

```

/*
 * dual_curves.c
 *
 * This file provides the functions
 *
 * void dual_curves(    Triangulation      *manifold,
 *                      int                max_size,
 *                      int                num_curves,
 *                      DualOneSkeletonCurve ***the_curves);
 *
 * void get_dual_curve_info(
 *                      DualOneSkeletonCurve *the_curve,
 *                      Complex              *complete_length,
 *                      Complex              *filled_length,
 *                      MatrixParity        *parity)
 *
 * void free_dual_curves(
 *                      int                num_curves,
 *                      DualOneSkeletonCurve **the_curves);
 *
 * dual_curves() computes a reasonable selection of simple closed curves
 * in a manifold's dual 1-skeleton.  The meaning of "reasonable selection"
 * will be clarified below in the description of the algorithm.
 *
 * Input arguments:
 *
 * manifold      is a pointer to the Triangulation of interest.
 *
 * max_size      is the maximum number of segments in the curves
 *                to be considered.  Six is a reasonable value.
 *
 * Output arguments:
 *
 * *num_curves   will be set to the number of curves the function finds.
 *
 * *the_curves   will be set to the address of an array containing
 *                pointers to the DualOneSkeletonCurves.  That is,
 *                (*the_curves)[i] will be a pointer to the i-th curve.
 *                If no nonparabolic curves are found (as happens
 *                with the Gieseking), *the_curves will be set to NULL.
 *
 * get_dual_curve_info() reports the complex length of a curve
 * in the dual 1-skeleton, relative to both the complete and filled
 * hyperbolic structures, and also its parity (orientation_preserving
 * or orientation_reversing).
 *
 * free_dual_curves() releases the array of DualOneSkeletonCurves
 * allocated by dual_curves().  (It releases both the
 * DualOneSkeletonCurves and the array of pointers to them.)
 *
 * Terminology: Throughout this file we will flip-flop freely between
 * the description of a curve as vertices and edges in the dual 1-skeleton
 * and its dual description as Tetrahedra and 2-cells in the original
 * Triangulation.  Please don't let this confuse you.
 *
 * The algorithm.
 *
 * The set of all simple closed curves in the dual 1-skeleton
 * divides naturally into homotopy classes.  Ideally, we'd like to
 * compute precisely one representative of each homotopy class,
 * and we'd like that representative to be unknotted in the sense
 * that it's isotopic to the unique geodesic in its homotopy class.
 * We won't always achieve this goal, but we'll do the best we can.
 *
 * The main obstacle to achieving the goal is the vast number of
 * simple closed curves to be considered, and the large number of
 * curves within each homotopy class.  Even for a given curve of
 * size n, we could start traversing it at any of its n vertices,
 * and in either of two directions.  To avoid this last problem, we
 * number all the Tetrahedra in the Triangulation and make two conventions:
 *
 * Convention #1: Each curve will have a "base Tetrahedron" which

