

```

/*
 * link_complement.c
 *
 * This file provides the function
 *
 *      Triangulation *triangulate_link_complement(
 *          KLPPProjection *aLinkProjection);
 *
 * which triangulates the complement of aLinkProjection.
 *
 * If aLinkProjection is empty returns NULL; otherwise returns
 * a pointer to the resulting Triangulation.
 */

/*
 *
 *      The Triangulation Algorithm
 *
 *
 * Introduction
 *
 * The triangulation algorithm is quite simple once you see the picture.
 * Unfortunately, there is no easy way to include a sketch in an ASCII
 * file, so you will need to sketch your own picture as I describe it.
 * First thicken the link so that each component becomes a solid torus.
 * Position it so that it lies near the equatorial 2-sphere in  $S^3$ .
 * As I describe the vertices, edges, faces and 3-cells in the
 * triangulation, try to draw them in your own picture.
 *
 *
 * Vertices
 *
 * Mark the solid tori (i.e. the link components) with
 *
 * (1) a longitude which runs along the top of each component,
 * (2) a longitude which runs along the bottom of each component,
 * (3) a meridian which encircles the understrand at each crossing,
 * (4) a meridian which encircles the overstrand at each crossing, and
 * (5) a meridian located halfway between each pair of adjacent crossings.
 *
 * These curves divide the surfaces of the solid tori into topological
 * squares which will serve as the ideal vertices of the triangulation.
 *
 * In addition, the initial triangulation will include two finite
 * (i.e. non-ideal) vertices, one at the south pole of  $S^3$  and one at
 * the north pole. (Don't worry -- we'll get rid of them at the end.)
 * You should thicken these two finite vertices to become solid balls,
 * for consistency with our picture of the link as a union of solid tori.
 *
 *
 * Edges
 *
 * The triangulation's edges are as follows.
 *
 * (1) At each crossing of the link projection, there are edges running
 *
 *      (A) from the south pole of  $S^3$  to the bottom of the understrand,
 *      (B) from the top of the understrand to the bottom of the overstrand,
 *      (C) from the top of the overstrand to the north pole of  $S^3$ .
 *
 * (2) Midway between each pair of adjacent crossings, there are edges
 *
 *      (A) from the south pole of  $S^3$  to the bottom of the link component,
 *      (B) from the top of the link component to the north pole of  $S^3$ .
 *
 * (3) In the center of each region of the link projection there is an edge
 *
 *      (A) from the south pole to the north pole.
 *
 *
 * Faces
 *
 * There are three types of faces (2-cells).
 *
 * (1) Triangular faces with edges of type (2A), (2B) and (3A).

```

```

*      In your picture each such face will look like a topological hexagon,
*      but only three of the edges are true edges of the triangulation.
*      The other three "edges" are arcs on the boundary components:
*      one arc is a half-meridian on a solid torus, another is an arc
*      along the solid 3-ball at the north pole, and the last is an arc
*      along the solid 3-ball at the south pole.
*
* (2) Triangular faces with edges of type (1A), (1B) and (2A).
*      As before, each face will look like a topological hexagon.
*      In addition to the three real edges, there are three arcs on
*      the boundary: one runs along the bottom of the overstrand,
*      a second is half a meridian encircling the understrand, and
*      the last is an arc on the 3-ball at the south pole.
*
* (3) Triangular faces with edges of type (1B), (1C) and (2B).
*      These are just like the faces described in the previous paragraph,
*      except that they are in the northern hemisphere instead of the
*      southern hemisphere.
*
* (4) Bigons with edges of type (1A) and (2A). Each looks like a
*      topological square, because it includes an arc running along
*      the bottom of an understrand, and an arc on the 3-ball at the
*      south pole.
*
* (5) Bigons with edges of type (1C) and (2B). These are just like
*      the bigons in the previous paragraph, only they connect the
*      overstrand to the north pole, rather than connecting the
*      understrand to the south pole.
*
* So . . . you should draw all the 2-cells of the above types wherever
* they make sense. (I'm hoping your understanding of what "makes sense"
* is clearer than any longwinded explanation I could provide.)
*
*
* 3-cells
*
* The 2-cells described above divide the link complement into 3-cells.
* The 3-cells are all identical to one another, and four of them surround
* each crossing. (So in particular the number of 3-cells is exactly
* four times the number of crossings.) Each 3-cell has six real faces
* (two of type (1) and one each of types (2), (3), (4) and (5) described
* in the preceding section), as well as four square pieces of the
* manifold's boundary (one on the understrand, one on the overstrand,
* one on the 3-ball at the south pole and one on the 3-ball at the north
* pole).
*
* At this point, our cell decomposition fails to be an ideal triangulation
* for two reasons:
*
* (1) The 3-cells aren't tetrahedra.
*
* (2) The vertices at the north and south pole are finite, not ideal.
*
* The following section on Shrinking Away Bigons resolves problem (1),
* and then the section on Removing Finite Vertices resolves problem (2).
*
*
* Shrinking Away Bigons
*
* We can get rid of the bigons simply by shrinking them away!
* Each is a rectangle with two real edges as well as two arcs on the
* boundary. Shrink the arcs on the boundary to zero length. This
* merges the two real edges into a single edge, and the bigon disappears.
*
*      Essential Assumption: Each link component has at least one
*      overcrossing and at least one undercrossing.
*
* The Essential Assumption guarantees that the two real edges being
* merged are always distinct. If the Essential Assumption were false,
* you'd have a circular chain of bigons. You could shrink all but one
* of the bigons, but when you went to shrink the last one you'd find
* its two real edges were actually the same edge, so you couldn't shrink
* away the bigon without changing the topology of the manifold.

