gluing;
    CuspNbhdPosition   *our_data,
                      *nbr_data;

    /*
     *      Find the three FaceIndices for the corners of the triangle.
     *      (f == v is excluded.)
     */
    for ( i = 0, ff = 0;
          i < 3;
          i++, ff++)
    {
        if (ff == v)
            ff++;
        f[i] = ff;
    }

    /*
     *      Let the corner f[0] be at the origin.
     */
    tet->cusp_nbhd_position->x[h][v][f[0]] = Zero;

    /*
     *      Let the corner f[1] be on the positive x-axis.
     */
    tet->cusp_nbhd_position->x[h][v][f[1]].real = tet->cross_section->edge_length[v][f[2]];
    tet->cusp_nbhd_position->x[h][v][f[1]].imag = 0.0;

    /*
     *      Use the TetShape to find the position of corner f[2].
     */
    cn_find_third_corner(tet, h, v, f[0], f[1], f[2]);
}

```



```

/*
 * Mark this triangle as being in_use.
 */
tet->cusp_nbhd_position->in_use[h][v] = TRUE;

/*
 * We'll now "recursively" set the remaining triangles of this
 * cusp cross section. We'll keep a queue of the triangles whose
 * positions have been set, but whose neighbors have not yet
 * been examined.
 */

queue = NEW_ARRAY(max_triangles, CuspTriangle);

queue[0].tet    = tet;
queue[0].h      = h;
queue[0].v      = v;

queue_begin = 0;
queue_end   = 0;

while (queue_begin <= queue_end)
{
    /*
     * Pull a CuspTriangle off the queue.
     */
    tri = queue[queue_begin++];

    /*
     * Consider each of its three neighbors.
     */
    for (ff = 0; ff < 4; ff++)
    {
        if (ff == tri.v)
            continue;

        gluing = tri.tet->gluing[ff];

        nbr.tet = tri.tet->neighbor[ff];
        nbr.h    = (parity[gluing] == orientation_preserving) ? tri.h : ! tri.h;
        nbr.v    = EVALUATE(gluing, tri.v);

        our_data = tri.tet->cusp_nbhd_position;
        nbr_data = nbr.tet->cusp_nbhd_position;

        /*
         * If the neighbor hasn't been set . . .
         */

        if (nbr_data->in_use[nbr.h][nbr.v] == FALSE)
        {
            /*
             * . . . set it . . .
             */

            f[0] = remaining_face[tri.v][ff];
            f[1] = remaining_face[ff][tri.v];
            f[2] = ff;

            for (i = 0; i < 3; i++)
                nbr_f[i] = EVALUATE(gluing, f[i]);

            for (i = 0; i < 2; i++)
                nbr_data->x[nbr.h][nbr.v][nbr_f[i]] = our_data->x[tri.h][tri.v][f[i]];

            cn_find_third_corner(nbr.tet, nbr.h, nbr.v, nbr_f[0], nbr_f[1], nbr_f[2]);

            nbr_data->in_use[nbr.h][nbr.v] = TRUE;

            /*
             * . . . and put it on the queue.
             */
            queue[++queue_end] = nbr;
        }
    }
}

```

```

    }
}

/*
 * An "unnecessary" error check.
 */
if (queue_begin > max_triangles)
    uFatalError("set_one_component", "cusp_neighborhoods");

/*
 * Free the queue.
 */
my_free(queue);
}

void cn_find_third_corner(
    Tetrahedron    *tet,    /* which tetrahedron */
    Orientation    h,      /* right_handed or left_handed sheet */
    VertexIndex    v,      /* which ideal vertex */
    FaceIndex      f0,     /* known corner */
    FaceIndex      f1,     /* known corner */
    FaceIndex      f2)     /* corner to be computed */
{
    /*
     * We want to position the Tetrahedron so that the following
     * two conditions hold.
     *
     * (1) The corners f0, f1 and f2 are arranged counterclockwise
     *     around the triangle's perimeter.
     *
     *
     *          f2
     *         / \
     *        /   \
     *       /-----\
     *      f0-----f1
     *
     * (2) The cusp cross section is seen with its preferred orientation.
     *     (Cf. the discussion in the second paragraph of section (2) in
     *     the documentation at the top of the file peripheral_curves.c.)
     *     If this is the right handed sheet (h == right_handed),
     *     the Tetrahedron should appear right handed.
     *     (Cf. the definition of Orientation in kernel_typedefs.h.)
     *     If this is the left handed sheet (h == left_handed), the
     *     Tetrahedron should appear left handed (the left_handed sheet has
     *     the opposite orientation of the Tetrahedron, so if this is the
     *     left handed sheet and the Tetrahedron is viewed in a left handed
     *     position, the sheet will be appear right handed -- got that?).
     *
     * Of course these two conditions may not be compatible.
     * If we position the corners as in (1) and then find that (2) doesn't
     * hold (or vice versa), then we must swap the indices f0 and f1.
     *
     * Note: We could force the conditions to hold by making our
     * recursive calls carefully and consistently, but fixing the
     * ordering of f0 and f1 as needed is simpler and more robust.
     */

    Orientation tet_orientation;
    FaceIndex    temp;
    Complex      s,
                t,
                z;

    /*
     * Position the tetrahedron as in Condition (1) above.
     * If the tetrahedron appears in its right_handed Orientation,
     * then remaining_face[f0][f1] == f2, according to the definition of
     * remaining_face[][] in tables.c. If the tetrahedron appears in
     * its left_handed Orientation, then remaining_face[f0][f1] == v.
     */
    tet_orientation = (remaining_face[f0][f1] == f2) ?
        right_handed :
        left_handed;

```

```

/*
 * Does the vertex cross section appear with its preferred orientation,
 * as discussed in Condition (2) above? If not, fix it.
 */
if (h != tet_orientation)
{
    temp = f0;
    f0 = f1;
    f1 = temp;

    tet_orientation = ! tet_orientation;
}

/*
 * Let s be the vector from f0 to f1,
 * t be the vector from f0 to f2,
 * z be the complex edge angle v/u.
 */

s = complex_minus( tet->cusp_nbhd_position->x[h][v][f1],
                  tet->cusp_nbhd_position->x[h][v][f0]);

/*
 * TetShapes are always stored relative to the right_handed Orientation.
 * If we're viewing the tetrahedron relative to the left_handed
 * Orientation, we need to use the conjugate-inverse instead.
 */
z = tet->shape[complete]->cwl[ultimate][edge3_between_vertices[v][f0]].rect;
if (tet_orientation == left_handed)
    z = complex_conjugate(complex_div(One, z));

t = complex_mult(z, s);

tet->cusp_nbhd_position->x[h][v][f2]
    = complex_plus(tet->cusp_nbhd_position->x[h][v][f0], t);
}

void get_cusp_neighborhood_translations(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index,
    Complex *meridian,
    Complex *longitude)
{
    Cusp *cusp;

    cusp = find_cusp(cusp_neighborhoods->its_triangulation, cusp_index);

    *meridian = complex_real_mult(cusp->displacement_exp, cusp->translation[M]);
    *longitude = complex_real_mult(cusp->displacement_exp, cusp->translation[L]);
}

CuspNbhdSegmentList *get_cusp_neighborhood_triangulation(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index)
{
    Cusp *cusp;
    CuspNbhdSegmentList *theSegmentList;
    CuspNbhdSegment *next_segment;
    Tetrahedron *tet,
    *nbr_tet;
    Complex (*x)[4][4];
    Boolean (*in_use)[4];
    VertexIndex v;
    Orientation h;
    FaceIndex f,
    nbr_f;

    /*
     * Make sure the EdgeClasses are numbered.
     */
    number_the_edge_classes(cusp_neighborhoods->its_triangulation);

```

```

/*
 * Find the requested Cusp.
 */
cusp = find_cusp(cusp_neighborhoods->its_triangulation, cusp_index);

/*
 * Allocate the wrapper for the array.
 */
theSegmentList = NEW_STRUCT(CuspNbhdSegmentList);

/*
 * We don't know ahead of time exactly how many CuspNbhdSegments
 * we'll need. Torus cusps report each segment once, but Klein
 * bottle cusps report each segment twice, once for each sheet.
 *
 * To get an upper bound on the number of segments,
 * assume all cusps are Klein bottle cusps.
 *
 *      n tetrahedra
 *      * 4 vertices/tetrahedron
 *      * 2 triangles/vertex      (left_handed and right_handed)
 *      * 3 sides/triangle
 *      / 2 sides/visible side    (no need to draw each edge twice)
 *
 *      = 12n visible sides
 */
theSegmentList->segment = NEW_ARRAY(12*cusp_neighborhoods->its_triangulation->
num_tetrahedra, CuspNbhdSegment);

/*
 * Keep a pointer to the first empty CuspNbhdSegment.
 */
next_segment = theSegmentList->segment;

for (tet = cusp_neighborhoods->its_triangulation->tet_list_begin.next;
     tet != &cusp_neighborhoods->its_triangulation->tet_list_end;
     tet = tet->next)
{
    x      = tet->cusp_nbhd_position->x;
    in_use = tet->cusp_nbhd_position->in_use;

    for (v = 0; v < 4; v++)
    {
        /*
         * If this isn't the cusp the user wants, ignore it.
         */
        if (tet->cusp[v] != cusp)
            continue;

        for (h = 0; h < 2; h++)      /* h = right_handed, left_handed */
        {
            if (in_use[h][v] == FALSE)
                continue;

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

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

                /*
                 * We want to report each segment only once, so we
                 * make the (arbitrary) convention that we report
                 * a segment only from the Tetrahedron whose address
                 * in memory is less. In the case of a Tetrahedron
                 * glued to itself, we report it from the lower
                 * FaceIndex.
                 */
                if (tet > nbr_tet || (tet == nbr_tet && f > nbr_f))
                    continue;
            }
        }
    }
}

```

