

```

/*
 * length_spectrum.c
 *
 * This file provides the functions
 *
 * void length_spectrum(   WEPolyhedron    *polyhedron,
 *                        double            cutoff_length,
 *                        Boolean           full_rigor,
 *                        Boolean           multiplicities,
 *                        double            user_radius,
 *                        MultiLength      **spectrum,
 *                        int               *num_lengths);
 *
 * void free_length_spectrum(MultiLength *spectrum);
 *
 * length_spectrum() takes the following inputs:
 *
 * *polyhedron    The manifold whose length spectrum we're seeking is
 *                given as a Dirichlet domain. The Dirichlet domain
 *                may be computed either from a Triangulation, or
 *                directly from a set of generating matrices.
 *
 * cutoff_length  length_spectrum() reports geodesics of length
 *                at most cutoff_length.
 *
 * full_rigor     If full_rigor is TRUE, length_spectrum() guarantees
 *                that it will find all geodesics of length at most
 *                cutoff_length, with correct multiplicities if
 *                the multiplicities parameter is also TRUE.
 *
 * multiplicities If both full_rigor and multiplicities are TRUE,
 *                length_spectrum() reports complex lengths with
 *                correct multiplicities. If multiplicities is TRUE
 *                and full_rigor is FALSE, length_spectrum() reports
 *                the multiplicities as best it can, but doesn't
 *                promise they will be correct. If multiplicities is
 *                FALSE, length_spectrum() reports all multiplicities
 *                as zero.
 *
 *                Note: The geodesics' topologies are computed iff
 *                multiplicities is TRUE.
 *
 * user_radius    When full_rigor is FALSE, length_spectrum() tiles
 *                out to the user_radius instead of tiling out to
 *                the rigorous tiling_radius it would otherwise use.
 *                It may or may not find all geodesics of length up
 *                to cutoff_length. When full_rigor is TRUE, the
 *                user_radius parameter is ignored.
 *
 * length_spectrum provides the best results when both full_rigor and
 * multiplicities are TRUE. However, both these options slow the program
 * down, so the UI should offer the user the opportunity of doing
 * quick & dirty computations with one or both options off.
 *
 * length_spectrum() provides the following outputs:
 *
 * spectrum       *spectrum is set to point to an array
 *                of MultiLengths.
 *
 * num_lengths    *num_lengths is set to the number of
 *                elements in the array.
 *
 * length_spectrum allocates the array *spectrum. When you are done with
 * it, please call free_length_spectrum() to deallocate the memory it
 * occupies.
 *
 * Theory.
 *
 * *****
 * 95/10/31
 * The results in this Theory section have appeared in print as
 *
 * C. Hodgson and J. Weeks, Symmetries, isometries and

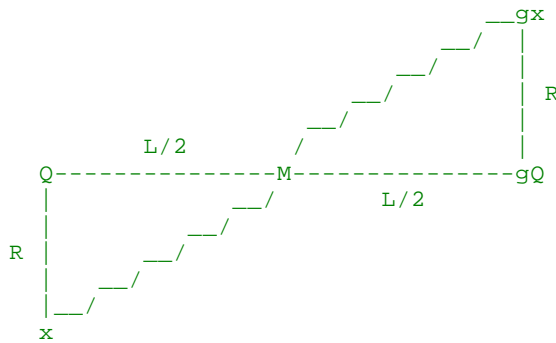
```

```

*          length spectra of closed hyperbolic 3-manifolds,      *
*          Experimental Mathematics 3 (1994) 261-274.            *
*                                                                *
*****
*
* The remainder of this top-of-file documentation proves some lemmas
* which will be needed in the code below.  I offer my deepest thanks to
* Craig Hodgson for his long-term collaboration on this work -- in
* particular for providing the key ideas in some of the following
* lemmas -- and also for his long-term friendship.
*
* Terminology.  Throughout this file D will be a Dirichlet domain
* with basepoint x.  A typical translate of D will be denoted gD,
* where g is an isometry in the group of covering transformations.
*
* We will be tiling hyperbolic space with copies of the Dirichlet domain D.
* In particular, we'll need to find all translates gD which move the
* basepoint x a distance less than some given distance s, i.e. we'll want
* to find all gD such that  $d(x, gx) < s$ .  The simplest algorithm is to
* start with D and recursively attach its neighbors, stopping the
* recursion when we reach translates gD whose basepoints are a distance
* greater than s from the origin, i.e. when  $d(x, gx) > s$ .  For an
* arbitrary fundamental domain (not necessarily a Dirichlet domain)
* with an arbitrary basepoint, this algorithm might fail: there could
* be a translate with basepoint a distance less than s from the origin,
* all of whose neighbors have basepoints a distance greater than s
* from the origin.  The simple recursive algorithm would not find such
* a translate.  Fortunately this cannot occur for a Dirichlet domain.
*
* Lemma 1 (Craig Hodgson).  Let D be a Dirichlet domain with basepoint x,
* and let gD be a translate of D such that for all neighbors hD of gD,
*  $d(x, hx) \geq d(x, gx)$ .  Then  $g = \text{identity}$ .
*
* Proof.  For each neighbor hD of gD, the inequality  $d(x, hx) \geq d(x, gx)$ 
* implies that x lies in the halfspace  $H_h$  consisting of points closer
* to gx than hx.  But gD is the intersection of all such  $H_h$ , so x must
* lie in gD.  Therefore  $gD = D$ .  Q.E.D.
*
* Each translate of the Dirichlet domain corresponds to an isometry
*  $H^3 \rightarrow H^3$  in the group of covering transformations, and each such
* isometry defines a complex length.  Please see complex_length.c for
* details on how the isometry determines the length.  We consider
* only hyperbolic and loxodromic isometries.  Elliptics and parabolics
* are not reported.
*
* Lemma 2.  To find all closed geodesics of length  $\leq L$ , it suffices
* to find all translates gD satisfying  $d(x, gx) < L + 2R$ , where R is the
* spine_radius defined in winged_edge.h and computed in
* compute_spine_radius() in Dirichlet_extras.c.
*
* Proof.  A closed geodesic must intersect a spine of D at some point P,
* because otherwise it would be a parabolic or a trivial curve.  Let g be
* the covering transformation corresponding to given geodesic.  Then
*  $d(P, gP) = L$ , and  $d(x, gx) \leq d(x, P) + d(P, gP) + d(gP, gx) \leq R + L + R$ .
*
* (Yes, I realize that the spine_radius is an infimum which may not be
* realized.  If you want to fill in the missing the details, you may
* replace R with  $R + \epsilon$ , and then let  $\epsilon$  go to zero.)
*
* Q.E.D.
*
* We can improve a bit on the estimate of  $L + 2R$ .
*
* Lemma 2'.  To find all closed geodesics of length  $\leq L$ , it suffices
* to find all translates gD satisfying  $d(x, gx) < 2 \operatorname{acosh}(\cosh R \cosh L/2)$ ,
* where R is the spine_radius defined in winged_edge.h and computed in
* compute_spine_radius() in Dirichlet_extras.c.
*
* Proof.  Consider the point Q where the geodesic passes closest to
* the basepoint x.  Because the geodesic is known to intersect the
* spine,  $d(x, Q) \leq d(x, P) \leq R$  (P is as in the proof of Lemma 2 above).
* The advantage of working with Q instead of P is that the segment from
* x to Q is orthogonal to the geodesic.  (As a special case Q could equal
* x, but our formula still holds.)  Let the point M be the midpoint

```

\* of the segment from Q to gQ.