```

```

* is the lowest-numbered Tetrahedron on the curve.
*
* Convention #2: Each curve is traversed by starting at its
* base Tetrahedron and going in the direction which takes you
* through the face of lower index (of the two faces of the base
* Tetrahedron which intersect the curve). For example, if a
* curve intersects faces 1 and 3 of its base Tetrahedron, then
* the canonical direction to traverse it is to start off through
* face 1, traverse the whole curve, and return through face 3.
*
* It's easy enough to detect whether two different curves are
* homotopic in the universal cover (they'll have the same Moebius
* transformation) but it's not so easy to detect when one is homotopic
* to a translate of the other. For this reason we keep only one curve
* for any given complex length. An unfortunate side effect of this
* decision is that when a manifold contains two geodesics of the
* same length, we'll be able to drill out only one of them (in most
* cases geodesics of the same length will be equivalent under some
* symmetry of the manifold, but nevertheless it would have been nice
* not to have had to impose this restriction). For each complex length,
* the curve we keep will have minimal combinatorial size, to minimize
* the chance of choosing a knotted representative of the homotopy class.
*
* [Modified 93/9/14 by JRW to compare the complex lengths of curves
* in the filled structure as well as in the complete structure.
* A curve will be discarded only if it has the same complex length
* as some other curve, relative to both the complete and the filled
* hyperbolic structures.]
*
* Within this file we are interested in the complex lengths of
* geodesics relative to the complete structure on the manifold,
* because curves which are parabolics relative to the complete
* structure will either be obviously parallel to the boundary
* (in which case drill_cusp() will fail), or not-so-obviously
* parallel to the boundary, in which case drill_cusp() will yield
* a nonhyperbolic manifold. But we also compute the complex
* lengths of geodesics relative to the filled structure, for
* the convenience of the user (e.g. the user might want to drill
* out a geodesic of minimal length in a certain closed manifold,
* perhaps as part of an effort to prove that two closed manifolds
* are isometric [symmetry_group_closed.c now does this automatically]).
*/

/*
* 95/10/1 JRW
* I was concerned about stack/heap collisions caused by recursive
* functions, and was going through the SnapPea kernel revising
* recursive algorithms to use heap space instead. However, I decided
* not to revise consider_its_neighbor() for the following reasons.
*
* (1) It's fairly complicated. Revising it would be time-consuming
* and error-prone.
*
* (2) The recursion is very "shallow". It never goes more than
* max_size levels deep. Typically max_size is about 6, or at
* most 8 or 10, because the number of curves grows exponentially
* with max_size. In other words, this is a harmless recursion.
*/

#include "kernel.h"

/*
* BIG_MODULUS is used to recognize when a complex number
* represents the point at infinity on the Riemann sphere.
* I haven't given much thought to the best value for
* BIG_MODULUS, because in all common cases a number
* will either clearly be infinite or clearly not be
* infinite.
*
* 93/10/9. An error message provided the occasion to think
* about the best value for BIG_MODULUS. Amazingly enough, after
* doing several dozen Dehn fillings on each of thousands of cusped
* census manifolds, I got the
*/

```

```

*      uFatalError("verify_mt_action", "dual_curves");
*
*  which indicated the previous value of BIG_MODULUS (namely 1e10)
*  was not big enough ["too big"? 2002/11/05].
*  The offending (revealing?) example is v2395(-1,1).
*  The values of fz and w are
*
*      fz = 1.2656500338200879e+8 + i 1.89154960743708773e+9
*      w  = 1.2656500287902592e+8 + i 1.89154960735500588e+9
*
*  Later another example occurred with
*
*      fz = -6.19138762819279958e+7 + i -6.56272444717916559e+7
*      w  = -6.19138762819635613e+7 + i -6.5627244471300831e+7
*
*  For most purposes I think I'll leave BIG_MODULUS at 1e10
*  (this ensures accurate computations), but for the error check
*  in verify_mt_action() I'll make a provision that if fz and w
*  have moduli in the range 1e5 - 1e10 and differ by a small
*  percentage, they should be considered equal.
*/

#define BIG_MODULUS      1e10
#define BIG_MODULUS1    1e5
#define FRACTIONAL_DIFF  1e-4 /* 2002/11/05 corrected to 1e-4; had been 1e4 */

/*
*  MoebiusTransformations which translate a distance less
*  than PARABOLIC_EPSILON are judged to be parabolics. I haven't
*  given a lot of thought to the best value for PARABOLIC_EPSILON,
*  because in practice I expect that in all common examples
*  the MoebiusTransformations will be clearly parabolic or
*  clearly not parabolic.
*/

#define PARABOLIC_EPSILON 1e-2

/*
*  Two complex lengths are considered equal iff their real
*  and imaginary parts are within LENGTH_EPSILON of each other.
*/

#define LENGTH_EPSILON    1e-5

/*
*  If the MoebiusTransformation's action on the base
*  Tetrahedron's fourth vertex isn't correct to within
*  MINIMAL_ACCURACY, a fatal error is generated.
*/

#define MINIMAL_ACCURACY 1e-6

static void      initialize_flags(Triangulation *manifold);
static void      consider_its_neighbor(Tetrahedron *tet, FaceIndex face, int
size, Complex corners[2][4], Orientation orientation, Tetrahedron *tet0, FaceIndex
face0, int max_size, Triangulation *manifold, DualOneSkeletonCurve **curve_tree);
static void      compute_corners(Complex corners[4], Complex nbr_corners[4],
FaceIndex face, FaceIndex entry_face, Permutation gluing, Orientation nbr_orientation,
ComplexWithLog cwl[3]);
static void      compute_Moebius_transformation(Orientation orientation, Complex
corners[4], MoebiusTransformation *mt);
static void      verify_mt_action(MoebiusTransformation *mt, Complex z, Complex
w);
static void      add_curve_to_tree(Triangulation *manifold, DualOneSkeletonCurve
**curve_tree, MatrixParity parity, Complex cl[2], int size);
static DualOneSkeletonCurve *package_up_the_curve(Triangulation *manifold, MatrixParity
parity, Complex cl[2], int size);
static void      replace_contents_of_node(DualOneSkeletonCurve *node,
Triangulation *manifold, MatrixParity parity, Complex cl[2], int size);
static void      convert_tree_to_pointer_array( DualOneSkeletonCurve *curve_tree
, int *num_curves, DualOneSkeletonCurve ***the_curves);
static int       count_the_curves(DualOneSkeletonCurve *curve_tree);
static void      write_node_addresses(DualOneSkeletonCurve *curve_tree,