```

```

* make_all_components_have_crossings() adds a nugatory crossing to each
* link component which doesn't already have both under- and overcrossings,
* so the Essential Assumption will always be satisfied.
*
* Once we've shrunk away all bigons, each 3-cell (which originally
* had six faces -- two bigons and four triangles) is left with only
* the four triangular faces. Furthermore, shrinking the bigons
* collapses each square region on the manifold's boundary (i.e. on
* the tubular neighborhood of the link and on the 3-balls at the
* north and south poles) to a triangular region. In other words,
* each 3-cell has become an ideal tetrahedron.
*
* Computational note: The computer code never explicitly deals with
* bigons. It simply sets up the correct gluings on the triangular faces,
* ignores the bigons, and everything comes out right!
*
* Removing Finite Vertices
*
* This is easy. We just call SnapPea's remove_finite_vertices() function.
* Please see the file finite_vertices.c for extensive documentation
* of the finite vertex removal algorithm.
*
* March 1997. If the link projection is obviously disconnected,
* triangulate_link_complement() now does a trivial type II Reidemeister
* move to make it connected. Similarly, if there are obvious unknotted
* components, it adds nugatory crossings. As a result, it can now
* triangulate the complements of ALL link projections.
* (The original design of storing the LCCrossings on a fixed array
* is no longer so nice now that we sometimes have to add crossings.
* Fortunately we add crossings only in rare circumstances.)
*/

#include "kernel.h"

/*
* The permutation 2310 is 10110100 = 0xB4.
*/
#define PERMUTATION2310 0xB4

/*
* We'll transfer the link projection to our own internal format, so we
* can tack on various fields for internal use without affecting the UI.
*/

typedef struct LCProjection LCProjection;
typedef struct LCCrossing LCCrossing;

struct LCProjection
{
    /*
    * These fields correspond to those in KLPPProjection.
    */
    int num_crossings;
    int num_free_loops;
    int num_components;
    LCCrossing *crossings;
};

struct LCCrossing
{
    /*
    * The first four fields correspond to those in KLPPProjection.
    */
    LCCrossing *neighbor[2][2];
    KLPStrandType strand[2][2];
    KLPCrossingType handedness;
    int component[2];

    /*
    * make_projection_connected() keeps track of which LCCrossings
    * it has placed on its queue.
    */

```

```

    Boolean        visited;

    /*
     * The four Tetrahedra incident to a crossing are numbered
     * according to the standard numbering of the quadrants,
     * except that we start with 0 instead of 1.
     *
     *
     *           KLPStrandY
     *           ^
     *       1  |  0
     *   ----+----> KLPStrandX
     *       2  |  3
     *           |
     */
    Tetrahedron    *tet[4];
};

static LCPProjection    *external_to_internal(KLPPProjection *external_link_projection);
static void            free_internal_projection(LCPProjection *internal_link_projection);
static void            add_nugatory_crossings_to_free_loops(LCPProjection *
    internal_link_projection);
static void            resize_crossing_array(LCPProjection *internal_link_projection, int
    new_array_size);
static void            make_projection_connected(LCPProjection *internal_link_projection);
static Boolean        projection_is_connected(LCPProjection *internal_link_projection,
    LCCrossing **crossing0, LCCrossing **crossing1);
static void            do_Reidemeister_II(LCPProjection *internal_link_projection,
    LCCrossing *crossing0, LCCrossing *crossing1);
static void            make_all_components_have_crossings(LCPProjection *
    internal_link_projection);
static void            add_nugatory_crossing(LCPProjection *internal_link_projection, int
    component_index);
static Triangulation    *create_basic_triangulation(LCPProjection *internal_link_projection)
;
static void            create_real_cusps(LCPProjection *internal_link_projection,
    Triangulation *manifold);
static void            create_finite_vertices(LCPProjection *internal_link_projection,
    Triangulation *manifold);
static void            add_peripheral_curves(LCPProjection *internal_link_projection);
static void            clear_peripheral_curves(LCPProjection *internal_link_projection);
static void            add_longitudes(LCPProjection *internal_link_projection);
static void            add_meridians(LCPProjection *internal_link_projection);
static void            adjust_longitudes(LCPProjection *internal_link_projection);

Triangulation *triangulate_link_complement(
    KLPPProjection    *aLinkProjection)
{
    LCPProjection    *internal_link_projection;
    Triangulation    *manifold;

    /*
     * We shouldn't be called with aLinkProjection == NULL,
     * but let's check just to be safe.
     */
    if (aLinkProjection == NULL)
        return NULL;

    /*
     * Ignore empty projections.
     */
    if (aLinkProjection->num_components == 0)
        return NULL;

    /*
     * Transfer aLinkProjection to our internal data structure.
     */
    internal_link_projection = external_to_internal(aLinkProjection);

    /*
     * Are there any free loops?
     */
    if (internal_link_projection->num_free_loops != 0)