\* The hyperbolic law of cosines bounds the distance from x to M  
 \* as  $\text{acosh}(\cosh R \cosh L/2)$ . The distance from M to gx is the same,  
 \* so the distance from x to gx is bounded by  $2 \text{acosh}(\cosh R \cosh L/2)$ .  
 \* Q.E.D.

\* Comment. Lemma 2' offers a 16-fold improvement over Lemma 2 when  
 \* R and L are large.

\* Proof of comment. (Note: I haven't checked this proof as carefully  
 \* as I checked the proofs of the official lemmas, but I think it's  
 \* basically correct.) For sufficiently large arguments,  
 \*  $\cosh(a) \sim \exp(a)/2$  and  $\text{acosh}(b) \sim \log(2b)$ . So  $2 \text{acosh}(\cosh R \cosh L/2)$   
 \*  $\sim 2 \log(2 \exp(R)/2 \exp(L/2)/2) = 2(R + L/2 - \log(2)) = L + 2R - 2\log(2)$ .  
 \* In other words, as L and R go to infinity, the bound offered by Lemma 2'  
 \* is  $2\log(2)$  less than the bound offered by Lemma 2. How much is this  
 \* improvement worth? The number of images we compute within a ball of  
 \* radius r is roughly proportional to the ball's volume. The area of  
 \* a ball of radius r in  $H^3$  is  $A = 4\pi (\sinh r)^2$ , which for large r  
 \* is about  $4\pi (\exp(r)/2)^2 = \pi \exp(2r)$ . So the ball's volume is  
 \* about  $(\pi/2) \exp(2r)$ . The ratio of the volumes of balls of radius  
 \* r and r' is  $\exp(2r')/\exp(2r) = \exp(2(r' - r))$ , which in the present  
 \* case is  $\exp(2(2\log 2)) = 2^4 = 16$ . Q.E.D.

\* The above lemmas assure us that we've found all group elements  
 \* corresponding to geodesics of length at most L. Now we consider  
 \* how best to remove the vast numbers of duplicates on our list, i.e.  
 \* the vast numbers of distinct group elements which are conjugate to  
 \* one another and therefore represent the same geodesic in the manifold.  
 \* We take a two-part strategy:

\* (1) We remove all group elements whose axes don't pass within  
 \* a distance R of the basepoint, where R is the spine radius  
 \* as above. (Every geodesic must intersect the spine, so we  
 \* are sure to retain at least one element in every conjugacy  
 \* class.) The documentation in `distance_to_origin()` below says  
 \* how the distance from an axis to the basepoint is computed.

\* (2) We check the remaining group elements for conjugacy, using  
 \* Lemma 3' below.

\* As with Lemmas 2 and 2', we first prove a simpler version of Lemma 3,  
 \* and then refine it to Lemma 3'.

\* Lemma 3. If g and g' are conjugate group elements each of whose axes  
 \* passes within a distance R of the basepoint, then there is a group  
 \* element h such that (1)  $g = h(g')(h^{-1})$  and (2) h moves the basepoint  
 \* a distance at most  $L/2 + 2R$ .

\* Proof. Let A (resp. A') be the axis of g (resp. g'), and let Q  
 \* (resp. Q') be the point on A (resp. A') closest to the basepoint x.  
 \* There are infinitely many covering transformations taking A to A';  
 \* let h be one which minimizes the distance from hQ to Q'. Because  
 \* the length of the underlying geodesic is at most L, the distance  
 \* from hQ to Q' is at most L/2 (it's L/2 and not L because if  
 \*  $L/2 < d(hQ, Q') < L$  then you've got the wrong h -- you need to  
 \* consider an h which takes Q to a point on the other side of Q').  
 \* It's easy to get a bound on the distance h moves the basepoint x:



```

/* When counting multiplicities, two complex lengths are considered
 * equal iff their real and imaginary parts agree to within
 * DUPLICATE_LENGTH_EPSILON. We use a fairly small value for
 * DUPLICATE_LENGTH_EPSILON so that we can resolve geodesics whose lengths
 * are equal in a cusped manifold but slightly different after a high-order
 * Dehn filling. This entails a risk that in low-accuracy situations we
 * might show equal lengths as different, but the user should be able to
 * figure out what's going on.
 */
#define DUPLICATE_LENGTH_EPSILON    1e-8

/*
 * CONJUGATOR_EPSILON provides an extra margin of safety to insure that
 * we find all relevant conjugators in eliminate_its_conjugates().
 * (There is also an epsilon added to the spine_radius used there.)
 */
#define CONJUGATOR_EPSILON          1e-3

/*
 * Two geodesics are checked for conjugacy iff their lengths differ by
 * at most POSSIBLE_CONJUGATE_EPSILON. We choose a fairly large value for
 * POSSIBLE_CONJUGATE_EPSILON to insure that no conjugacies are missed.
 * The only possible harm a large value can do is slow down the algorithm
 * a bit as it does some "unnecessary" checks when geodesics of slightly
 * different lengths are present (as in a high-order Dehn filling).
 */
#define POSSIBLE_CONJUGATE_EPSILON  1e-3

/*
 * The tiling of hyperbolic space by translates gD of the Dirichlet domain
 * is stored on a binary tree. Each node in the tree is a Tile structure
 * containing the group element g associated with the given translate.
 */

typedef struct Tile
{
    /*
     * A translate gD of the Dirichlet domain is determined
     * by the group element g.
     */
    O31Matrix      g;

    /*
     * Please see complex_length.c for details on how the
     * complex length is defined and computed.
     */
    Complex        length;

    /*
     * Is the group element g an orientation_preserving
     * or orientation_reversing isometry?
     */
    MatrixParity    parity;

    /*
     * Is the geodesic topologically a circle or a mirrored interval?
     */
    Orbifold1       topology;

    /*
     * The to_be_eliminated flag is used locally in eliminate_powers()
     * and eliminate_conjugates().
     */
    Boolean         to_be_eliminated;

    /*
     * The tiles are organized in two different way.
     */

    /*
     * Organization #1.
     */