```

```

    DualOneSkeletonCurve **the_array, int *count);

void dual_curves(
    Triangulation      *manifold,
    int                max_size,
    int                *num_curves,
    DualOneSkeletonCurve ***the_curves)
{
    Tetrahedron        *tet0;
    FaceIndex          face0;
    Complex             corners0[2][4];
    DualOneSkeletonCurve *curve_tree;
    int                i;

    /*
     * If the manifold does not have a hyperbolic
     * structure, return no curves.
     */

    if
    (
        (
            manifold->solution_type[complete] != geometric_solution
            && manifold->solution_type[complete] != nongeometric_solution
        )
        ||
        (
            manifold->solution_type[filled] != geometric_solution
            && manifold->solution_type[filled] != nongeometric_solution
            && manifold->solution_type[filled] != flat_solution
        )
    )
    {
        *num_curves = 0;
        *the_curves = NULL;
        return;
    }

    /*
     * Make sure the Tetrahedra are numbered.
     */

    number_the_tetrahedra(manifold);

    /*
     * curve_tree is a pointer to the root of a binary
     * tree containing all the curves found so far.
     * Initialize it to NULL.
     */

    curve_tree = NULL;

    /*
     * Set the tet_on_curve and face_on_curve[] flags
     * to FALSE to show that the curve is initially empty.
     */

    initialize_flags(manifold);

    /*
     * Consider each possible base Tetrahedron.
     */

    for (tet0 = manifold->tet_list_begin.next;
         tet0 != &manifold->tet_list_end;
         tet0 = tet0->next)
    {
        /*
         * Mark the base Tetrahedron.
         */
        tet0->tet_on_curve = TRUE;

        /*