```

        * Don't report edges which are part of the arbitrary
        * subdivision of the canonical cell decomposition
        * into tetrahedra. We rely on the fact that
        * proto_canonize() has computed the tilts and left
        * them in place. The sum of the tilts will never be
        * positive for a subdivision of the canonical cell
        * decomposition. If it's close to zero, ignore that
        * face.
        */
    if (tet->tilt[f] + nbr_tet->tilt[nbr_f] > -CONCAVITY_EPSILON)
        continue;

    /*
     * This edge has passed all its tests, so record it.
     */
    next_segment->endpoint[0] = complex_real_mult(cusp->displacement_exp,
x[h][v][remaining_face[f][v]]);
    next_segment->endpoint[1] = complex_real_mult(cusp->displacement_exp,
x[h][v][remaining_face[v][f]]);
    next_segment->start_index = tet->edge_class[edge_between_vertices[v]
[remaining_face[f][v]]->index;
    next_segment->middle_index = tet->edge_class[edge_between_faces[v][f]]
->index;
    next_segment->end_index = tet->edge_class[edge_between_vertices[v]
[remaining_face[v][f]]->index;

    /*
     * Move on.
     */
    next_segment++;
}
}
}

/*
 * How many segments did we find?
 *
 * (ANSI C will subtract the pointers correctly, automatically
 * dividing by sizeof(CuspNbhdSegment).)
 */
theSegmentList->num_segments = next_segment - theSegmentList->segment;

/*
 * Did we find more segments than we had allocated space for?
 * This should be impossible, but it doesn't hurt to check.
 */
if (theSegmentList->num_segments > 12*cusp_neighborhoods->its_triangulation->
num_tetrahedra)
    uFatalError("get_cusp_neighborhood_triangulation", "cusp_neighborhoods");

return theSegmentList;
}

void free_cusp_neighborhood_segment_list(
    CuspNbhdSegmentList *segment_list)
{
    if (segment_list != NULL)
    {
        if (segment_list->segment != NULL)
            my_free(segment_list->segment);

        my_free(segment_list);
    }
}

CuspNbhdHoroballList *get_cusp_neighborhood_horoballs(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index,
    Boolean full_list,
    double cutoff_height)
{

```

```

    Cusp                *cusp;
    CuspNbhdHoroballList *theHoroballList;

    /*
     * Find the requested Cusp.
     */
    cusp = find_cusp(cusp_neighborhoods->its_triangulation, cusp_index);

    /*
     * Provide a small margin to allow for roundoff error.
     */
    cutoff_height -= CUTOFF_HEIGHT_EPSILON;

    /*
     * Use the appropriate algorithm for finding
     * the quick or full list of horoballs.
     */
    if (full_list == FALSE)
        theHoroballList = get_quick_horoball_list(cusp_neighborhoods, cusp);
    else
        theHoroballList = get_full_horoball_list(cusp_neighborhoods, cusp, cutoff_height);

    /*
     * Sort the horoballs in order of increasing size.
     */
    qsort( theHoroballList->horoball,
           theHoroballList->num_horoballs,
           sizeof(CuspNbhdHoroball),
           &compare_horoballs);

    /*
     * There's a chance that get_full_horoball_list() may produce duplicate
     * horoballs (when a 2-cell passes through a horoball's north pole) or
     * that get_quick_horoball_list() may produce duplicate horoballs
     * (when face horoballs coincide). Remove any such duplications.
     */
    cull_duplicate_horoballs(cusp, theHoroballList);

    return theHoroballList;
}

static CuspNbhdHoroballList *get_quick_horoball_list(
    CuspNeighborhoods *cusp_neighborhoods,
    Cusp *cusp)
{
    CuspNbhdHoroballList *theHoroballList;
    CuspNbhdHoroball *next_horoball;

    /*
     * Allocate the wrapper for the array.
     */
    theHoroballList = NEW_STRUCT(CuspNbhdHoroballList);

    /*
     * We don't know ahead of time exactly how many CuspNbhdHoroballs
     * we'll need. Torus cusps report each horoball once, but Klein
     * bottle cusps report each horoball twice, once for each sheet.
     * To get an upper bound on the number of horoballs, assume all
     * cusps are Klein bottle cusps. We report two types of horoballs.
     *
     * Edge Horoballs
     *
     * Edge horoballs are horoballs which the given cusp sees along an
     * edge of the canonical triangulation (i.e. along a vertical edge
     * in the usual upper half space picture). The total number of
     * edges in the canonical triangulation is the same as the number
     * of tetrahedra (by an Euler characteristic argument), so the
     * following gives an upper bound on the number of edge horoballs.
     *
     *      n edges
     *      * 2 endpoints/edge
     *      * 2 sheets/endpoint          (left_handed and right_handed)
     */

```

```

*           = 4n edge horoballs
*
*   Face Horoballs
*
*   Face horoballs are horoballs which the given cusp sees across
*   a face of the canonical triangulation. The number of triangles
*   in the cusp triangulation provides an upper bound on the number
*   of face horoballs.
*
*           n tetrahedra
*           * 4 vertices/tetrahedron
*           * 2 triangles/vertex      (left_handed and right_handed)
*
*           = 8n visible sides
*
*   Therefore the total number of horoballs we will report will be
*   at most  $4n + 8n = 12n$ . (The maximum will be realized in the case
*   of a manifold like the Gieseking with one nonorientable cusp.)
*/
theHoroballList->horoball = NEW_ARRAY(12*cusp_neighborhoods->its_triangulation->
num_tetrahedra, CuspNbhdHoroball);

/*
*   Keep a pointer to the first empty CuspNbhdHoroball.
*/
next_horoball = theHoroballList->horoball;

/*
*   Find the edge horoballs.
*/
get_quick_edge_horoballs(    cusp_neighborhoods->its_triangulation,
                             cusp,
                             &next_horoball);

/*
*   Find the face horoballs.
*/
get_quick_face_horoballs(    cusp_neighborhoods->its_triangulation,
                             cusp,
                             &next_horoball);

/*
*   How many horoballs did we find?
*
*   (ANSI C will subtract the pointers correctly, automatically
*   dividing by sizeof(CuspNbhdHoroball).)
*/
theHoroballList->num_horoballs = next_horoball - theHoroballList->horoball;

/*
*   Did we find more horoballs than we had allocated space for?
*   This should be impossible, but it doesn't hurt to check.
*/
if (theHoroballList->num_horoballs > 12*cusp_neighborhoods->its_triangulation->
num_tetrahedra)
    uFatalError("get_cusp_neighborhood_triangulation", "cusp_neighborhoods");

return theHoroballList;
}

```

```

static void get_quick_edge_horoballs(
    Triangulation    *manifold,
    Cusp             *cusp,
    CuspNbhdHoroball **next_horoball)
{
    EdgeClass        *edge;
    double           radius;
    Tetrahedron      *tet;
    Complex           (*x)[4][4];
    Boolean           (*in_use)[4];
    VertexIndex      v[2];
    int              i;
    int              other_index;
}

```

```

Orientation          h;

for (edge = manifold->edge_list_begin.next;
     edge != &manifold->edge_list_end;
     edge = edge->next)
{
    /*
     * Consider a horosphere of Euclidean height h in the upper half
     * space model.  Integrate along a vertical edge connecting the
     * horosphere to the horosphere at infinity to compute the distance
     * between the two as
     *
     *      d = integral of dz/z from z=h to z=1
     *          = log 1 - log h
     *          = - log h
     * or
     *      h = exp(-d)
     *
     * set_cusp_neighborhood_displacement() calls compute_cusp_stoppers(),
     * which in turn calls compute_intercusp_distances(), so we may use
     * the edge->intercusp_distance fields for d.
     */
    radius = 0.5 * exp( - edge->intercusp_distance);

    /*
     * Dereference tet, x and in_use for clarity.
     */
    tet      = edge->incident_tet;
    x        = tet->cusp_nbhd_position->x;
    in_use   = tet->cusp_nbhd_position->in_use;

    /*
     * Consider each of the edge's endpoints.
     */
    v[0] = one_vertex_at_edge[edge->incident_edge_index];
    v[1] = other_vertex_at_edge[edge->incident_edge_index];

    for (i = 0; i < 2; i++)
    {
        /*
         * Are we at the right cusp?
         */
        if (tet->cusp[v[i]] != cusp)
            continue;

        /*
         * What is the index of the other cusp?
         */
        other_index = tet->cusp[v[!i]]->index;

        for (h = 0; h < 2; h++)          /* h = right_handed, left_handed */
        {
            if (in_use[h][v[i]] == FALSE)
                continue;

            (*next_horoball)->center      = complex_real_mult(cusp->displacement_exp, ✎
x[h][v[i]][v[!i]]);
            (*next_horoball)->radius      = radius;
            (*next_horoball)->cusp_index   = other_index;

            (*next_horoball)++;
        }
    }
}

static void get_quick_face_horoballs(
    Triangulation      *manifold,
    Cusp                *cusp,
    CuspNbhdHoroball   **next_horoball)
{
    /*
     * There are several ways we might find the location and size of

```