```

```

    * To make checking for duplicates easy, the Tiles are kept on
    * a binary tree. The sort and search key is a more or less
    * arbitrary function defined in the code. The next_subtree field
    * is used locally within already_on_tree() and free_tiling()
    * to avoid doing recursions on the system stack; the latter run
    * the risk of stack/heap collisions.
    */
    struct Tile      *left_child,
                   *right_child;
    double          key;
    struct Tile      *next_subtree;

    /*
    * Organization #2.
    *
    * The function tile() needs to keep track of which Tiles have
    * not yet had their neighbors checked. Its keeps pending Tiles
    * on a doubly-linked list.
    */
    struct Tile      *prev,
                   *next;
} Tile;

static void          tile(WEPolyhedron *polyhedron, double tiling_radius, Tile **tiling);
static double        key_value(O3lMatrix m);
static Boolean       already_on_tree(Tile *root, Tile *tile);
static void          add_to_tree(Tile *root, Tile *tile);
static int           count_translates(Tile *root);
static void          find_good_geodesics(Tile *tiling, int num_translates, Tile ***
    geodesic_list, int *num_good_geodesics, double cutoff_length, double spine_radius);
static Boolean       tile_is_good(Tile *tile, double cutoff_length, double spine_radius);
static double        distance_to_origin(Tile *tile);
static void          sort_by_length(Tile **geodesic_list, int num_good_geodesics);
static int CDECL     compare_lengths(const void *tile0, const void *tile1);
static void          eliminate_powers(Tile **geodesic_list, int *num_good_geodesics, double
    cutoff_length);
static void          eliminate_its_powers(Tile **geodesic_list, int num_good_geodesics, int
    i0, double cutoff_length);
static void          eliminate_conjugates(Tile **geodesic_list, int *num_good_geodesics,
    Tile *tiling, int num_translates, double spine_radius);
static void          make_conjugator_list(Tile ***conjugator_list, int *num_conjugators,
    Tile *tiling, int num_translates);
static void          add_conjugators_to_list(Tile *root, Tile **conjugator_list, int *
    num_conjugators);
static int CDECL     compare_translation_distances(const void *tile0, const void *tile1);
static void          initialize_elimination_flags(Tile **geodesic_list, int
    num_good_geodesics);
static void          eliminate_its_conjugates(Tile **geodesic_list, int num_good_geodesics,
    int i0, Tile **conjugator_list, int num_conjugators, double spine_radius);
static void          compress_geodesic_list(Tile **geodesic_list, int *num_good_geodesics);
static Boolean       is_manifold_orientable(WEPolyhedron *polyhedron);
static void          copy_lengths(Tile **geodesic_list, int num_good_geodesics, MultiLength
    **spectrum, int *num_lengths, Boolean multiplicities, Boolean manifold_is_orientable);
static void          free_tiling(Tile *root);

void length_spectrum(
    WEPolyhedron    *polyhedron,
    double          cutoff_length,
    Boolean          full_rigor,
    Boolean          multiplicities,
    double          user_radius,
    MultiLength      **spectrum,
    int              *num_lengths)
{
    Tile            *tiling,
                   **geodesic_list;
    int             num_translates,
                   num_good_geodesics;

    /*
    * If full_rigor is TRUE,

```

```

*      we want to find all closed geodesics of length at most
*      cutoff_length.  By Lemma 2' above, it suffices to find all
*      translates gD satisfying  $d(x, gx) \leq 2 \operatorname{arccosh}(\cosh(R) * \cosh(L/2))$ .
*      If full_rigor is FALSE,
*      tile out to the user_radius and hope for the best.
*/
tile(
    polyhedron,
    full_rigor ?
        2 * arccosh( cosh(polyhedron->spine_radius) * cosh(cutoff_length/2) ) :
        user_radius,
    &tiling);

/*
*   How many translates did we find?
*/
num_translates = count_translates(tiling);

/*
*   Make a list of all group elements satisfying the following
*   three conditions:
*
*   (1) The real part of the complex length is greater than zero.
*       In the orientation-preserving case this means the isometry
*       is hyperbolic or loxodromic.  In the orientation-reversing
*       case, it's a glide reflection.  Either way, the isometry
*       factors as a translation along an axis, followed by a
*       (possibly trivial) rotation or a reflection.
*
*   (2) The translation distance along the axis is at most
*       cutoff_length (plus epsilon).
*
*   (3) The distance from the axis to the basepoint is at most
*       polyhedron->spine_radius.  Every geodesic intersects the
*       spine, so we can't lose any geodesics this way, although
*       we will happily lose some unnecessary conjugates.  In any
*       case, this condition is necessary for checking for conjugacy
*       later on.
*
*   find_good_geodesics() will allocate an array of pointers of
*   type (Tile *), and write in the addresses of all group elements
*   satisfying the above three conditions.  It will report the number
*   of such pointers in num_good_geodesics.
*/
find_good_geodesics(    tiling,
                        num_translates,
                        &geodesic_list,
                        &num_good_geodesics,
                        cutoff_length + LENGTH_EPSILON,
                        polyhedron->spine_radius + SPINE_EPSILON);

/*
*   Sort the geodesic_list by order of increasing lengths.
*/
sort_by_length(geodesic_list, num_good_geodesics);

/*
*   We want only primitive group elements.  Discard elements which are
*   squares or higher powers of the primitives.  (We'll still have two
*   group elements for each lift of a geodesic -- they'll be inverses
*   of one another -- but we'll deal with them later after we've checked
*   for conjugacy.)
*/
eliminate_powers(geodesic_list, &num_good_geodesics, cutoff_length + 2*LENGTH_EPSILON);

/*
*   If multiplicities is TRUE, we want to retain precisely one
*   element on the geodesic_list corresponding to each geodesic
*   in the manifold.
*   We cull the geodesic_list in place, moving good pointers
*   toward the beginning of the list to take the place of pointers
*   which have been removed.
*/
if (multiplicities == TRUE)