```

```

    * Put the corners of the base Tetrahedron
    * in the standard position.
    */
for (i = 0; i < 2; i++)      /* i = complete, filled */
{
    corners0[i][0] = Infinity;
    corners0[i][1] = Zero;
    corners0[i][2] = One;
    corners0[i][3] = tet0->shape[i]->cwl[ultimate][0].rect;
}

/*
 * Consider each possible initial face for the curve.
 * By Convention #2 above, we may assume the initial
 * face is not face 3.
 */
for (face0 = 0; face0 < 3; face0++)
    consider_its_neighbor( tet0, face0, 1,
                           corners0, right_handed,
                           tet0, face0, max_size,
                           manifold, &curve_tree);

/*
 * Unmark the base Tetrahedron.
 */
tet0->tet_on_curve = FALSE;
}

/*
 * curve_tree will now point to a binary tree containing
 * the DualOneSkeletonCurves. Write the addresses of the
 * nodes into an array, as specified in the documentation
 * at the top of this file.
 */

convert_tree_to_pointer_array(curve_tree, num_curves, the_curves);
}

static void initialize_flags(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i;

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

        for (i = 0; i < 4; i++)
            tet->face_on_curve[i] = FALSE;
    }
}

static void consider_its_neighbor(
    Tetrahedron *tet,
    FaceIndex face,
    int size,
    Complex corners[2][4],
    Orientation orientation,
    Tetrahedron *tet0,
    FaceIndex face0,
    int max_size,
    Triangulation *manifold,
    DualOneSkeletonCurve **curve_tree)
{
    Tetrahedron *nbr;
    Permutation gluing;
    FaceIndex nbr_face,
              entry_face;
    Orientation nbr_orientation;

```

```

Complex          nbr_corners[2][4];
MoebiusTransformation mt[2];
Complex          cl[2];
int              i;

/*
 * We want to examine the Tetrahedron incident
 * to face "face" of Tetrahedron "tet".
 */
nbr          = tet->neighbor[face];
gluing       = tet->gluing[face];
entry_face   = EVALUATE(gluing, face);
nbr_orientation = (parity[gluing] == orientation_preserving) ?
                  orientation :
                  ! orientation;

/*
 * Is nbr the base Tetrahedron?
 * If so, process the curve and return.
 */
if (nbr == tet0)
{
    /*
     * We've found a curve adhering to Convention #2
     * iff we're reentering the base Tetrahedron at
     * a higher numbered face than the one we left at.
     */
    if (entry_face > face0)
    {
        /*
         * Process this curve.
         */

        /*
         * Compute the locations of the nbr_corners.
         */
        for (i = 0; i < 2; i++) /* i = complete, filled */
            compute_corners(corners[i], nbr_corners[i], face, entry_face,
                           gluing, nbr_orientation,
                           nbr->shape[i]->cwl[ultimate]);

        /*
         * For the complete structure and also for the filled structure,
         * compute the MoebiusTransformation which takes the original
         * base Tetrahedron to nbr.
         */
        for (i = 0; i < 2; i++) /* i = complete, filled */
            compute_Moebius_transformation(
                nbr_orientation, nbr_corners[i], &mt[i]);

        /*
         * The computation of the MoebiusTransformation used
         * only the location of corners 0, 1 and 2. As a check
         * against errors, let's see whether the MoebiusTransformation
         * also takes corner 3 to the right place.
         */
        for (i = 0; i < 2; i++) /* i = complete, filled */
            verify_mt_action(
                &mt[i],
                tet0->shape[i]->cwl[ultimate][0].rect,
                nbr_corners[i][3]);

        /*
         * Compute the complex length of the geodesic corresponding
         * to the covering transformation represented by mt.
         */
        for (i = 0; i < 2; i++) /* i = complete, filled */
            cl[i] = complex_length_mt(&mt[i]);

        /*
         * Ignore parabolics (relative to the complete structure).
         */
        if (fabs(cl[complete].real) < PARABOLIC_EPSILON)
            return;
    }
}