```

* the face horoballs.
*
* (1) Use the TetShape to locate the center, and then use the
* lemma below to find the size.
*
* This method is fairly efficient computationally, and lets
* us use the existing function compute_fourth_corner() from
* choose_generators.c.
*
* (2) Ignore the TetShape, and rely entirely on the intercusp_distances
* to find both the location and size.
*
* This method is conceptually straightforward. Using the lemma
* below, one obtains three equations involving the location (x,y)
* and the height h of the face horoball. The equations are
* quadratic in x and y, but they are monic, so subtracting
* equations gives linear dependencies between x, y and h.
* One can solve for x and y in terms of h, and obtain a quadratic
* equation to solve for h. It's easy to prove that the lesser
* value of h will be the desired solution. Confession: I haven't
* actually worked out the equation for h. It seems like it would
* be messy.
*
* (3) Work in the Minkowski space model, and use linear algebra
* to compute the horoball as a vector on the light cone.
*
* For background ideas, see
*
* Weeks, Convex hulls and isometries of cusped hyperbolic
* 3-manifolds, Topology Appl. 52 (1993) 127-149
* and
* Sakuma and Weeks, The generalized tilt formula,
* Geometriae Dedicata 55 (1995) 115-123.
*
* The method might prove to be more-or-less equivalent to (2).
* By Lemma 4.2(c) of Weeks, the equation  $\langle u, v \rangle = \text{constant}$  gives
* all the horospheres v a fixed distance from a horosphere u.
* So to find a horosphere a given distance from three given
* horospheres, one ends up intersecting three hyperplanes in
*  $E^3(1)$  to get a line, and then intersecting the line with the
* upper light cone. As in approach (2), the calculations are
* initially linear, but become quadratic at the end. Again, I
* haven't worked through the details.
*
* (4) Find a matrix in  $PSL(2, C)$  which takes an ideal tetrahedron
* in standard position to the desired ideal tetrahedron.
*
* This is the approach used in snappea 1.3. The formulas are
* simpler than you might expect. The main disadvantage is that
* the 1.3 treatment applies only to orientable manifolds. It
* might be possible to fix it up using MoebiusTransformations.
*
* We use method (1), because it seems simplest.
*
* Lemma. Consider two horospheres of Euclidean height h1 and h2 (resp.)
* in the upper half space model of hyperbolic 3-space. If the
* Euclidean distances between their centers (on the sphere at infinity)
* is c, then the hyperbolic distance d between the horospheres is
*
* 
$$d = \log( c^2 / h1 \cdot h2 )$$

*
* Proof. Draw yourself a picture of the horospheres (or horocycles --
* a 2D cross sectional picture will serve just as well). Label the
* distances h1, h2, c and d. Now sketch a Euclidean hemisphere of
* radius c centered at the base of the first horosphere; this is
* a plane in hyperbolic space. Reflect the whole picture in this
* plane (in Euclidean terms, the reflection is an inversion in the
* hemisphere). One of the horospheres gets taken to a horizontal
* Euclidean plane at height  $c^2/h1$ . The other horosphere remains
* (setwise) invariant. It is now obvious that the shortest distance
* from one horosphere to the other is along the vertical arc connecting
* them. The distance is the integral of  $dz/z$  from  $h=h2$  to  $h=c^2/h1$ ,
* which works out to be  $\log( c^2 / h1 \cdot h2 )$ . QED

```

```

* Comment. We don't need it for the present code, but I can't
* resist pointing out that the above lemma has a nice intrinsic
* formulation, which doesn't rely on the upper half space model.
* Let H be the horosphere which appears as a horizontal plane  $z == 1$ 
* in the upper half space model, and draw in the vertical geodesics
* connecting it to each of the two horospheres mentioned in the lemma.
* Let  $a = -\log(h_1)$  and  $b = -\log(h_2)$  be the respective distances from
* H to each of the old horospheres. Interpret c as the distance along
* H from one of those segments to the other. Now redraw the picture
* in, say, the Poincare ball model. It'll be more symmetric now,
* since there's no longer a preferred "horosphere at infinity".
* You'll have an ideal triangle, with a horosphere at each vertex.
* The quantities a, b and d are the length of the shortest geodesics
* between horospheres, while c is the distance along a horosphere
* between two such geodesics. The above lemma becomes
*
* Lemma.  $2 \log c = d - a - b$ .
*
* With better notation, namely a, b and c are the distances between
* cusp cross sections, and A, B and C are the distances along the
* cusps, the lemma becomes
*
* 
$$2 \log A = a - b - c$$

* 
$$2 \log B = b - c - a$$

* 
$$2 \log C = c - a - b$$

*
* Add two of those equations (say the first two) to get
*
* 
$$\log AB = -c$$

*
* As a special case, when  $c == 0$ ,  $AB = 1$ .
*/

Tetrahedron    *tet;
Complex        (*x)[4][4];
Boolean        (*in_use)[4];
VertexIndex    u,
               v,
               w,
               missing_corner;
Permutation    gluing;
Complex        corner[4];
Orientation    h;
double         height_u,
               exp_d,
               c_squared;

for (tet = manifold->tet_list_begin.next;
     tet != &manifold->tet_list_end;
     tet = tet->next)
{
    x      = tet->cusp_nbhd_position->x;
    in_use = tet->cusp_nbhd_position->in_use;

    for (v = 0; v < 4; v++)
    {
        /*
        * Are we at the right cusp?
        */
        if (tet->cusp[v] != cusp)
            continue;

        gluing = tet->gluing[v];

        for (h = 0; h < 2; h++) /* h = right_handed, left_handed */
        {
            if (in_use[h][v] == FALSE)
                continue;

            /*
            * Prepare for a call to compute_fourth_corner().
            */
            for (w = 0; w < 4; w++)
                if (w != v)

```

```

        corner[EVALUATE(gluing, w)] = complex_real_mult(cusp->
displacement_exp, x[h][v][w]);
        missing_corner = EVALUATE(gluing, v);

        /*
         * Call compute_fourth_corner() to compute
         * corner[missing_corner].
         */
        compute_fourth_corner(
            corner,
            missing_corner,
            (parity[gluing] == orientation_preserving) ? h : !h,
            tet->neighbor[v]->shape[complete]->cw1[ultimate]);

        /*
         * The missing_corner gives us the horoball's center.
         */
        (*next_horoball)->center = corner[missing_corner];

        /*
         * Prepare to use the above lemma to compute the radius.
         */

        /*
         * Let u be any vertex of the original Tetrahedron except v.
         */
        u = !v;

        /*
         * According to the explanation in get_quick_edge_horoballs(),
         * the height of the edge horoball at vertex u is
         * exp( - intercusp_distance).
         */
        height_u = exp( - tet->edge_class[edge_between_vertices[u][v]]->
intercusp_distance);

        /*
         * A different intercusp_distance gives the distance d
         * in the lemma.
         */
        exp_d = exp(tet->neighbor[v]->edge_class[edge_between_vertices[EVALUATE
(gluing,u)][missing_corner]]->intercusp_distance);

        /*
         * Compute the squared distance between the edge horoball
         * at vertex u and the face horoball we are interested in.
         */
        c_squared = complex_modulus_squared(complex_minus(
            (*next_horoball)->center,
            complex_real_mult(cusp->displacement_exp, x[h][v][u])));

        /*
         * Apply the lemma.
         *
         *      exp(d) = c^2 / h1*h2
         *  =>
         *      h1 = c^2 / exp(d)*h2
         */
        (*next_horoball)->radius = 0.5 * c_squared / (exp_d * height_u);

        /*
         * Note the cusp index of the new horoball.
         */
        (*next_horoball)->cusp_index = tet->neighbor[v]->cusp[missing_corner]->
index;

        /*
         * Move on.
         */
        (*next_horoball)++;
    }
}
}

```

```

static CuspNbhdHoroballList *get_full_horoball_list(
    CuspNeighborhoods *cusp_neighborhoods,
    Cusp *cusp,
    double cutoff_height)
{
    /*
     * We want to find all horoballs of Euclidean height at least
     * cutoff_height, up to the  $Z + Z$  action of the group of covering
     * transformations of the cusp. (We work with the double cover
     * of Klein bottle cusps, so in effect all cusps are torus cusps.)
     *
     * Let  $M'$  be  $H^3 / (Z + Z)$ , where the  $Z + Z$  is the group of covering
     * transformations of the cusp. Visualize  $M'$  as a chimney in the
     * upper half space model; when its sides are glued together its
     * parallelogram cross section becomes the torus cross section
     * of the cusp.
     *
     * Our plan is to lift ideal tetrahedra from the original manifold  $M$ 
     * to the chimney manifold  $M'$ . We begin with the tetrahedra incident
     * to the chimney's cusp (i.e. its top end), and then gradually tile
     * our way downward. Whenever a new tetrahedron introduces a new
     * ideal vertex, we consider the horoball centered at that vertex.
     * If its Euclidean height is greater than cutoff_height, we add it
     * to a list. Our challenge is to find an algorithm which does as
     * little tiling as possible, yet still finds all horoballs higher
     * than the cutoff_height.
     *
     * The Naive Algorithm
     *
     * The naive algorithm is to consider the neighbors of each tetrahedron
     * already in the tiling. If adding a neighbor would introduce no
     * new vertices, add it. If adding a neighbor would introduce a new
     * vertex, add it iff the horoball at the new vertex is higher than
     * the cutoff_height.
     *
     * Unfortunately the naive algorithm fails. The Whitehead link
     * provides a counterexample. Visualize the Whitehead link as an
     * octahedron with faces identified. The ideal vertices at the
     * "north and south poles" form one cusp ("the red cusp") while the
     * "equatorial ideal vertices" form the other cusp ("the blue cusp").
     * Push the blue cusp cross section forward until it meets itself,
     * but retract the red cusp cross section until it's tiny. The
     * canonical cell decomposition is a subdivision of the octahedron
     * into two square pyramids (a "northern" and a "southern" one).
     * SnapPea will, of course, arbitrarily subdivide each pyramid into
     * two tetrahedra. Now consider what happens when we apply the naive
     * algorithm to this example, with the red cusp at infinity. Each
     * of the initial tetrahedra has a red vertex at infinity, and three
     * blue vertices on the horizontal plane. Its three neighbors to the
     * sides are other tetrahedra of the same type (red at infinity and
     * blue on the horizontal plane). Its underneath neighbor shares the
     * same three blue vertices, and introduce a new red vertex on the
     * horizontal plane. But because the red horoball is tiny, the naive
     * algorithm will say not to add this tetrahedron. So no new tetrahedra
     * will be added, and the algorithm will terminate. The naive
     * algorithm has therefore failed, because it's missed blue horoballs
     * of varying sizes. (Assuming we've chosen the size of the tiny
     * red cusp cross section to be small enough that the largest red
     * horoballs are smaller than the medium sized blue one.)
     *
     * The naive algorithm's failure was the bad news. The good news
     * is that if we take into account the varying sizes of the horoballs,
     * the algorithm can be patched up and made to work. First a few
     * background lemmas.
     *
     * Lemma 1. For each horoball  $H$ , there is (a lift of) an edge
     * of the canonical cell decomposition which connects  $H$  to some
     * larger horoball  $H'$ .
     *
     * Proof. The horoball  $H$  is surrounded by (lifts of) 2-cells
     * of the Ford complex. Consider a 2-cell  $F$  which lies above some
     * point of  $H$  (in the upper half space model).  $F$  is dual to an edge
    */
}

```