```

```

        eliminate_conjugates(    geodesic_list,
                                &num_good_geodesics,
                                tiling,
                                num_translates,
                                polyhedron->spine_radius + CONJUGATE_SPINE_EPSILON);

/*
 * Allocate space for the spectrum, copy in the lengths, parities,
 * topologies and multiplicities, and report its size.
 *
 * If multiplicities == FALSE, set all multiplicities to zero.
 *
 * If no lengths are present, set *spectrum = NULL.
 */
copy_lengths(    geodesic_list,
                num_good_geodesics,
                spectrum,
                num_lengths,
                multiplicities,
                is_manifold_orientable(polyhedron));

/*
 * Free local storage.
 */
my_free(geodesic_list);
free_tiling(tiling);
}

void free_length_spectrum(
    MultiLength *spectrum)
{
    if (spectrum != NULL)
        my_free(spectrum);
}

static void tile(
    WEPolyhedron    *polyhedron,
    double          tiling_radius,
    Tile            **tiling)
{
    Tile    queue_begin,
            queue_end,
            *identity,
            *tile,
            *nbr;
    double  cosh_tiling_radius;
    WEFace  *face;

    /*
     * Assorted methodological comments:
     *
     * When tiling one needs a way to decide whether the neighbors
     * of a given lift have already been found. One could work out
     * a fancy constant-time algorithm, but it's best just to keep all
     * the lifts we've found so far on a binary tree, and check the tree
     * whenever we need to know whether a given lift has already been
     * found. The search runs in log n time, where  $n \sim e^r$  and r is the
     * radius we're tiling to, so each query will require about
     *  $\log(e^r) \sim r$  comparisons. This is a reasonable price to pay.
     * The constant time algorithm would be messy to write. The tree
     * algorithm is easy to write, robust, and fairly fast, since each of
     * the log n operations is completely trivial (it might even beat the
     * constant time algorithm for all relevant cases). Even when  $n \sim 1e6$ 
     * the tree depth will be only  $1.4 * \log_2(1e6) \sim 1.4 * 20 \sim 30$ .
     *
     * We also "waste" a factor of two by letting a lift A discover that
     * one of its neighbors is some other lift B, and then later on letting
     * lift B discover that one of its neighbors is lift A. A fancier
     * algorithm would have lift B remember that lift A is its neighbor
     * on a certain face, and avoid the duplication of effort. But I felt
     * that such an algorithm would be harder to write and maintain, so I
     * chose the simpler algorithm instead.
    */

```



```

*
* If users tend to send the algorithm off on long, long computations,
* we could include a parameter which puts a maximum on the number
* of lifts it's willing to compute. But I haven't done this.
* Eventually, though, the computation will use the services of
* a long-computation-in-progress facility, which will allow aborts.
*/

/*
* Initialize the doubly-linked list.
*
* Tiles will be added to the list as soon as they are computed, and
* removed from the list once all their neighbors have been computed.
*/
queue_begin.prev = NULL;
queue_begin.next = &queue_end;
queue_end.prev = &queue_begin;
queue_end.next = NULL;

/*
* Create a Tile for the identity element.
*/

identity = NEW_STRUCT(Tile);
o3l_copy(identity->g, O3l_identity);
identity->length = Zero;
identity->parity = orientation_preserving;
identity->topology = orbifold1_unknown;

/*
* Put the identity on the binary tree.
*/

identity->left_child = NULL;
identity->right_child = NULL;
identity->key = key_value(O3l_identity);
identity->next_subtree = NULL;

*tiling = identity;

/*
* Put the identity on the double-linked list.
*/

INSERT_BEFORE(identity, &queue_end);

/*
* Compute cosh(tiling_radius + TILING_EPSILON) for later convenience.
*/

cosh_tiling_radius = cosh(tiling_radius + TILING_EPSILON);

/*
* We're ready to roll. Our algorithm is
*
* while (the queue is not empty)
*     pull a Tile off the beginning of the queue
*     for each of its neighbors
*         if (the neighbor is not beyond the tiling radius
*             && the neighbor is not already on the tree)
*             add the neighbor to the tree
*             add the neighbor to the end of the queue
*/

/*
* While the queue is not empty . . .
*/
while (queue_begin.next != &queue_end)
{
    /*
    * Pull a Tile off the beginning of the queue.
    */
    tile = queue_begin.next;
    REMOVE_NODE(tile);

```

```

/*
 * For each of its neighbors . . .
 */
for (face = polyhedron->face_list_begin.next;
     face != &polyhedron->face_list_end;
     face = face->next)
{
    /*
     * Tentatively allocate a Tile.
     */
    nbr = NEW_STRUCT(Tile);

    /*
     * Compute the neighbor's group element and key value.
     */
    o3l_product(tile->g, *face->group_element, nbr->g);
    nbr->key = key_value(nbr->g);
    nbr->next_subtree = NULL;

    /*
     * If nbr->g is not too far away and not already on the tree,
     * we compute its length and parity, initialize its topology,
     * and add it to both the tree and the queue.
     * Otherwise we discard it.
     */
    if (nbr->g[0][0] < cosh_tiling_radius
        && already_on_tree(*tiling, nbr) == FALSE)
    {
        nbr->length = complex_length_o3l(nbr->g);
        nbr->parity = gl4R_determinant(nbr->g) > 0.0 ?
                     orientation_preserving :
                     orientation_reversing;
        nbr->topology = orbifold1_unknown;
        add_to_tree(*tiling, nbr);
        INSERT_BEFORE(nbr, &queue_end);
    }
    else
    {
        /*
         * Either the neighbor was beyond the tiling_radius or
         * already on the binary tree. Either way we discard it.
         */
        my_free(nbr);
    }
}
}

static double key_value(
    O3lMatrix m)
{
    /*
     * Define a sort key for storing Tiles on the binary tree.
     * Ideally we'd like a key with the property that nearby group
     * elements (differing only by roundoff error) will have close
     * key values, and distant group elements will have distant key
     * values. But homeomorphisms from  $\mathbb{R}^6$  to  $\mathbb{R}$  are impossible.
     * So we try for the next best thing, a key that usually maps
     * distant group elements to distant key values. First note that
     * since the origin lies in the interior of the Dirichlet domain
     * and away from the singular set, the group elements are uniquely
     * determined by where they map the origin. Furthermore, the x[0]
     * coordinate of any point in hyperbolic space (in the Minkowski
     * space model) is completely determined by the x[1], x[2] and x[3]
     * coordinates. So for our sort key we use a random looking linear
     * combination of the x[1], x[2] and x[3] coordinates. The reason
     * for making it random looking is to minimize the chances that
     * distinct images of the basepoint will lie on the same level surface
     * of the sort key function.
     *
     * There could conceivably be problems in guessing the roundoff
     * error in the key values, because the possible values of m[][]
     * span several orders of magnitude. I'll have to think some more
     * about that.
    */
}

```

```

    *
    * Recall that the first column of an O3lMatrix gives the image
    * of the origin.
    */

    return( m[1][0] * 0.47865745183883625637
           + m[2][0] * 0.14087522034920476458
           + m[3][0] * 0.72230196622481940253);
}

static Boolean already_on_tree(
    Tile      *root,
    Tile      *tile)
{
    Tile      *subtree_stack,
              *subtree;
    double    delta;
    Boolean    left_flag,
              right_flag;

    /*
     * Reliability is our first priority. Speed is second.
     * So if tile->key and root->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 = root;
    if (root != NULL)
        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 tile and the subtree's root.
         */
        delta = tile->key - subtree->key;

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

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