```

```

    /*
     * Add the final segment to close the curve.
     */
    tet->face_on_curve[face]      = TRUE;
    nbr->face_on_curve[entry_face] = TRUE;

    /*
     * Add the curve to the list, unless a curve of
     * equal complex length and smaller or equal
     * combinatorial size is already there.
     */
    add_curve_to_tree(manifold, curve_tree, mt[0].parity, cl, size);

    /*
     * Remove the final segment of the curve before
     * continuing on to look for other possibilities.
     */
    tet->face_on_curve[face]      = FALSE;
    nbr->face_on_curve[entry_face] = FALSE;
}

return;
}

/*
 * Is nbr a Tetrahedron which has already been visited
 * (other than the base Tetrahedron, which was handled above)?
 * If so, return.
 */
if (nbr->tet_on_curve == TRUE)
    return;

/*
 * If nbr's index is less than the index of the base Tetrahedron,
 * then Convention #1 dictates that we return without doing anything.
 */
if (nbr->index < tet0->index)
    return;

/*
 * If size == max_size, then we should stop the recursion.
 */
if (size == max_size)
    return;

/*
 * nbr has passed all the above tests, so add it to the curve...
 */
nbr->tet_on_curve = TRUE;
tet->face_on_curve[face]      = TRUE;
nbr->face_on_curve[entry_face] = TRUE;

/*
 * ...compute the positions of its corners...
 */
for (i = 0; i < 2; i++) /* i = complete, filled */
    compute_corners(corners[i], nbr_corners[i], face, entry_face,
                    gluing, nbr_orientation,
                    nbr->shape[i]->cwl[ultimate]);

/*
 * ...recursively consider each of its neighbors...
 */
for (nbr_face = 0; nbr_face < 4; nbr_face++)
    if (nbr_face != entry_face)
        consider_its_neighbor(
            nbr, nbr_face, size + 1,
            nbr_corners, nbr_orientation,
            tet0, face0, max_size,
            manifold, curve_tree);

/*
 * ...and remove it from the curve.

```

```

    */
    nbr->tet_on_curve = FALSE;
    tet->face_on_curve[face] = FALSE;
    nbr->face_on_curve[entry_face] = FALSE;
}

static void compute_corners(
    Complex      corners[4],
    Complex      nbr_corners[4],
    FaceIndex    face,
    FaceIndex    entry_face,
    Permutation  gluing,
    Orientation  nbr_orientation,
    ComplexWithLog cwl[3])
{
    int i;

    /*
     * Knock off the three easy ones.
     */
    for (i = 0; i < 4; i++)
        if (i != face)
            nbr_corners[EVALUATE(gluing, i)] = corners[i];

    /*
     * Then call compute_fourth_corner() to find the
     * fourth corner in terms of the first three.
     */
    compute_fourth_corner( nbr_corners,
                          entry_face,
                          nbr_orientation,
                          cwl);
}

static void compute_Moebius_transformation(
    Orientation  orientation,
    Complex      corners[4],
    MoebiusTransformation *mt)
{
    /*
     * The base Tetrahedron originally had its corners
     * at (infinity, 0, 1, z). We've now traced out a
     * curve which, when lifted to the universal cover,
     * leads to a translate of the base Tetrahedron with
     * corner coordinates given by the array corners[].
     * The associated covering transformation will be
     * the Moebius transformation which takes
     *
     *      infinity -> corners[0]
     *      0         -> corners[1]
     *      1         -> corners[2]
     *
     * and has the specified Orientation.
     *
     * Because {infinity, 0, 1} are invariant under complex
     * conjugation, we can compute the SL2CMatrix without
     * worrying about the Orientation. (The Orientation
     * will, of course, determine the parity field of the
     * MoebiusTransformation.)
     *
     * An SL2CMatrix taking
     *
     *      infinity -> c0
     *      0         -> c1
     *      1         -> c2
     *
     * is given by
     *
     *      c0(c2 - c1) z + c1(c0 - c2)
     * w = -----
     *      (c2 - c1) z + (c0 - c2)
     */
}