```

* of the canonical cell decomposition which connects H to some other
* horoball H'. F lies above H, so by Lemma 2 below, H' is larger
* than H. QED
*
* Lemma 2. Consider two horoballs H and H'. If H' has a larger
* Euclidean height than H when viewed in some fixed way in the upper
* half space model of hyperbolic 3-space, then the plane P lying
* midway between them appears as a Euclidean hemisphere enclosing
* H and excluding H'. In particular, every point of H is directly
* below some point of P, while no point of H' is.
*
* Proof. Draw the horoballs and construct P. QED
*
* Definition. Two horoballs are "edge-connected" if (a lift of) an
* edge of the canonical cell decomposition connects one to the other.
*
* Lemma 3. Let H' be a horoball which is edge-connected to a smaller
* horoball H. Then the Euclidean distance c between their centers
* (on the boundary plane of the upper half space model) is
*
* 
$$c = \sqrt{a * b * \exp(d)}$$

*
* where
*
*     a = Euclidean height of H'
*     b = Euclidean height of H
*     d = hyperbolic distance from H' to H.
*
* Proof. The lemma in get_quick_face_horoballs() says that
*  $d = \log(c^2 / a*b)$ . Solve for  $c = \sqrt{a * b * \exp(d)}$ . QED
*
* Lemma 4. Let H' be a horoball which is edge-connected to a smaller
* horoball H. If the Euclidean height of H is at least cutoff_height,
* then the Euclidean distance c between the centers of H and H' is
* at least
*
* 
$$c \geq \sqrt{a * \text{cutoff\_height} * \exp(\text{min\_d})}$$

*
* where a is the height of H' and min_d is the least distance from
* the horoball H' to any other horoball.
*
* Proof. Follows immediately from Lemma 3.
*
* Comment. The exp(min_d) factor makes H' act like a bigger horoball
* than it really is. If you were to increase the cusp displacement
* by min_d, the height of H' would increase to  $a*\exp(\text{min\_d})$ .
*
* Definition. (A lift of) an edge of the canonical triangulation
* is "potentially useful" if one endpoint lies at the center of
* a horoball H' of height at least cutoff_height, and the distance
* between its two endpoints is at least c (as defined in Lemma 4).
* (As a special case, vertical edges (in the upper half space) are
* always "potentially useful". The informal justification for this
* is that the horosphere at infinity is infinity large and its center
* is infinitely far away.)
*
* Definition. (A lift of) an ideal tetrahedron is "potentially useful"
* iff it contains at least one potentially useful edge.
*
* The Corrected Algorithm
*
* As before, begin with the tetrahedra incident to the chimney's cusp
* and gradually tile downward. For each tetrahedron already in the
* tiling, consider its four neighbors and add those which are
* potentially useful.
*
* Lemma 5. Let H' be a horoball higher than the cutoff_height.
* If the Corrected Algorithm adds one potentially useful tetrahedron
* incident to H', then it adds them all.
*
* Proof. Look at the surface of the horoball H', which intrinsically
* is a Euclidean plane E. An edge of the triangulation intersects
* the plane E in point P. The edge is potentially useful iff P lies
* within a disk D (of intrinsic radius a/c in the Euclidean geometry
* of the horosphere E, but we don't need that fact). A tetrahedron
* incident to H' is potentially useful iff it intersects the disk D.

```

```

* The set of all such tetrahedra forms a connected set (this follows
* from the path connectedness of the disk D). Therefore if the
* algorithm adds one such tetrahedron, it will add them all. QED
*
* Proposition 6. The Corrected Algorithm finds all horoballs higher
* than the cutoff_height.
*
* Proof. Let H be a horoball of maximal height (greater than the
* cutoff_height) which the algorithm missed. By Lemma 1, there
* is a higher horoball H', and an edge connecting H' to H. The
* edge is potentially useful, by Lemma 4 and the definition of a
* potentially useful edge. By the assumed maximal height of H
* (among all horoballs which the Corrected Algorithm should have
* found but didn't), we know that the algorithm did find H', i.e.
* it added some potentially useful tetrahedron incident to the center
* of H'. By Lemma 5, it must have added all potentially useful
* tetrahedra incident to H', and therefore must have found H. QED
*
* Corollary 7. We can refine the Corrected Algorithm as follows.
* For each tetrahedron T already added, we consider only those
* neighbors T' incident to a face of T which contains at least
* one potentially useful edge.
*
* Proof. The proof of Proposition 6 still works. QED
*/

TilingHoroball      *horoball_linked_list;
TilingQueue         tiling_queue;
TilingTet           *tiling_tree_root,
                   *tiling_tet,
                   *tiling_nbr;

Complex             meridian,
                   longitude;

double              parallelogram_to_square[2][2];
FaceIndex           f;
CuspNbhdHoroballList *theHoroballList;

/*
* We don't know a priori how many horoballs we'll find.
* So we temporarily keep them on a NULL-terminated linked list,
* and transfer them to an array when we're done.
*
* To avoid recording multiple copies of each horoball, we make the
* convention that each horoball is recorded only by the TilingTet
* which contains its north pole. If the north pole lies on the
* boundary of two TilingTets, they both record it.
* get_cusp_neighborhood_horoballs() will remove the duplications.
* If three or more TilingTets meet at the north pole, then a vertical
* edge connects the horoball to infinity in the upper half space model;
* read_initial_tetrahedra() records such horoballs without duplication.
* (Other strategies are possible, like preferring the Tetrahedron
* with the lower address in memory, but the present approach is
* least vulnerable to roundoff error.)
*/
horoball_linked_list = NULL;

/*
* We'll need to store the potentially useful tetrahedra in two ways.
*
* Queue
*   The Tetrahedra which have been added, but whose neighbors have
*   not been examined, go on a queue, so we know which one
*   to process next. When we remove a tetrahedron from the queue
*   we examine its neighbors. We use a queue rather than a stack
*   so that we tile generally downwards (rather than snaking around)
*   in hopes of obtaining the best numerical precision.
*
* Tree
*   All tetrahedra which have been added are kept on a tree, so that
*   we can tell whether new tetrahedra are duplications of old ones
*   or not. (Note: Checking whether a tetrahedron is "the same as"
*   an old one means checking whether they are equivalent under
*   the  $Z + Z$  action of the covering transformations.
*

```

```

    * The TilingTet structure supports both the queue and the tree,
    * simultaneously and independently.
    */

/*
    * Initialize the data structures.
    */
tiling_queue.begin = NULL;
tiling_queue.end = NULL;
tiling_tree_root = NULL;

/*
    * For each cusp, compute the quantity exp(min_d) needed in Lemma 4.
    */
compute_exp_min_d(cusp_neighborhoods->its_triangulation);

/*
    * Compute the current meridional and longitudinal translations.
    */
meridian = complex_real_mult(cusp->displacement_exp, cusp->translation[M]);
longitude = complex_real_mult(cusp->displacement_exp, cusp->translation[L]);

/*
    * prepare_sort_key() will need a linear transformation which
    * maps a fundamental parallelogram for the cusp (or the double
    * cover, in the case of a Klein bottle cusp) to the unit square.
    */
compute_parallelogram_to_square(meridian, longitude, parallelogram_to_square);

/*
    * Read in the tetrahedra incident to the vertex at infinity,
    * and record the incident horoballs.
    *
    * Note: We check the horoballs when we put TilingTets onto the
    * tiling_queue (rather than when we pull it off) so we can handle
    * the special case of the initial tetrahedra more efficiently.
    */
read_initial_tetrahedra(    cusp_neighborhoods->its_triangulation,
                           cusp,
                           &tiling_queue,
                           &tiling_tree_root,
                           &horoball_linked_list,
                           cutoff_height);

/*
    * Carry out the Corrected Algorithm, refined as in Lemma 7.
    */
while (tiling_queue.begin != NULL)
{
    tiling_tet = get_tiling_tet_from_queue(&tiling_queue);

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

        if (tiling_tet->neighbor_found[f] == FALSE
            && face_contains_useful_edge(tiling_tet, f, cutoff_height) == TRUE)
        {
            tiling_nbr = make_neighbor_tiling_tet(tiling_tet, f);

            prepare_sort_key(tiling_nbr, parallelogram_to_square);

            if (tiling_tet_on_tree(tiling_nbr, tiling_tree_root, meridian, longitude) =
= FALSE)
            {
                add_horoball_if_necessary(tiling_nbr, &horoball_linked_list,
cutoff_height);
                add_tiling_tet_to_tree(tiling_nbr, &tiling_tree_root);
                add_tiling_tet_to_queue(tiling_nbr, &tiling_queue);
            }
            else
                my_free(tiling_nbr);
        }
    }
}

/*

```

```

    *   Free the TilingTets.
    */
    free_tiling_tet_tree(tiling_tree_root);

    /*
    *   Transfer the horoballs from the linked list
    *   to a CuspNbhdHoroballList, and free the linked list.
    */
    theHoroballList = transfer_horoballs(&horoball_linked_list);

    return theHoroballList;
}

static void compute_exp_min_d(
    Triangulation *manifold)
{
    /*
    *   Compute the quantity exp(min_d) needed
    *   in Lemma 4 of get_full_horoball_list().
    */