```

```

    /*
     * Check this Tile if the key values match.
     */
    if (left_flag && right_flag)
        if (o3l_equal(subtree->g, tile->g, ISOMETRY_EPSILON))
            return TRUE;
    }

    return FALSE;
}

static void add_to_tree(
    Tile    *root,
    Tile    *tile)
{
    /*
     * already_on_tree() has already checked that tile->g does not
     * appear on the tree. So here we just add it in the appropriate
     * spot, based on the key value.
     */

    Tile    **location;

    location = &root;

    while (*location != NULL)
    {
        if (tile->key <= (*location)->key)
            location = &(*location)->left_child;
        else
            location = &(*location)->right_child;
    }

    *location = tile;

    tile->left_child    = NULL;
    tile->right_child   = NULL;
}

static int count_translates(
    Tile    *root)
{
    Tile    *subtree_stack,
            *subtree;
    int      num_translates;

    /*
     * Implement the tree traversal 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 = root;
    if (root != NULL)
        root->next_subtree = NULL;

    /*
     * Initialize the count to zero.
     */
    num_translates = 0;

    /*
     * 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_child != NULL)
        {
            subtree->left_child->next_subtree = subtree_stack;
            subtree_stack = subtree->left_child;
        }
        if (subtree->right_child != NULL)
        {
            subtree->right_child->next_subtree = subtree_stack;
            subtree_stack = subtree->right_child;
        }

        /*
         * Count the subtree's root node.
         */
        num_translates++;
    }

    return num_translates;
}

static void find_good_geodesics(
    Tile      *tiling,
    int       num_translates,
    Tile      ***geodesic_list,
    int       *num_good_geodesics,
    double    cutoff_length,
    double    spine_radius)
{
    Tile      *subtree_stack,
              *subtree;

    /*
     * The most good geodesics we could have would be num_translates,
     * so we'll allocate a geodesic_list of this length, even though
     * we probably won't use all its entries.
     */
    *geodesic_list = NEW_ARRAY(num_translates, Tile *);

    /*
     * *num_good_geodesics will keep track of how many pointers have
     * been put on the geodesic_list. Initialize it to zero.
     */
    *num_good_geodesics = 0;

    /*
     * Implement the recursive counting 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 tiling tree.
     */
    subtree_stack = tiling;
    if (tiling != NULL)
        tiling->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_child != NULL)
    {
        subtree->left_child->next_subtree = subtree_stack;
        subtree_stack = subtree->left_child;
    }
    if (subtree->right_child != NULL)
    {
        subtree->right_child->next_subtree = subtree_stack;
        subtree_stack = subtree->right_child;
    }

    /*
     * If the subtree's root is a good tile, add it
     * to the geodesic_list.
     */
    if (tile_is_good(subtree, cutoff_length, spine_radius))
        (*geodesic_list)[(*num_good_geodesics)++] = subtree;
}

static Boolean tile_is_good(
    Tile      *tile,
    double    cutoff_length,
    double    spine_radius)
{
    /*
     * tile_is_good() tests the three conditions given in
     * length_spectrum() above. It's essential that we test Condition #3
     * after Condition #1, because the distance to the axis is undefined
     * for parabolics. We test Condition #2 before Condition #3 because
     * it's a little faster computationally.
     */

    /*
     * Condition #1. Is tile->length.real > 0?
     */
    if (tile->length.real < LENGTH_EPSILON)
        return FALSE;

    /*
     * Condition #2. Does the isometry translate its axis a distance
     * less than cutoff_length?
     *
     * length_spectrum() has already added LENGTH_EPSILON to cutoff_length.
     */
    if (tile->length.real > cutoff_length)
        return FALSE;

    /*
     * Condition #3. Does the axis pass within a distance
     * spine_radius of the origin?
     *
     * length_spectrum() has already added SPINE_EPSILON to spine_radius.
     */
    if (distance_to_origin(tile) > spine_radius)
        return FALSE;

    /*
     * All three conditions are satisfied, so return TRUE.
     */
    return TRUE;
}

```

```

static double distance_to_origin(
    Tile      *tile)
{
    Tile      square;
    double    cosh_d,
              cosh_s,
              cos_t;

    /*
     * tile_is_good() has already checked Condition #1, so we know
     * that we're looking at a translation along a geodesic, perhaps
     * followed by a rotation or reflection about that geodesic.
     * (See complex_length.c for the full classification of isometries.)
     * In the orientation-preserving case the isometry tile->g is
     * conjugate to
     *
     *      cosh s  sinh s   0   0
     *      sinh s  cosh s   0   0
     *      0       0       cos t -sin t
     *      0       0       sin t  cos t
     *
     * while in the orientation-reversing case it's conjugate to
     *
     *      cosh s  sinh s   0   0
     *      sinh s  cosh s   0   0
     *      0       0       -1   0
     *      0       0       0   1
     *
     * The latter case is hard to handle here, so if tile->g is
     * orientation-reversing, we compute its square. The square will
     * of course be orientation-preserving and have the same axis as
     * tile->g, so we may use it instead of tile->g itself.
     */

    if (tile->parity == orientation_reversing)
    {
        o3l_product(tile->g, tile->g, square.g);
        square.length.real = 2 * tile->length.real;
        square.length.imag = 0.0;
        square.parity      = orientation_preserving;

        return distance_to_origin(&square);
    }

    /*
     * We may now assume the isometry is orientation-preserving,
     * so in some coordinate system it takes the form
     *
     *      cosh s  sinh s   0   0
     *      sinh s  cosh s   0   0
     *      0       0       cos t -sin t
     *      0       0       sin t  cos t
     *
     * In this same coordinate system we may, without loss of generality,
     * assume that the basepoint lies at (cosh r, 0, sinh r, 0), where
     * r is the distance from the basepoint to the axis of the isometry.
     * The image of the basepoint under the isometry is then
     *
     *      ( cosh s  sinh s   0   0 ) (cosh r)   (cosh r cosh s)
     *      ( sinh s  cosh s   0   0 ) ( 0 ) = (cosh r sinh s)
     *      ( 0       0       cos t -sin t ) (sinh r) (sinh r cos t )
     *      ( 0       0       sin t  cos t ) ( 0 )   (sinh r sin t )
     *
     * The distance d which the isometry moves the basepoint may be
     * computed using the formula -cosh d = <basepoint, g(basepoint)>.
     *
     * - cosh d =
     *   < (cosh r, 0, sinh r, 0),
     *     (cosh r cosh s, cosh r sinh s, sinh r cos t, sinh r sin t) >
     * = - cosh^2 r cosh s + sinh^2 r cos t
     *
     * Fortunately we already know
     *
     *      cosh d = tile->g[0][0]
    
```

```

    *      s      = tile->length.real
    *      t      = tile->length.imag
    *
    * so we may solve for r.
    *
    *      r = acosh(sqrt( ( cosh d - cos t ) / ( cosh s - cos t ) ))
    *
    * Note that the argument of the sqrt() function is, in theory,
    * always at least one.
    */

cosh_d = tile->g[0][0];
cosh_s = cosh(tile->length.real);
cos_t  = cos(tile->length.imag);

/*
 * Make sure cosh d really is greater than cosh s, even accounting
 * for roundoff error.
 */
if (cosh_d < cosh_s)
{
    if (cosh_d > cosh_s - COSH_EPSILON)
    /*
     * The error is small, so we assume cosh d should equal
     * cosh s, and we return r = 0.0.
     */
        return 0.0;
    else
    /*
     * The error is large.  Something went wrong.
     */
        uFatalError("distance_to_origin", "length_spectrum");
}

/*
 * Use the above formula to solve for r, and return the answer.
 */
return arccosh(safe_sqrt( ( cosh_d - cos_t ) / ( cosh_s - cos_t ) ));
}

static void sort_by_length(
    Tile      **geodesic_list,
    int       num_good_geodesics)
{
    /*
     * Sort the elements on the geodesic_list by order of increasing length.
     *
     * Probably all implementations of qsort() would handle the case
     * num_good_geodesics == 0 correctly, but why take chances?
     */

    if (num_good_geodesics > 0)
    {
        qsort( geodesic_list,
                num_good_geodesics,
                sizeof(Tile *),
                compare_lengths);
    }
}

static int CDECL compare_lengths(
    const void *tile0,
    const void *tile1)
{
    /*
     * This comparison function does not put a well-defined linearing
     * ordering on the set of all complex lengths, nor could it possibly
     * do so in any reasonable way.  (Exercise for the reader: Find three
     * complex lengths a, b and c such that this function reports a < b,
     * b < c and c < a.) But as long as roundoff errors are less than
     * DUPLICATE_LENGTH_EPSILON and the differences between the true
     * lengths of geodesics are greater than DUPLICATE_LENGTH_EPSILON,
     * it should work fine.
     */