```



```

* (This matrix must, of course, be normalized to have
* determinant one.)
*
* In the special case that c0 is infinite, the formula
* reduces to
*
*      w = (c2 - c1) z + c1
*
* In the special case that c1 is infinite, the formula
* reduces to
*
*      c0 z + (c2 - c0)
*      w = -----
*             z
*
* In the special case that c2 is infinite, the formula
* reduces to
*
*      c0 z - c1
*      w = -----
*           z - 1
*
*/

/*
* Evaluate the appropriate formula from above.
* Don't worry yet about normalizing to have
* determinant one.
*/

if (complex_modulus(corners[0]) > BIG_MODULUS)
{
    /*
    * c0 is infinite.
    * Use the special formula
    *
    *      w = (c2 - c1) z + c1
    */
    mt->matrix[0][0] = complex_minus(corners[2], corners[1]);
    mt->matrix[0][1] = corners[1];
    mt->matrix[1][0] = Zero;
    mt->matrix[1][1] = One;
}
else if (complex_modulus(corners[1]) > BIG_MODULUS)
{
    /*
    * c1 is infinite.
    * Use the special formula
    *
    *      c0 z + (c2 - c0)
    *      w = -----
    *             z
    */
    mt->matrix[0][0] = corners[0];
    mt->matrix[0][1] = complex_minus(corners[2], corners[0]);
    mt->matrix[1][0] = One;
    mt->matrix[1][1] = Zero;
}
else if (complex_modulus(corners[2]) > BIG_MODULUS)
{
    /*
    * c2 is infinite.
    * Use the special formula
    *
    *      c0 z - c1
    *      w = -----
    *           z - 1
    */
    mt->matrix[0][0] = corners[0];
    mt->matrix[0][1] = complex_negate(corners[1]);
    mt->matrix[1][0] = One;
    mt->matrix[1][1] = MinusOne;
}
else

```

```

{
    /*
     * None of {c0, c1, c2} is infinite.
     * Use the general formula
     *
     *      c0(c2 - c1) z + c1(c0 - c2)
     *      w = -----
     *      (c2 - c1) z + (c0 - c2)
     *
     * Note that for computational efficiency we
     * evaluate the terms in the denominator first.
     */
    mt->matrix[1][0] = complex_minus(corners[2], corners[1]);
    mt->matrix[1][1] = complex_minus(corners[0], corners[2]);
    mt->matrix[0][0] = complex_mult(corners[0], mt->matrix[1][0]);
    mt->matrix[0][1] = complex_mult(corners[1], mt->matrix[1][1]);
}

/*
 * Normalize matrix to have determinant one.
 */
sl2c_normalize(mt->matrix);

/*
 * Set the MoebiusTransformation's parity.
 * The base Tetrahedron had the right_handed Orientation,
 * so the MoebiusTransformation will be orientation_preserving
 * iff the translate of the base Tetrahedron also has the
 * right_handed Orientation.
 */
mt->parity = (orientation == right_handed) ?
             orientation_preserving :
             orientation_reversing;
}

static void verify_mt_action(
    MoebiusTransformation *mt,
    Complex z,
    Complex w)
{
    Complex fz;

    /*
     * Does mt take z to w?
     */

    /*
     * If the MoebiusTransformation is orientation_reversing,
     * we must first replace z by its complex conjugate.
     */

    if (mt->parity == orientation_reversing)
        z = complex_conjugate(z);

    /* Evaluate
     *
     * f(z) = (az + b)/(cz + d)
     *
     * and compare the result to w.
     */

    fz = complex_div(
        complex_plus(
            complex_mult(mt->matrix[0][0], z),
            mt->matrix[0][1]
        ),
        complex_plus(
            complex_mult(mt->matrix[1][0], z),
            mt->matrix[1][1]
        )
    );

    /*