    Cusp *cusp;
    EdgeClass *edge;
    double exp_d;
    VertexIndex v[2];
    int i;

    /*
    *   Initialize all exp_min_d's to infinity.
    */

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

        cusp->exp_min_d = DBL_MAX;

    /*
    *   The closest horoball to a given cusp will lie along an edge
    *   of the canonical cell decomposition, so look at all edges
    *   to find the true exp_min_d's.
    */

    for (edge = manifold->edge_list_begin.next;
        edge != &manifold->edge_list_end;
        edge = edge->next)
    {
        /*
        *   set_cusp_neighborhood_displacement() calls compute_cusp_stoppers(),
        *   which in turn calls compute_intercusp_distances(), so we may use
        *   the edge->intercusp_distance fields for exp_d.
        */
        exp_d = exp(edge->intercusp_distance);

        v[0] = one_vertex_at_edge[edge->incident_edge_index];
        v[1] = other_vertex_at_edge[edge->incident_edge_index];

        for (i = 0; i < 2; i++)
        {
            cusp = edge->incident_tet->cusp[v[i]];

            if (cusp->exp_min_d > exp_d)
                cusp->exp_min_d = exp_d;
        }
    }
}

static void compute_parallelogram_to_square(
    Complex meridian,
    Complex longitude,
    double parallelogram_to_square[2][2])
{

```



```

/*
 * prepare_sort_key() needs a linear transformation which takes
 * a meridian to (1,0) and a longitude to (0,1), so TilingTets which
 * are equivalent under the  $\mathbb{Z} + \mathbb{Z}$  action of the group of covering
 * translations of the cusp be assigned corner coordinates which
 * differ by integers. The required linear transformation is
 * the inverse of
 *
 *          ( meridian.real  longitude.real )
 *          ( meridian.imag  longitude.imag )
 */

double det;

det = meridian.real * longitude.imag - meridian.imag * longitude.real;

parallelogram_to_square[0][0] = longitude.imag / det;
parallelogram_to_square[0][1] = - longitude.real / det;
parallelogram_to_square[1][0] = - meridian.imag / det;
parallelogram_to_square[1][1] = meridian.real / det;
}

static void read_initial_tetrahedra(
    Triangulation *manifold,
    Cusp *cusp,
    TilingQueue *tiling_queue,
    TilingTet **tiling_tree_root,
    TilingHoroball **horoball_linked_list,
    double cutoff_height)
{
    Tetrahedron *tet;
    Complex (*x)[4][4];
    Boolean (*in_use)[4];
    VertexIndex v, w;
    Orientation h;
    TilingTet *tiling_tet;
    EdgeIndex edge_index;
    EdgeClass *edge;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        x = tet->cusp_nbhd_position->x;
        in_use = tet->cusp_nbhd_position->in_use;

        for (v = 0; v < 4; v++)
        {
            if (tet->cusp[v] != cusp)
                continue;

            for (h = 0; h < 2; h++) /* h = right_handed, left_handed */
            {
                if (in_use[h][v] == FALSE)
                    continue;

                tiling_tet = NEW_STRUCT(TilingTet);

                tiling_tet->underlying_tet = tet;
                tiling_tet->orientation = h;

                for (w = 0; w < 4; w++)
                    if (w != v)
                    {
                        /*
                         * Please see get_quick_edge_horoballs() for
                         * an explanation of the horoball height.
                         */
                        edge_index = edge_between_vertices[v][w];
                        edge = tet->edge_class[edge_index];

                        tiling_tet->corner[w] = complex_real_mult(cusp->

```

```

displacement_exp, x[h][v][w]);
    tiling_tet->horoball_height[w] = exp( - edge->intercusp_distance);
    tiling_tet->neighbor_found[w] = TRUE;

    /*
     * To avoid duplications, record the TilingHoroball
     * iff this is the preferred tet and edge_index
     * to see it from.
     */

    if (edge->incident_tet == tet
        && edge->incident_edge_index == edge_index
        && tiling_tet->horoball_height[w] >= cutoff_height)

        add_tiling_horoball_to_list(tiling_tet, w,
horoball_linked_list);
    }
    else
    {
        tiling_tet->corner[w] = Infinity;
        tiling_tet->horoball_height[w] = DBL_MAX;
        tiling_tet->neighbor_found[w] = FALSE;
    }

    /*
     * Give each tiling_tet a random value of the sort key,
     * to keep the tree broad.
     */
    tiling_tet->key = 0.5 * ((double) rand() / (double) RAND_MAX);

    add_tiling_tet_to_queue(tiling_tet, tiling_queue);
    add_tiling_tet_to_tree(tiling_tet, tiling_tree_root);
}
}
}

static TilingTet *get_tiling_tet_from_queue(
    TilingQueue *tiling_queue)
{
    TilingTet *tiling_tet;

    tiling_tet = tiling_queue->begin;

    if (tiling_queue->begin != NULL)
        tiling_queue->begin = tiling_queue->begin->next;

    if (tiling_queue->begin == NULL)
        tiling_queue->end = NULL;

    return tiling_tet;
}

static void add_tiling_tet_to_queue(
    TilingTet *tiling_tet,
    TilingQueue *tiling_queue)
{
    tiling_tet->next = NULL;

    if (tiling_queue->end != NULL)
    {
        tiling_queue->end->next = tiling_tet;
        tiling_queue->end = tiling_tet;
    }
    else
    {
        tiling_queue->begin = tiling_tet;
        tiling_queue->end = tiling_tet;
    }
}

```

```

static void add_tiling_horoball_to_list(
    TilingTet      *tiling_tet,
    VertexIndex    v,
    TilingHoroball **horoball_linked_list)
{
    TilingHoroball *tiling_horoball;

    tiling_horoball = NEW_STRUCT(TilingHoroball);

    tiling_horoball->data.center      = tiling_tet->corner[v];
    tiling_horoball->data.radius      = 0.5 * tiling_tet->horoball_height[v];
    tiling_horoball->data.cusp_index  = tiling_tet->underlying_tet->cusp[v]->index;

    tiling_horoball->next = *horoball_linked_list;
    *horoball_linked_list = tiling_horoball;
}

static Boolean face_contains_useful_edge(
    TilingTet      *tiling_tet,
    FaceIndex      f,
    double         cutoff_height)
{
    /*
     * Note: We may assume that the face f has no vertices at the point
     * at infinity in upper half space. The reason is that the initial
     * tetrahedra have neighbor_found[] == TRUE for their side faces, and
     * get_full_horoball_list() calls us only if neighbor_found[f] is FALSE.
     */

    /*
     * How many vertices incident to face f have horoballs
     * higher than cutoff_height?
     */

    int          num_big_horoballs;
    VertexIndex  v,
                big_vertex;
    double       min_separation_sq;

    num_big_horoballs = 0;

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

        if (tiling_tet->horoball_height[v] > cutoff_height)
        {
            num_big_horoballs++;
            big_vertex = v;
        }
    }

    /*
     * If there are no big horoballs,
     * the face cannot contain a useful edge.
     */
    if (num_big_horoballs == 0)
        return FALSE;

    /*
     * If there are two or more big horoballs,
     * the face must contain a useful edge.
     */
    if (num_big_horoballs >= 2)
        return TRUE;

    /*
     * At this point we know that the unique large horoball lies
     * at the vertex big_vertex. There will be a useful edge iff
     * the distance from big_vertex to some other vertex of face f
     * is at least sqrt( height_of_big_vertex * cutoff_height * exp(min_d) ).
     * For a detailed explanation, please see Lemma 4 and the definition

```

```

    * of "useful edge" in get_full_horoball_list().
    */

    min_separation_sq = tiling_tet->horoball_height[big_vertex]
        * cutoff_height
        * tiling_tet->underlying_tet->cusp[big_vertex]->exp_min_d;

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

        if (complex_modulus_squared(
            complex_minus( tiling_tet->corner[big_vertex],
                          tiling_tet->corner[v] )
            ) > min_separation_sq)

            return TRUE;
    }

    return FALSE;
}

static TilingTet *make_neighbor_tiling_tet(
    TilingTet *tiling_tet,
    FaceIndex f)
{
    Tetrahedron *tet,
        *nbr;
    Permutation gluing;
    TilingTet *tiling_nbr;
    VertexIndex v,
        w,
        ff,
        some_vertex;
    double exp_d,
        c_squared;

    /*
     * Find the underlying tetrahedra and the gluing between them.
     */
    tet = tiling_tet->underlying_tet;
    nbr = tet->neighbor[f];
    gluing = tet->gluing[f];

    /*
     * Set up the new TilingTet.
     */

    tiling_nbr = NEW_STRUCT(TilingTet);

    tiling_nbr->underlying_tet = nbr;
    tiling_nbr->orientation = (parity[gluing] == orientation_preserving) ?
        tiling_tet->orientation :
        ! tiling_tet->orientation;

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

        w = EVALUATE(gluing, v);

        tiling_nbr->corner[w] = tiling_tet->corner[v];
        tiling_nbr->horoball_height[w] = tiling_tet->horoball_height[v];
        tiling_nbr->neighbor_found[w] = FALSE;
    }

    /*
     * Deal with the remaining corner.
     */

    ff = EVALUATE(gluing, f);

```

```

/*
 * Call compute_fourth_corner() to locate the remaining ideal vertex.
 */
compute_fourth_corner(  tiling_nbr->corner,
                        ff,
                        tiling_nbr->orientation,
                        nbr->shape[complete]->cwl[ultimate]);