```



```

    */

    Complex diff;

    diff = complex_minus((*Tile **)tile0->length, (*Tile **)tile1->length);

    if (diff.real < -DUPLICATE_LENGTH_EPSILON)
        return -1;

    if (diff.real >  DUPLICATE_LENGTH_EPSILON)
        return +1;

    if (diff.imag < 0.0)
        return -1;

    if (diff.imag > 0.0)
        return +1;

    return 0;
}

static void eliminate_powers(
    Tile      **geodesic_list,
    int       *num_good_geodesics,
    double    cutoff_length)
{
    int i;

    /*
     * Initialize the tile->to_be_eliminated flags to FALSE.
     */
    initialize_elimination_flags(geodesic_list, *num_good_geodesics);

    /*
     * Look at each element on the geodesic list in turn.
     *
     * If it's to_be_eliminated, skip it.
     *
     * Otherwise, if any of its powers appear on the geodesic_list,
     * mark them to_be_eliminated.
     */

    for (i = 0; i < *num_good_geodesics; i++)

        if (geodesic_list[i]->to_be_eliminated == FALSE)

            eliminate_its_powers(    geodesic_list,
                                     *num_good_geodesics,
                                     i,
                                     cutoff_length);

    /*
     * Compress the geodesic_list by eliminating pointers to tiles which
     * are to_be_eliminated. The remaining good pointers move towards
     * the beginning of the list, overwriting the eliminated pointers.
     */
    compress_geodesic_list(geodesic_list, num_good_geodesics);
}

static void eliminate_its_powers(
    Tile      **geodesic_list,
    int       num_good_geodesics,
    int       i0,                /* index of geodesic under consideration */
    double    cutoff_length)     /* 2*LENGTH_EPSILON has already been added in */
{
    Tile      the_power;
    int       i;

    /*
     * Look at each power n > 1 of geodesic_list[i0]->g whose length
     * is less than the cutoff_length.
     */

```

```

for
(
    o3l_product(geodesic_list[i0]->g, geodesic_list[i0]->g, the_power.g),
    the_power.length.real = 2 * geodesic_list[i0]->length.real;

    the_power.length.real < cutoff_length;

    o3l_product(the_power.g, geodesic_list[i0]->g, the_power.g),
    the_power.length.real += geodesic_list[i0]->length.real
)
/*
 * Does any element of the geodesic_list correspond to the_power?
 */
for
(
    i = i0 + 1;
    i < num_good_geodesics
    && geodesic_list[i]->length.real
        < the_power.length.real + LENGTH_EPSILON;
    i++
)
    if (geodesic_list[i]->length.real
        > the_power.length.real - LENGTH_EPSILON)
    {
        if (o3l_equal(geodesic_list[i]->g, the_power.g, ISOMETRY_EPSILON) == TRUE)
        {
            geodesic_list[i]->to_be_eliminated = TRUE;
            break;
        }
    }
}

static void eliminate_conjugates(
    Tile    **geodesic_list,
    int     *num_good_geodesics,
    Tile    *tiling,
    int     num_translates,
    double  spine_radius)
{
    int     i;

    /*
     * Our task is to recognize which elements of the geodesic_list are
     * conjugate to one another, and eliminate the duplicates, keeping
     * exactly one element of each conjugacy class. We know that
     * each element on the geodesic_list corresponds to a geodesic with
     *  $0 < \text{length} \leq L$ , and its axis passes within a distance  $R$  of the
     * basepoint. So Lemma 3' at the top of this file says that
     * any conjugating elements we need will move the basepoint a distance
     * at most  $2 \operatorname{acosh}(\cosh R \cosh L/4)$ . Fortunately the tiling contains
     * all group elements moving the basepoint a distance at most
     *  $2 \operatorname{acosh}(\cosh R \cosh L/2)$ , so we have all the conjugators we need,
     * and then some.
     *
     * Actually, we take the culling a step further, and make sure we
     * have exactly one Tile for each geodesic. That is, whenever two
     * distinct conjugacy classes correspond to the same geodesic
     * (with opposite directions), we keep only one.
     */

    Tile    **conjugator_list;
    int     num_conjugators;

    /*
     * If the geodesic_list is empty, there is no work to be done.
     * The subsequent code would actually run fine even with an empty
     * geodesic_list, but better not to take chances, just in case
     * someday I make modifications.
     */
    if (*num_good_geodesics == 0)
        return;

    /*