```

```

    * fz and w should either be very close, or
    * both should be infinite. Flag an error
    * if this is not the case.
    */

if
(
    complex_modulus(complex_minus(fz, w)) > MINIMAL_ACCURACY
    &&
    (
        complex_modulus(fz) < BIG_MODULUS
        || complex_modulus(w) < BIG_MODULUS
    )
    &&
    (
        complex_modulus(fz) < BIG_MODULUS1
        || complex_modulus(w) < BIG_MODULUS1
        || complex_modulus(complex_div(complex_minus(fz, w), fz)) > FRACTIONAL_DIFF
    )
)
    uFatalError("verify_mt_action", "dual_curves");
}

static void add_curve_to_tree(
    Triangulation      *manifold,
    DualOneSkeletonCurve **curve_tree,
    MatrixParity        parity,
    Complex             cl[2], /* complex length of geodesic */
    int                 size) /* combinatorial size of curve */
{
    DualOneSkeletonCurve *node;
    int position;

    /*
     * First check for the special case that the
     * curve_tree might be empty.
     */

    if (*curve_tree == NULL)
    {
        *curve_tree = package_up_the_curve(manifold, parity, cl, size);
        return;
    }

    /*
     * if (the tree does not yet contain a curve of
     * the given complex length)
     * add the current curve to the tree
     * else
     * if the current curve is combinatorially shorter
     * than the old curve of the same length,
     * replace it.
     */

    node = *curve_tree;
    while (TRUE)
    {
        /*
         * Set the integer "position" to -1 if we belong
         * somewhere to the left of this node, to +1 if we
         * belong to the right, and to 0 if we belong here.
         *
         * Modified 93/9/14 by JRW. If two curves have the same complex
         * length in the complete structure but different complex lengths
         * in the filled structure, then we want to list them separately.
         *
         * Modified 94/10/8 by JRW. Sort first by filled length,
         * then by complete length. The user is probably paying more
         * attention to filled lengths than complete ones.
         */

        if (cl[filled].real < node->length[filled].real - LENGTH_EPSILON)
            position = -1;

```

```

    else if (cl[filled].real > node->length[filled].real + LENGTH_EPSILON)
        position = +1;
    else if (cl[filled].imag < node->length[filled].imag - LENGTH_EPSILON)
        position = -1;
    else if (cl[filled].imag > node->length[filled].imag + LENGTH_EPSILON)
        position = +1;

    else if (cl[complete].real < node->length[complete].real - LENGTH_EPSILON)
        position = -1;
    else if (cl[complete].real > node->length[complete].real + LENGTH_EPSILON)
        position = +1;
    else if (cl[complete].imag < node->length[complete].imag - LENGTH_EPSILON)
        position = -1;
    else if (cl[complete].imag > node->length[complete].imag + LENGTH_EPSILON)
        position = +1;

    else
        position = 0;

    switch (position)
    {
        case -1:
            if (node->left_child != NULL)
                node = node->left_child;
            else
            {
                node->left_child = package_up_the_curve(manifold, parity, cl, size);
                return;
            }
            break;

        case +1:
            if (node->right_child != NULL)
                node = node->right_child;
            else
            {
                node->right_child = package_up_the_curve(manifold, parity, cl, size);
                return;
            }
            break;

        case 0:
            if (size < node->size)
                replace_contents_of_node(node, manifold, parity, cl, size);
            return;
    }
}

/*
 * The program never reaches this point.
 */

static DualOneSkeletonCurve *package_up_the_curve(
    Triangulation *manifold,
    MatrixParity parity,
    Complex cl[2],
    int size)
{
    DualOneSkeletonCurve *node;

    node = NEW_STRUCT(DualOneSkeletonCurve);
    node->tet_intersection = NEW_ARRAY(manifold->num_tetrahedra, DualOneSkeletonCurvePiece)
    ;

    replace_contents_of_node(node, manifold, parity, cl, size);

    node->left_child = NULL;
    node->right_child = NULL;

    return node;
}

```

```

static void replace_contents_of_node(
    DualOneSkeletonCurve *node,
    Triangulation *manifold,
    MatrixParity parity,
    Complex cl[2],
    int size)
{
    Tetrahedron *tet;
    int i;

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

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

            node->tet_intersection[tet->index][i] = tet->face_on_curve[i];

    node->parity = parity;
    node->length[complete] = cl[complete];
    node->length[filled] = cl[filled];
    node->size = size;
}