```

```

    add_nugatory_crossings_to_free_loops(internal_link_projection);

    /*
     * If the link projection isn't already connected,
     * add a few "unnecessary" crossings to make it so.
     */
    make_projection_connected(internal_link_projection);

    /*
     * Make sure each link component has at least one overcrossing
     * and at least one undercrossing, adding nugatory crossings if needed.
     */
    make_all_components_have_crossings(internal_link_projection);

    /*
     * Set up the basic Triangulation data structure. Allocate the
     * Tetrahedra and set their neighbor and gluing fields.
     * Don't worry about the Cusps or EdgeClasses just yet.
     */
    manifold = create_basic_triangulation(internal_link_projection);

    /*
     * The triangulation we just created is already oriented.
     */
    manifold->orientability = oriented_manifold;

    /*
     * Set up the cusps.
     */
    create_real_cusps(internal_link_projection, manifold);
    create_finite_vertices(internal_link_projection, manifold);

    /*
     * Set up the peripheral curves.
     */
    add_peripheral_curves(internal_link_projection);

    /*
     * We're done with the internal_link_projection.
     * The external projection (aLinkProjection) which got passed in
     * remains untouched.
     */
    free_internal_projection(internal_link_projection);

    /*
     * Add a generic set of EdgeClasses.
     */
    create_edge_classes(manifold);
    orient_edge_classes(manifold);

    /*
     * Absorb the finite vertices at the north and south poles
     * into the cusps.
     */
    remove_finite_vertices(manifold);

    /*
     * Try to find the complete hyperbolic structure.
     */
    find_complete_hyperbolic_structure(manifold);

    return manifold;
}

static LCPProjection *external_to_internal(
    KLPPProjection *external_link_projection)
{
    LCPProjection *internal_link_projection;
    int i, j, k;

    internal_link_projection = NEW_STRUCT(LCPProjection);

```

```

internal_link_projection->num_crossings      = external_link_projection->num_crossings;
internal_link_projection->num_free_loops     = external_link_projection->num_free_loops;
internal_link_projection->num_components     = external_link_projection->num_components;
internal_link_projection->crossings         = NEW_ARRAY(internal_link_projection->num_crossings, LCCrossing);

for (i = 0; i < internal_link_projection->num_crossings; i++)
{
    for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
            internal_link_projection->crossings[i].neighbor[j][k]
                = internal_link_projection->crossings
                    + (external_link_projection->crossings[i].neighbor[j][k]
                        - external_link_projection->crossings);

    for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
            internal_link_projection->crossings[i].strand[j][k]
                = external_link_projection->crossings[i].strand[j][k];

    internal_link_projection->crossings[i].handedness
        = external_link_projection->crossings[i].handedness;

    for (j = 0; j < 2; j++)
        internal_link_projection->crossings[i].component[j]
            = external_link_projection->crossings[i].component[j];

    internal_link_projection->crossings[i].visited = FALSE;

    for (j = 0; j < 4; j++)
        internal_link_projection->crossings[i].tet[j] = NULL;
}

return internal_link_projection;
}

static void free_internal_projection(
    LCProjection *internal_link_projection)
{
    if (internal_link_projection != NULL)
    {
        if (internal_link_projection->crossings != NULL)
            my_free(internal_link_projection->crossings);

        my_free(internal_link_projection);
    }
}

static void add_nugatory_crossings_to_free_loops(
    LCProjection *internal_link_projection)
{
    LCCrossing *new_crossing;
    Boolean *component_has_crossings;
    int next_component,
        i,
        j;

    if (internal_link_projection->num_free_loops <= 0)
        uFatalError("add_nugatory_crossings_to_free_loops", "link_complement");

    /*
     * Transfer the existing crossings to a bigger array.
     */
    resize_crossing_array( internal_link_projection,
                           internal_link_projection->num_crossings
                           + internal_link_projection->num_free_loops);

    /*
     * Note which components already have crossings.
     */
    component_has_crossings = NEW_ARRAY(internal_link_projection->num_components, Boolean);
    for (i = 0; i < internal_link_projection->num_components; i++)