/*
 * Use the lemma from get_quick_face_horoballs() to compute
 * the height of the remaining horoball.
 */
some_vertex = ! ff;
exp_d = exp(nbr->edge_class[edge_between_vertices[ff][some_vertex]]->
intercusp_distance);
c_squared = complex_modulus_squared(complex_minus(
                        tiling_nbr->corner[ff],
                        tiling_nbr->corner[some_vertex]));
tiling_nbr->horoball_height[ff] =
    c_squared / (exp_d * tiling_nbr->horoball_height[some_vertex]);

/*
 * Don't backtrack to the TilingTet we just came from.
 */
tiling_nbr->neighbor_found[ff] = TRUE;

/*
 * get_full_horoball_list() will decide whether to add tiling_nbr
 * to the linked list and tree, and whether to add the new horoball
 * to the horoball list.
 */
tiling_nbr->next          = NULL;
tiling_nbr->left          = NULL;
tiling_nbr->right         = NULL;
tiling_nbr->key           = 0.0;

return tiling_nbr;
}

static void prepare_sort_key(
    TilingTet    *tiling_tet,
    double       parallelogram_to_square[2][2])
{
    VertexIndex v;
    Complex      transformed_corner[4];

    static const double coefficient[4][2] = {{37.0, 25.0}, {43.0, 13.0}, {2.0, 29.0}, {11.0,
    , 7.0}};

    /*
     * Special case: To avoid questions of numerical accuracy, assign
     * the "illegal" key value of -1 to TilingTets incident to infinity
     * in upper half space. read_initial_tetrahedra() puts all such
     * TilingTets on the tree, so none need be added again.
     */
    for (v = 0; v < 4; v++)
        if (complex_modulus(tiling_tet->corner[v]) > KEY_INFINITY)
        {
            tiling_tet->key = -1.0;
            return;
        }

    /*
     * Recall that we are tiling  $H^3 / (Z + Z)$ , where the  $Z + Z$  is
     * the group of covering transformations of the cusp. In other words,
     * two TilingTets are equivalent iff corresponding corners differ
     * by some combination of meridional and/or longitudinal translations.
     * The linear transformation parallelogram_to_square maps a meridian
     * to (1,0) and a longitude to (0,1). We apply it to the TilingTets'
     * corners, so corresponding corners will differ by integers.
     */
    for (v = 0; v < 4; v++)

```

```

{
    transformed_corner[v].real = parallelogram_to_square[0][0] * tiling_tet->corner[v].real
    + parallelogram_to_square[0][1] * tiling_tet->corner[v].imag;
    transformed_corner[v].imag = parallelogram_to_square[1][0] * tiling_tet->corner[v].real
    + parallelogram_to_square[1][1] * tiling_tet->corner[v].imag;
}

/*
 * To implement a binary tree, we need a search key which is well
 * defined under the action of the meridional and longitudinal
 * translations. In terms of the transformed_corners, it should be
 * well defined under integer translations. Any integer linear
 * combination of the real and imaginary parts of the transformed
 * corners will do. We choose a random looking one, to reduce the
 * chances that distinct points will be assigned the same value of
 * the search key. (Of course the algorithm works correctly in any
 * case -- it's just faster if all the search key values are distinct.)
 * The linear combination provides a continuous map from the transformed
 * corners modulo integers to the reals modulo integers, i.e. to the
 * circle. We then map the circle to the interval [0, 1/2] in a
 * continuous way. (It's a two-to-one map, but that's unavoidable.)
 */

/*
 * Form a random looking integer combination of the corner coordinates.
 */
tiling_tet->key = 0.0;
for (v = 0; v < 4; v++)
{
    tiling_tet->key += coefficient[v][0] * transformed_corner[v].real;
    tiling_tet->key += coefficient[v][1] * transformed_corner[v].imag;
}

/*
 * Take the fractional part.
 */
tiling_tet->key -= floor(tiling_tet->key);

/*
 * Fold the unit interval [0,1] onto the half interval [0, 1/2]
 * in ensure continuity.
 */
if (tiling_tet->key > 0.5)
    tiling_tet->key = 1.0 - tiling_tet->key;
}

static Boolean tiling_tet_on_tree(
    TilingTet    *tiling_tet,
    TilingTet    *tiling_tree_root,
    Complex      meridian,
    Complex      longitude)
{
    TilingTet    *subtree_stack,
    *subtree;
    double       delta;
    Boolean      left_flag,
    right_flag;
    FaceIndex    f;

    /*
     * As a special case, TilingTets incident to infinity in upper half
     * space are already all on the tree. prepare_sort_key() marks
     * duplicates of such TilingTets with a key value of -1. (Computing
     * and comparing the usual key value is awkward when some of the
     * numbers are infinite.)
     */
    if (tiling_tet->key == -1.0)
        return TRUE;

    /*
     * Reliability is our first priority. Speed is second. So if
     * tiling_tet->key and subtree->key are close, we want to search both
     * the left and right subtrees. Otherwise we search only one or the

```

```

    * other. We implement the recursion using our own stack, rather than
    * the system stack, to avoid the possibility of a stack/heap collision
    * during deep recursions.
    */

/*
 * Initialize the stack to contain the whole tree.
 */
subtree_stack = tiling_tree_root;
if (tiling_tree_root != NULL)
    tiling_tree_root->next_subtree = NULL;

/*
 * Process the subtrees on the stack,
 * adding additional subtrees as needed.
 */
while (subtree_stack != NULL)
{
    /*
     * Pull a subtree off the stack.
     */
    subtree = subtree_stack;
    subtree_stack = subtree_stack->next_subtree;
    subtree->next_subtree = NULL;

    /*
     * Compare the key values of the tiling_tet and the subtree's root.
     */
    delta = tiling_tet->key - subtree->key;

    /*
     * Which side(s) should we search?
     */
    left_flag = (delta < +KEY_EPSILON);
    right_flag = (delta > -KEY_EPSILON);

    /*
     * Put the subtrees we need to search onto the stack.
     */
    if (left_flag && subtree->left)
    {
        subtree->left->next_subtree = subtree_stack;
        subtree_stack = subtree->left;
    }
    if (right_flag && subtree->right)
    {
        subtree->right->next_subtree = subtree_stack;
        subtree_stack = subtree->right;
    }

    /*
     * Check this TilingTet if the key values match.
     */
    if (left_flag && right_flag)
    {
        /*
         * Are the TilingTets translations of one another?
         */
        if (same_corners(tiling_tet, subtree, meridian, longitude))
        {
            /*
             * subtree is a TilingTet which may or may not have been
             * processed yet. If not, then when we do process it, we
             * know there's no need to recreate tiling_tet's "parent".
             */
            for (f = 0; f < 4; f++)
                subtree->neighbor_found[f] |= tiling_tet->neighbor_found[f];

            return TRUE;
        }
    }
}

return FALSE;
}

```

```

static Boolean same_corners(
    TilingTet    *tiling_tet1,
    TilingTet    *tiling_tet2,
    Complex      meridian,
    Complex      longitude)
{
    /*
     * Are tiling_tet1 and tiling_tet2 translations of the same tetrahedron
     * in  $H^3/(Z + Z)$  ?
     *
     * Note: This function does not take into account the size of the
     * TilingTets. Two TilingsTets which were very tiny and very close
     * could cause a false positive, and such TilingTets could be mistakenly
     * omitted from the tiling. But that's not likely to happen for any
     * computationally feasible value of cutoff_epsilon.
     */

    Complex      offset,
                fractional_part,
                diff;
    double       num_meridians,
                num_longitudes,
                error;
    VertexIndex v;

    /*
     * Is the offset between a pair of corresponding vertices
     * an integer combination of meridians and longitudes?
     */

    offset = complex_minus( tiling_tet2->corner[0],
                           tiling_tet1->corner[0]);

    fractional_part = offset;

    num_meridians = floor(fractional_part.imag / meridian.imag + 0.5);
    fractional_part = complex_minus(
        fractional_part,
        complex_real_mult(num_meridians, meridian));

    num_longitudes = floor(fractional_part.real / longitude.real + 0.5);
    fractional_part = complex_minus(
        fractional_part,
        complex_real_mult(num_longitudes, longitude));

    if (complex_modulus(fractional_part) > CORNER_EPSILON)
        return FALSE;

    /*
     * Do all pairs of corresponding vertices differ by the same offset?
     */

    for (v = 1; v < 4; v++)
    {
        diff = complex_minus( tiling_tet2->corner[v],
                              tiling_tet1->corner[v]);
        error = complex_modulus(complex_minus(offset, diff));
        if (error > CORNER_EPSILON)
            return FALSE;
    }

    return TRUE;
}

static void add_tiling_tet_to_tree(
    TilingTet    *tiling_tet,
    TilingTet    **tiling_tree_root)
{
    /*
     * tiling_tet_on_tree() has already checked that tiling_tet is not
     * a translation of any TilingTet already on the tree. So here we
     * just add it in the appropriate spot, based on the key value.
     */
}

```



```

    */
    TilingTet      **location;

    location = tiling_tree_root;

    while (*location != NULL)
    {
        if (tiling_tet->key <= (*location)->key)
            location = &(*location)->left;
        else
            location = &(*location)->right;
    }

    *location = tiling_tet;

    tiling_tet->left  = NULL;
    tiling_tet->right = NULL;
}

static void add_horoball_if_necessary(
    TilingTet      *tiling_tet,
    TilingHoroball **horoball_linked_list,
    double          cutoff_height)
{
    VertexIndex v;

    for (v = 0; v < 4; v++)
    {
        /*
         * Ignore horoballs which are too small.
         */
        if (tiling_tet->horoball_height[v] < cutoff_height)
            continue;

        /*
         * Recall the convention made in get_full_horoball_list() that
         * each horoball is recorded only by the TilingTet
         * which contains its north pole. If the north pole lies on the
         * boundary of two TilingTets, they both record it.
         */
        if (contains_north_pole(tiling_tet, v) == TRUE)
            add_tiling_horoball_to_list(tiling_tet, v, horoball_linked_list);
    }
}

static Boolean contains_north_pole(
    TilingTet      *tiling_tet,
    VertexIndex v)
{
    /*
     * Check whether vertex v lies within the triangle defined
     * by the remaining three vertices.
     */

    int i;
    VertexIndex w[3];
    Complex u[3];
    double s[3], det;