```

```

    * Organize the possible conjugators on a list.
    * That is, make a list of (Tile *) pointers, with one pointer to
    * each element in the tiling tree. Sort the list by order of
    * increasing basepoint translation distance d(x,g(x0)).
    */
make_conjugator_list(&conjugator_list, &num_conjugators, tiling, num_translates);

/*
 * Initialize the tile->to_be_eliminated flags to FALSE.
 */
initialize_elimination_flags(geodesic_list, *num_good_geodesics);

/*
 * Look at each element on the geodesic list in turn.
 *
 * If it's to_be_eliminated, skip it.
 *
 * Otherwise, compute all its conjugates which pass within a distance
 * R of the basepoint, where R is the spine_radius (cf. top-of-file
 * documentation). Compare each conjugate to all other elements on
 * the geodesic_list which have the same complex length (recall that
 * the geodesic_list is sorted by complex length, so the potential
 * conjugates will all be nearby), and if any matches are found, mark
 * them to_be_eliminated. Do the same to eliminate the element's
 * inverse and all its conjugates.
 *
 * While we're at it, we'll also check whether each element is
 * conjugate to its own inverse, and thereby determine its topology.
 */

for (i = 0; i < *num_good_geodesics; i++)
    if (geodesic_list[i]->to_be_eliminated == FALSE)
        eliminate_its_conjugates(    geodesic_list,
                                      *num_good_geodesics,
                                      i,
                                      conjugator_list,
                                      num_conjugators,
                                      spine_radius);

/*
 * Compress the geodesic_list by eliminating pointers to tiles which
 * are to_be_eliminated. The remaining good pointers move towards
 * the beginning of the list, overwriting the eliminated pointers.
 */
compress_geodesic_list(geodesic_list, num_good_geodesics);

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

static void make_conjugator_list(
    Tile    ***conjugator_list,
    int     *num_conjugators,
    Tile    *tiling,
    int     num_translates)
{
    /*
     * Allocate space for the conjugator_list.
     */
    *conjugator_list = NEW_ARRAY(num_translates, Tile *);

    /*
     * Initialize the count to zero.
     */
    *num_conjugators = 0;

    /*
     * Recursively add pointers to all tiling elements to the list.
     */

```

```

    add_conjugators_to_list(tiling, *conjugator_list, num_conjugators);

    /*
     * Do a quick error check.
     */
    if (*num_conjugators != num_translates)
        uFatalError("make_conjugator_list", "length_spectrum");

    /*
     * Sort the list by order of increasing basepoint translation
     * distance. The basepoint translation distance is acosh(tile->g[0][0])
     * and acosh() is monotonic, so we may sort directly on tile->g[0][0].
     */
    qsort(*conjugator_list, *num_conjugators, sizeof(Tile *),
        compare_translation_distances);
}

```

```

static void add_conjugators_to_list(
    Tile      *root,
    Tile      **conjugator_list,
    int       *num_conjugators)
{
    Tile      *subtree_stack,
              *subtree;

    /*
     * Implement the recursive counting 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 = root;
    if (root != NULL)
        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_child != NULL)
        {
            subtree->left_child->next_subtree = subtree_stack;
            subtree_stack = subtree->left_child;
        }
        if (subtree->right_child != NULL)
        {
            subtree->right_child->next_subtree = subtree_stack;
            subtree_stack = subtree->right_child;
        }

        /*
         * Add the subtree's root node.
         */
        conjugator_list[(*num_conjugators)++] = subtree;
    }
}

```

```

static int CDECL compare_translation_distances(
    const void *tile0,
    const void *tile1)
{
    double diff;

    diff = (*(Tile **)tile0)->g[0][0] - (*(Tile **)tile1)->g[0][0];

    if (diff < 0.0)
        return -1;

    if (diff > 0.0)
        return +1;

    return 0;
}

static void initialize_elimination_flags(
    Tile **geodesic_list,
    int num_good_geodesics)
{
    int i;

    for (i = 0; i < num_good_geodesics; i++)
        geodesic_list[i]->to_be_eliminated = FALSE;
}

static void eliminate_its_conjugates(
    Tile **geodesic_list,
    int num_good_geodesics,
    int i0, /* index of geodesic under consideration */
    Tile **conjugator_list,
    int num_conjugators,
    double spine_radius)
{
    double conjugator_cutoff;
    Tile the_conjugate,
        the_inverse,
        the_inverse_conjugate;
    int i,
        j;

    /*
     * We want to find all conjugates of geodesic_list[i0] or its inverse
     * which pass within a distance R of the basepoint. According to
     * Lemma 3' at the top of this file, it suffices to consider conjugators
     * which move the basepoint a distance at most 2 acosh(cosh R cosh L/4),
     * where R is the spine_radius and L is the length of geodesic_list[i0].
     *
     * While we're at it, we'll also check whether geodesic_list[i0] is
     * conjugate to its own inverse, and thereby determine its topology.
     */

    /*
     * The identity appears on the conjugator_list, so the inverse
     * of geodesic_list[i0] will be marked to_be_eliminated, along
     * with all its conjugates.
     */

    /*
     * Set up the inverse.
     */
    o3l_invert(geodesic_list[i0]->g, the_inverse.g);
    the_inverse.length = geodesic_list[i0]->length;
    the_inverse.parity = geodesic_list[i0]->parity;

    /*
     * We'll consider conjugators whose [0][0] entry is at most
     * cosh( 2 acosh(cosh R cosh L/4) ).
     */
    conjugator_cutoff = cosh( 2 * arccosh(

```

```

    cosh(spine_radius) * cosh(geodesic_list[i0]->length.real/4)) )
    + CONJUGATOR_EPSILON;

/*
 * While we're at it, we might as well check whether geodesic_list[i0]
 * is conjugate to its own inverse. If so, it will be topologically
 * a mirrored interval. If not, it will be topologically a circle.
 * We assume it's a circle until we discover otherwise.
 */
geodesic_list[i0]->topology = orbifold_s1;

/*
 * Fortunately the conjugator_list is sorted by basepoint translation
 * distance (i.e. by tile->g[0][0]), so we can start at the beginning
 * of the list and go until tile->g[0][0] exceeds conjugator_cutoff.
 */
for ( j = 0;
      j < num_conjugators
      && conjugator_list[j]->g[0][0] <= conjugator_cutoff;
      j++)
{
    /*
     * Conjugate geodesic_list[i0] by conjugator_list[j]
     * to obtain the_conjugate.
     */
    o3l_conjugate( geodesic_list[i0]->g,
                  conjugator_list[j]->g,
                  the_conjugate.g);
    the_conjugate.length = geodesic_list[i0]->length;
    the_conjugate.parity = geodesic_list[i0]->parity;

    /*
     * Conjugate the_inverse by conjugator_list[j]
     * to obtain the_inverse_conjugate.
     */
    o3l_conjugate( the_inverse.g,
                  conjugator_list[j]->g,
                  the_inverse_conjugate.g);
    the_inverse_conjugate.length = the_inverse.length;
    the_inverse_conjugate.parity = the_inverse.parity;

    /*
     * Does the_conjugate equal the_inverse?
     */
    if (o3l_equal(the_conjugate.g, the_inverse.g, ISOMETRY_EPSILON) == TRUE)
        geodesic_list[i0]->topology = orbifold_mI;

    /*
     * If the_conjugate's axis doesn't come within a distance R of
     * the basepoint, then it can't possibly be on the geodesic_list.
     *
     * length_spectrum() has already added CONJUGATE_SPINE_EPSILON
     * to spine_radius. We want to err on the side of considering
     * too many possible conjugates rather than too few.
     */
    if (distance_to_origin(&the_conjugate) > spine_radius)
        continue;

    /*
     * Compare the_conjugate and the_inverse_conjugate to each
     * geodesic_list[i] which has the same real length as
     * geodesic_list[i0], up to roundoff error. (For an
     * orientation-preserving geodesic in a nonorientable
     * manifold, the_conjugate and geodesic_list[i0] might have
     * opposite torsions.)
     */
    for ( i = i0 + 1; i < num_good_geodesics; i++)
    {
        /*
         * If geodesic_list[i] is already marked for elimination
         * we can skip it.
         */
        if (geodesic_list[i]->to_be_eliminated == TRUE)
            continue;
    }
}