```

```

static void convert_tree_to_pointer_array(
    DualOneSkeletonCurve *curve_tree,
    int *num_curves,
    DualOneSkeletonCurve ***the_curves)
{
    int count;

    /*
     * First handle the special case that no curves were found.
     */
    if (curve_tree == NULL)
    {
        *num_curves = 0;
        *the_curves = NULL;
        return;
    }

    /*
     * Count the curves.
     */
    *num_curves = count_the_curves(curve_tree);

    /*
     * Allocate the array for the pointers.
     */
    *the_curves = NEW_ARRAY(*num_curves, DualOneSkeletonCurve *);

    /*
     * Write the addresses of the nodes into the array.
     */
    count = 0;
    write_node_addresses(curve_tree, *the_curves, &count);

    /*
     * A quick error check.
     */
    if (count != *num_curves)
        uFatalError("convert_tree_to_pointer_array", "dual_curves");
}

```

```

static int count_the_curves(
    DualOneSkeletonCurve *curve_tree)
{
    DualOneSkeletonCurve *subtree_stack,
    *subtree;
    int num_curves;

```

```

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

/*
 * Initialize the count to zero.
 */
num_curves = 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_curves++;
}

return num_curves;
}

static void write_node_addresses(
    DualOneSkeletonCurve *curve_tree,
    DualOneSkeletonCurve **the_array,
    int *count)
{
    DualOneSkeletonCurve *subtree_stack,
                        *subtree;

    /*
     * Implement the recursive 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 product_tree.
     */
    subtree_stack = curve_tree;
    if (curve_tree != NULL)
        curve_tree->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 it has no further subtrees, append it to the array.
     *
     * Otherwise break it into three chunks:
     *
     *     the left subtree
     *     this node
     *     the right subtree
     *
     * and push them onto the stack in reverse order, so that they'll
     * come off in the correct order. Set this node's left_child and
     * right_child fields to NULL, so the next time it comes off the
     * stack we'll know the subtrees have been accounted for.
     */
    if (subtree->left_child == NULL && subtree->right_child == NULL)
    {
        the_array[( *count )++] = subtree;
    }
    else
    {
        /*
         * Push the right subtree (if any) onto the stack.
         */
        if (subtree->right_child != NULL)
        {
            subtree->right_child->next_subtree = subtree_stack;
            subtree_stack = subtree->right_child;
            subtree->right_child = NULL;
        }

        /*
         * Push this node onto the stack.
         * (Its left_child and right_child fields will soon be NULL.)
         */
        subtree->next_subtree = subtree_stack;
        subtree_stack = subtree;

        /*
         * Push the left subtree (if any) onto the stack.
         */
        if (subtree->left_child != NULL)
        {
            subtree->left_child->next_subtree = subtree_stack;
            subtree_stack = subtree->left_child;
            subtree->left_child = NULL;
        }
    }
}

}

void get_dual_curve_info(
    DualOneSkeletonCurve    *the_curve,
    Complex                  *complete_length,
    Complex                  *filled_length,
    MatrixParity             *parity)
{
    if (complete_length != NULL)
        *complete_length = the_curve->length[complete];

    if (filled_length != NULL)
        *filled_length = the_curve->length[filled];

    if (parity != NULL)
        *parity = the_curve->parity;
}

```

```
}

void free_dual_curves(
    int          num_curves,
    DualOneSkeletonCurve **the_curves)
{
    int i;

    /*
     * If num_curves is zero, then no storage should
     * have been allocated in the first place.
     */
    if (num_curves == 0)
    {
        if (the_curves == NULL)
            return;
        else
            uFatalError("free_dual_curves", "dual_curves");
    }

    /*
     * Free each DualOneSkeletonCurve.
     */
    for (i = 0; i < num_curves; i++)
    {
        my_free(the_curves[i]->tet_intersection);
        my_free(the_curves[i]);
    }

    /*
     * Free the pointer array.
     */
    my_free(the_curves);
}
```