    /*
     * Label the remaining three vertices w[0], w[1] and w[2]
     * as you go counterclockwise around the triangle they define
     * on the boundary of upper half space.
     */
    *
    *
    *
    *
    *
    *

```

w[2]

w[0]-----w[1]

```

    *
    *
    *

```

```

    * If v lies inside that triangle we'll return TRUE;
    * otherwise we'll return FALSE.
    */

w[0] = !v;

if (tiling_tet->orientation == right_handed)
{
    w[1] = remaining_face[v][w[0]];
    w[2] = remaining_face[w[0]][v];
}
else
{
    w[1] = remaining_face[w[0]][v];
    w[2] = remaining_face[v][w[0]];
}

/*
 * The vector u[i] runs from v to w[i].
 *
 *
 *           w[2]
 *          / | \
 *         /  v  \
 *        / /   \ \
 *       / /     \ \
 *      w[0]-----w[1]
 *
 */
for (i = 0; i < 3; i++)
    u[i] = complex_minus(tiling_tet->corner[w[i]], tiling_tet->corner[v]);

/*
 * s[i] is the squared length of the triangle's i-th side.
 */
for (i = 0; i < 3; i++)
    s[i] = complex_modulus_squared(complex_minus(tiling_tet->corner[w[(i+1)%3]],
    tiling_tet->corner[w[i]]));

/*
 * If v lies in the triangle's interior, we of course return TRUE.
 * But if v lies (approximately) on one of the triangle's sides, we
 * also want to return TRUE, so that in ambiguous cases horoballs are
 * recorded twice, not zero times.
 *
 * We need a scale invariant measure of the signed distance from v
 * to each side of the triangle, so that we can apply our error epsilon
 * in a meaningful way. (We don't want to return TRUE for *all* tiny
 * triangles, simply because they are tiny!) The determinant
 *
 *
 *           | u[i].real   u[i+1].real |
 *      det = |            |
 *           | u[i].imag   u[i+1].imag |
 *
 *
 * gives twice the area of the triangle (v, w[i], w[i+1]).
 * Therefore det/dist(w[i], w[i+1]) gives the triangle's altitude,
 * and det/dist(w[i], w[i+1])^2 = det/s[i] gives the ratio of
 * the altitude to the length of the side. If that ratio is at least
 * -NORTH_POLE_EPSILON for all sides, we return TRUE.
 */
for (i = 0; i < 3; i++)
{
    det = u[i].real * u[(i+1)%3].imag - u[i].imag * u[(i+1)%3].real;
    if (det / s[i] < -NORTH_POLE_EPSILON)
        return FALSE;
}

return TRUE;
}

static void free_tiling_tet_tree(
    TilingTet *tiling_tree_root)
{
    TilingTet *subtree_stack,
    *subtree;

```

```

/*
 * Implement the recursive freeing algorithm using our own stack
 * rather than the system stack, to avoid the possibility of a
 * stack/heap collision.
 */

/*
 * Initialize the stack to contain the whole tree.
 */
subtree_stack = tiling_tree_root;
if (tiling_tree_root != NULL)
    tiling_tree_root->next_subtree = NULL;

/*
 * Process the subtrees on the stack one at a time.
 */
while (subtree_stack != NULL)
{
    /*
     * Pull a subtree off the stack.
     */
    subtree = subtree_stack;
    subtree_stack = subtree_stack->next_subtree;
    subtree->next_subtree = NULL;

    /*
     * If the subtree's root has nonempty left and/or right subtrees,
     * add them to the stack.
     */
    if (subtree->left != NULL)
    {
        subtree->left->next_subtree = subtree_stack;
        subtree_stack = subtree->left;
    }
    if (subtree->right != NULL)
    {
        subtree->right->next_subtree = subtree_stack;
        subtree_stack = subtree->right;
    }

    /*
     * Free the subtree's root node.
     */
    my_free(subtree);
}
}

static CuspNbhdHoroballList *transfer_horoballs(
    TilingHoroball **horoball_linked_list)
{
    CuspNbhdHoroballList *theHoroballList;
    TilingHoroball *the_tiling_horoball,
                  *the_dead_horoball;
    int i;

    /*
     * Allocate the wrapper.
     */
    theHoroballList = NEW_STRUCT(CuspNbhdHoroballList);

    /*
     * Count the horoballs.
     */
    for (the_tiling_horoball = *horoball_linked_list,
         theHoroballList->num_horoballs = 0;
         the_tiling_horoball != NULL;
         the_tiling_horoball = the_tiling_horoball->next,
         theHoroballList->num_horoballs++)
        ;

    /*
     * If we found some horoballs, allocate an array
     * for the CuspNbhdHoroballs.
     */

```

```

    */
    if (theHoroballList->num_horoballs > 0)
        theHoroballList->horoball = NEW_ARRAY(theHoroballList->num_horoballs,
        CuspNbhdHoroball);
    else
        theHoroballList->horoball = NULL;

    /*
     * Copy the data from the linked list to the array.
     */
    for (    the_tiling_horoball = *horoball_linked_list, i = 0;
           the_tiling_horoball != NULL;
           the_tiling_horoball = the_tiling_horoball->next, i++)

        theHoroballList->horoball[i] = the_tiling_horoball->data;

    /*
     * Free the linked list.
     */
    while (*horoball_linked_list != NULL)
    {
        the_dead_horoball = *horoball_linked_list;
        *horoball_linked_list = the_dead_horoball->next;
        my_free(the_dead_horoball);
    }

    return theHoroballList;
}

void free_cusp_neighborhood_horoball_list(
    CuspNbhdHoroballList *horoball_list)
{
    if (horoball_list != NULL)
    {
        if (horoball_list->horoball != NULL)
            my_free(horoball_list->horoball);

        my_free(horoball_list);
    }
}

static int CDECL compare_horoballs(
    const void *horoball0,
    const void *horoball1)
{
    if (((CuspNbhdHoroball *)horoball0)->radius < ((CuspNbhdHoroball *)horoball1)->radius)
        return -1;
    else if (((CuspNbhdHoroball *)horoball0)->radius > ((CuspNbhdHoroball *)horoball1)->
    radius)
        return +1;
    else
        return 0;
}

static void cull_duplicate_horoballs(
    Cusp *cusp,
    CuspNbhdHoroballList *aHoroballList)
{
    int    original_num_horoballs,
           i,
           j,
           k;
    Complex meridian,
           longitude,
           delta;
    double cutoff_radius,
           mult;
    Boolean distinct;

    /*

```

```

    * Note the meridional and longitudinal translations.
    */
    meridian = complex_real_mult(cusp->displacement_exp, cusp->translation[M]);
    longitude = complex_real_mult(cusp->displacement_exp, cusp->translation[L]);

/*
 * Examine each horoball on the list.
 * If it's distinct from all previously examined horoballs, keep it.
 * Otherwise ignore it.
 *
 * We could implement this algorithm by copying the horoballs
 * we want to keep from the array aHoroballList->horoball onto
 * a new array. But it's simpler just to copy the array onto itself.
 * (This sounds distressing at first, but if you think it through
 * you'll realize that it's perfectly safe.)
 *
 * The index i keeps track of the horoball we're examining.
 * The index j keeps track of where we're writing it to.
 */

original_num_horoballs = aHoroballList->num_horoballs;

for (i = 0, j = 0; j < original_num_horoballs; j++)
{
    /*
     * If the j-th horoball is distinct from all previous ones, copy
     * it into the i-th position of the array. In practice, of course,
     * we compare it only to previous horoballs of the same radius.
     * We may assume that get_cusp_neighborhood_horoballs() has
     * already sorted the horoballs in order of increasing size.
     */

    /*
     * Assume the j-th horoball is distinct from horoballs
     * 0 through i - 1, unless we discover otherwise.
     */
    distinct = TRUE;

    /*
     * What is the smallest radius we should consider?
     */
    cutoff_radius = aHoroballList->horoball[j].radius - DUPLICATE_RADIUS_EPSILON;

    /*
     * Start with horoball i - 1, and work downwards until either
     * we reach horoball 0, or the radii drop below the cutoff_radius.
     */
    for (k = i; --k >= 0; )
    {
        /*
         * If horoball k is too small, there is no need to examine
         * the remaining ones, which are even smaller.
         */
        if (aHoroballList->horoball[k].radius < cutoff_radius)
            break;

        /*
         * Let delta be the difference between the center of j and
         * the center of k, modulo the  $\mathbb{Z} + \mathbb{Z}$  action of the group
         * of covering transformations of the cusp.
         */
        delta = complex_minus(aHoroballList->horoball[j].center,
                               aHoroballList->horoball[k].center);
        mult = floor(delta.imag / meridian.imag + 0.5);
        delta = complex_minus(delta, complex_real_mult(mult, meridian));
        mult = floor(delta.real / longitude.real + 0.5);
        delta = complex_minus(delta, complex_real_mult(mult, longitude));

        /*
         * If the distance between the centers of horoballs j and k is
         * less than the radius, then the horoballs must be equivalent.
         */
        if (complex_modulus(delta) < cutoff_radius)
        {

```

```

        distinct = FALSE;
        break;
    }
}

if (distinct == TRUE)
{
    aHoroballList->horoball[i] = aHoroballList->horoball[j];
    i++;
}
else
    aHoroballList->num_horoballs--;
}
}