```

```

        /*
        * As soon as geodesic_list[i] has a length which differs from
        * the length of geodesic_list[i0] by more than roundoff error
        * we can break from the i loop (recall that the geodesic_list
        * is sorted by complex length).
        */
        if ( geodesic_list[i] ->length.real
            - geodesic_list[i0] ->length.real
            > POSSIBLE_CONJUGATE_EPSILON
            )
            break;

        /*
        * Does geodesic_list[i]->g equal the_conjugate.g or
        * the_inverse_conjugate.g ?
        */
        if (o3l_equal(geodesic_list[i]->g, the_conjugate.g, ISOMETRY_EPSILON)
            || o3l_equal(geodesic_list[i]->g, the_inverse_conjugate.g, ISOMETRY_EPSILON))
        /*
        * Set geodesic_list[i]->to_be_eliminated to TRUE, because
        * geodesic_list[i]->g is conjugate to geodesic_list[i0]->g.
        */
        geodesic_list[i]->to_be_eliminated = TRUE;
    }
}

static void compress_geodesic_list(
    Tile **geodesic_list,
    int *num_good_geodesics)
{
    int n,
        i;

    /*
    * The variable 'n' keeps track of how many (Tile *) pointers
    * have been kept.
    */

    n = 0;

    /*
    * Copy pointers we want to keep into a contiguous block at the
    * beginning of the geodesic_list.
    */

    for (i = 0; i < *num_good_geodesics; i++)
        if (geodesic_list[i]->to_be_eliminated == FALSE)
            geodesic_list[n++] = geodesic_list[i];

    /*
    * Update *num_good_geodesics.
    */

    *num_good_geodesics = n;
}

static Boolean is_manifold_orientable(
    WEPolyhedron *polyhedron)
{
    WEFace *face;

    for (face = polyhedron->face_list_begin.next;
        face != &polyhedron->face_list_end;
        face = face->next)

        if (gl4R_determinant(*face->group_element) < 0.0)

            return FALSE;
}

```

```

    return TRUE;
}

static void copy_lengths(
    Tile      **geodesic_list,
    int       num_good_geodesics,
    MultiLength **spectrum,
    int       *num_lengths,
    Boolean    multiplicities,
    Boolean    manifold_is_orientable)
{
    int       i,
             j;
    MultiLength *multilength_array;

    /*
     * The case num_good_geodesics == 0 is handled separately because
     * we don't want to allocate an array of zero length.
     */
    if (num_good_geodesics == 0)
    {
        *spectrum      = NULL;
        *num_lengths    = 0;
        return;
    }

    /*
     * First allocate an array that's sure to be long enough.
     * Once we've found all the MultiLengths we'll copy them into an
     * array of precisely the right size.
     */
    multilength_array = NEW_ARRAY(num_good_geodesics, MultiLength);

    /*
     * Initialize *num_lengths to zero.
     */
    *num_lengths = 0;

    /*
     * By the way, if multiplicities == TRUE, then the topologies will
     * be set to orbifold_sl or orbifold_mI. Otherwise they'll all be
     * set to orbifold_unknown. Either way, the following code does
     * what it should.
     */

    /*
     * Each Tile on the geodesic_list either defines a new MultiLength
     * or increases the multiplicity of an old one.
     */
    for (i = 0; i < num_good_geodesics; i++)
    {
        /*
         * Compare geodesic_list[i] to all multilengths of the same
         * real length (recall that the geodesic_list is sorted by
         * length, so the list of MultiLengths will be too).
         */
        for (j = *num_lengths - 1; TRUE; --j)
        {
            /*
             * If we either exhaust the multilength_array or reach
             * an element whose real length is less than that of
             * geodesic_list[i], then we know that geodesic_list[i]
             * defines a new MultiLength.
             */
            if
            (
                j < 0
                ||
                geodesic_list[i]->length.real
                - multilength_array[j].length.real
                > DUPLICATE_LENGTH_EPSILON
            )

```



```

    {
        multilength_array[*num_lengths].length      = geodesic_list[i]->length;
        multilength_array[*num_lengths].parity      = geodesic_list[i]->parity;
        multilength_array[*num_lengths].topology    = geodesic_list[i]->
topology;
        multilength_array[*num_lengths].multiplicity = 1;

        /*
         * If the manifold or orbifold is nonorientable, the sign
         * of the torsion is arbitrary, so report it as positive.
         */
        if (manifold_is_orientable == FALSE)
            multilength_array[*num_lengths].length.imag =
                fabs(multilength_array[*num_lengths].length.imag);

        (*num_lengths)++;

        break;
    }

    /*
     * If geodesic_list[i] has the same torsion, parity and
     * topology as multilength_array[j], then we increment the
     * multiplicity of multilength_array[j]. (The above test
     * insures that the lengths are equal up to roundoff error.)
     */
    if
    (
        geodesic_list[i]->parity == multilength_array[j].parity
    &&
        geodesic_list[i]->topology == multilength_array[j].topology
    &&
        fabs(
            (
                manifold_is_orientable ?
                geodesic_list[i]->length.imag :
                fabs(geodesic_list[i]->length.imag)
            )
            - multilength_array[j].length.imag
        )
        < DUPLICATE_LENGTH_EPSILON
    )
    {
        multilength_array[j].multiplicity++;
        break;
    }
}

/*
 * If multiplicities is FALSE, report all multiplicities as zero.
 */
if (multiplicities == FALSE)
    for (j = 0; j < *num_lengths; j++)
        multilength_array[j].multiplicity = 0;

/*
 * Allocate the array of MultiLengths which we'll pass to the UI.
 */
*spectrum = NEW_ARRAY(*num_lengths, MultiLength);

/*
 * Copy in the data.
 */
for (j = 0; j < *num_lengths; j++)
    (*spectrum)[j] = multilength_array[j];

/*
 * Free the temporary array.
 */
my_free(multilength_array);
}

```

```
static void free_tiling(
    Tile      *root)
{
    Tile      *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 = root;
    if (root != NULL)
        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_child != NULL)
        {
            subtree->left_child->next_subtree = subtree_stack;
            subtree_stack = subtree->left_child;
        }
        if (subtree->right_child != NULL)
        {
            subtree->right_child->next_subtree = subtree_stack;
            subtree_stack = subtree->right_child;
        }

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