CuspNbhdSegmentList *get_cusp_neighborhood_Ford_domain(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index)
{
    Cusp *cusp;
    CuspNbhdSegmentList *theSegmentList;
    CuspNbhdSegment *next_segment;
    Tetrahedron *tet,
    *nbr_tet;
    Complex (*x)[4][4];
    Boolean (*in_use)[4];
    VertexIndex v,
    nbr_v,
    u,
    nbr_u,
    w[3];
    Orientation h,
    nbr_h;
    FaceIndex f,
    nbr_f;
    Permutation gluing;
    int i;
    Complex corner[3],
    delta,
    inward_normal,
    offset,
    p;
    double length,
    tilt,
    a[2],
    b[2],
    c[2],
    det;

    /*
     * Find the requested Cusp.
     */
    cusp = find_cusp(cusp_neighborhoods->its_triangulation, cusp_index);

    /*
     * Allocate the wrapper for the array.
     */
    theSegmentList = NEW_STRUCT(CuspNbhdSegmentList);

    /*
     * We don't know ahead of time exactly how many CuspNbhdSegments
     * we'll need. Torus cusps report each segment once, but Klein
     * bottle cusps report each segment twice, once for each sheet.
     *
     * To get an upper bound on the number of segments,
     * assume all cusps are Klein bottle cusps.
     *
     *      n tetrahedra
     *      * 4 vertices/tetrahedron
     *      * 2 triangles/vertex      (left_handed and right_handed)
     *      * 3 sides/triangle
     *      / 2 Ford edges/side      (no need to draw each edge twice)
     */

```

```

/*      = 12n Ford edges
*/
theSegmentList->segment = NEW_ARRAY(12*cusp_neighborhoods->its_triangulation->
num_tetrahedra, CuspNbhdSegment);

/*
 * Keep a pointer to the first empty CuspNbhdSegment.
 */
next_segment = theSegmentList->segment;

/*
 * Compute the Ford domain's vertices.
 */
for (tet = cusp_neighborhoods->its_triangulation->tet_list_begin.next;
     tet != &cusp_neighborhoods->its_triangulation->tet_list_end;
     tet = tet->next)
{
    x         = tet->cusp_nbhd_position->x;
    in_use    = tet->cusp_nbhd_position->in_use;

    for (v = 0; v < 4; v++)
    {
        /*
         * If this isn't the cusp the user wants, ignore it.
         */
        if (tet->cusp[v] != cusp)
            continue;

        for (h = 0; h < 2; h++)          /* h = right_handed, left_handed */
        {
            if (in_use[h][v] == FALSE)
                continue;

            /*
             * There are at least two ways to locate the Ford vertex.
             *
             * (1) Use Theorem 3.1 of
             *
             * M. Sakuma and J. Weeks, The generalized tilt
             * formula, Geometriae Dedicata 55 (115-123) 1995,
             *
             * which states that the Euclidean distance in the
             * cusp from (the projection of) the Ford vertex
             * to a side of the enclosing triangle equals the
             * tilt on that side. (Or better yet, see the
             * preprint version of the article, which has a lot
             * more pictures and fuller explanations.)
             *
             * (2) Write down the equations for the three planes
             * which lie halfway between the cusp at infinity
             * and the cusp at each of the three remaining ideal
             * vertices. Each such plane appears at a Euclidean
             * hemisphere. Subtracting the equations for two
             * such hemispheres gives a linear equation, and
             * two such linear equations may be solved
             * simultaneously to locate the Ford vertex.
             *
             * Either of the above approaches should work fine.
             * Here we choose approach (1) because it looks a tiny
             * bit simpler numerically.
             */

            /*
             * Label the triangles corners w[0], w[1] and w[2],
             * going counterclockwise around the triangle.
             *
             *           w[2]
             *          / \
             *         /   \
             *        /-----\
             *       w[0]-----w[1]
             *
             */

            w[0] = !v;

```

```
w[0] = !v;
```

```

if (h == right_handed)
{
    w[1] = remaining_face[w[0]][v];
    w[2] = remaining_face[v][w[0]];
}
else
{
    w[1] = remaining_face[v][w[0]];
    w[2] = remaining_face[w[0]][v];
}

/*
 * Record the triangle's corners.
 */
for (i = 0; i < 3; i++)
    corner[i] = complex_real_mult(cusp->displacement_exp, x[h][v][w[i]]);

/*
 *
 *          w[2]
 *        /  \
 *  -----/---*---\-----
 *        /      \
 *      w[0]-----w[1]
 *
 * The Ford vertex lies on a line parallel to a side of
 * the triangle at a distance "tilt" away (by Theorem 3.1
 * of Sakuma & Weeks). Find the equations of such lines
 * (the third is redundant -- it could perhaps be used
 * to enhance accuracy if desired).
 */
for (i = 0; i < 2; i++)
{
    /*
     * Make yourself a sketch as you follow along.
     */

    delta = complex_minus(corner[(i+1)%3], corner[i]);

    inward_normal.real = +delta.imag;
    inward_normal.imag = -delta.real;

    length = complex_modulus(inward_normal);

    tilt = tet->tilt[w[(i+2)%3]];

    offset = complex_real_mult(tilt/length, inward_normal);

    p = complex_plus(corner[i], offset);

    /*
     * The equation of the desired line is
     *
     *      y - p.imag      delta.imag
     *      ----- = -----
     *      x - p.real      delta.real
     *
     * Cross multiply to get
     *
     *      delta.imag * x - delta.real * y
     *      = delta.imag * p.real - delta.real * p.imag
     *
     * This last equation also has a natural cross product
     * interpretation: delta X (x,y) = p X (x,y).
     *
     * Record the equation as ax + by = c.
     */
    a[i] = delta.imag;
    b[i] = -delta.real;
    c[i] = delta.imag * p.real - delta.real * p.imag;
}

/*
 * Solve the matrix equation

```



```

        *
        *      ( a[0]  b[0] ) (x) = (c[0])
        *      ( a[1]  b[1] ) (y)   (c[1])
        *  =>
        *      (x) = _1_ ( b[1] -b[0] ) (c[0])
        *      (y)   det (-a[1]  a[0] ) (c[1])
        */

    det = a[0]*b[1] - a[1]*b[0];

    FORD_VERTEX(x,h,v).real = (b[1]*c[0] - b[0]*c[1]) / det;
    FORD_VERTEX(x,h,v).imag = (a[0]*c[1] - a[1]*c[0]) / det;
}
}

/*
 * Record the Ford domain edges.
 */
for (tet = cusp_neighborhoods->its_triangulation->tet_list_begin.next;
     tet != &cusp_neighborhoods->its_triangulation->tet_list_end;
     tet = tet->next)
{
    x      = tet->cusp_nbhd_position->x;
    in_use = tet->cusp_nbhd_position->in_use;

    for (v = 0; v < 4; v++)
    {
        /*
         * If this isn't the cusp the user wants, ignore it.
         */
        if (tet->cusp[v] != cusp)
            continue;

        for (h = 0; h < 2; h++) /* h = right_handed, left_handed */
        {
            if (in_use[h][v] == FALSE)
                continue;

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

                gluing = tet->gluing[f];

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

                /*
                 * We want to report each segment only once, so we
                 * make the (arbitrary) convention that we report
                 * a segment only from the Tetrahedron whose address
                 * in memory is less. In the case of a Tetrahedron
                 * glued to itself, we report it from the lower
                 * FaceIndex.
                 */
                if (tet > nbr_tet || (tet == nbr_tet && f > nbr_f))
                    continue;

                /*
                 * Don't report Ford edges dual to 2-cells which are
                 * part of the arbitrary subdivision of the canonical
                 * cell decomposition into tetrahedra. (They'd have
                 * length zero anyway, but we want to be consistent
                 * with how we report the triangulation. We rely on
                 * the fact that proto_canonize() has computed the
                 * tilts and left them in place. The sum of the tilts
                 * will never be positive for a subdivision of the
                 * canonical cell decomposition. If it's close to
                 * zero, ignore the Ford edge dual to that face.
                 */
                if (tet->tilt[f] + nbr_tet->tilt[nbr_f] > -CONCAVITY_EPSILON)
                    continue;
            }
        }
    }
}

```

```

    /*
     * This edge has passed all its tests, so record it.
     * Keep in mind that the coordinate systems in
     * neighboring Tetrahedra may differ by translations.
     */

    nbr_v = EVALUATE(gluing, v);
    nbr_h = (parity[gluing] == orientation_preserving) ? h : !h;

    next_segment->endpoint[0] = FORD_VERTEX(tet->cusp_nbhd_position->x,
    h, v);
    next_segment->endpoint[1] = FORD_VERTEX(nbr_tet->cusp_nbhd_position->x,
    nbr_h, nbr_v);

    /*
     * The segment indices are currently used only
     * for the triangulation, not the Ford domain.
     */
    next_segment->start_index = -1;
    next_segment->middle_index = -1;
    next_segment->end_index = -1;

    /*
     * Compensate for the (possibly) translated
     * coordinate systems. Compare the position of
     * a vertex u as seen by tet and nbr_tet.
     */

    u = remaining_face[v][f];
    nbr_u = EVALUATE(gluing, u);

    next_segment->endpoint[1] = complex_plus
    (
        next_segment->endpoint[1],
        complex_real_mult
        (
            cusp->displacement_exp,
            complex_minus
            (
                tet->cusp_nbhd_position->x[h][v][u],
                nbr_tet->cusp_nbhd_position->x[nbr_h][nbr_v][nbr_u]
            )
        )
    );

    /*
     * Move on.
     */
    next_segment++;
}
}
}

/*
 * How many segments did we find?
 *
 * (ANSI C will subtract the pointers correctly, automatically
 * dividing by sizeof(CuspNbhdSegment).)
 */
theSegmentList->num_segments = next_segment - theSegmentList->segment;

/*
 * Did we find more segments than we had allocated space for?
 * This should be impossible, but it doesn't hurt to check.
 */
if (theSegmentList->num_segments > 12*cusp_neighborhoods->its_triangulation->
num_tetrahedra)
    uFatalError("get_cusp_neighborhood_Ford_domain", "cusp_neighborhoods");

return theSegmentList;
}

```