```

```

        component_has_crossings[i] = FALSE;
    for (i = 0; i < internal_link_projection->num_crossings; i++)
    {
        component_has_crossings[internal_link_projection->crossings[i].component
[KLPStrandX]] = TRUE;
        component_has_crossings[internal_link_projection->crossings[i].component
[KLPStrandY]] = TRUE;
    }
    next_component = 0;

    /*
     * Add the new nugatory crossings, one for each free loop.
     */
    while (internal_link_projection->num_free_loops > 0)
    {
        new_crossing = &internal_link_projection->crossings[internal_link_projection->
num_crossings];

        for (i = 0; i < 2; i++) /* i = KLPStrandX, KLPStrandY */
            for (j = 0; j < 2; j++) /* j = KLPBackward, KLPForward */
            {
                new_crossing->neighbor[i][j] = new_crossing;
                new_crossing->strand [i][j] = !i;
            }

        new_crossing->handedness = KLPHalfTwistCL;

        /*
         * Move next_component to the first component which has no crossings.
         */
        while (component_has_crossings[next_component] == TRUE)
        {
            next_component++;
            if (next_component == internal_link_projection->num_components)
                uFatalError("add_nugatory_crossings_to_free_loops", "link_complement");
        }

        new_crossing->component[KLPStrandX] = next_component;
        new_crossing->component[KLPStrandY] = next_component;
        component_has_crossings[next_component] = TRUE;

        internal_link_projection->num_crossings++;
        internal_link_projection->num_free_loops--;
    }

    /*
     * Free local memory.
     */
    my_free(component_has_crossings);
}

static void resize_crossing_array(
    LCPProjection *internal_link_projection,
    int new_array_size)
{
    /*
     * Resize the crossing array.
     * Do NOT change num_crossings.
     */

    LCCrossing *old_array,
                *new_array;
    int i,
        j,
        k;

    if (new_array_size < internal_link_projection->num_crossings)
        uFatalError("resize_crossing_array", "link_complement");

    old_array = internal_link_projection->crossings;
    new_array = NEW_ARRAY(new_array_size, LCCrossing);

    for (i = 0; i < internal_link_projection->num_crossings; i++)

```

```
{
    /*
     * Copy everything . . .
     */
    new_array[i] = old_array[i];

    /*
     * . . . then revise the pointers.
     */
    for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
            new_array[i].neighbor[j][k] = &new_array[old_array[i].neighbor[j][k] -
old_array];
}

/*
 * Just to be safe, set uninitialized pointers to NULL.
 */
for (i = internal_link_projection->num_crossings; i < new_array_size; i++)
{
    for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
            new_array[i].neighbor[j][k] = NULL;
    for (j = 0; j < 4; j++)
        new_array[i].tet[j] = NULL;
}

my_free(old_array);

internal_link_projection->crossings = new_array;
}

static void make_projection_connected(
    LCPProjection      *internal_link_projection)
{
    /*
     * Check whether the projection is disconnected, and if it is,
     * do a Reidemeister type II move to create an "unnecessary" overlap
     * which merges two of the connected components. Repeat as necessary.
     */
    /*
     *
     * \   /           becomes       |   |
     *  \| /             \          |   |
     *   \|              \|         |   |
     *    /               /          |   |
     *   /                /           \
     */
    LCCrossing      *crossing0,
                    *crossing1;

    while (projection_is_connected( internal_link_projection,
                                    &crossing0,
                                    &crossing1) == FALSE)

        do_Reidemeister_II(internal_link_projection, crossing0, crossing1);
}

static Boolean projection_is_connected(
    LCPProjection      *internal_link_projection,
    LCCrossing         **crossing0,
    LCCrossing         **crossing1)
{
    int              i,
                    j,
                    queue_begin,
                    queue_end;
    LCCrossing       *queue,
                    *crossing;
    Boolean           is_connected;
```



```

/*
 * Check whether the projection is connected.
 * If it's connected,
 *     set *crossing0 and *crossing1 to NULL, and return TRUE.
 * If it isn't connected,
 *     set *crossing0 and *crossing1 to be pointers to LCCrossings
 *     in different connected components, and return FALSE.
 */

/*
 * This routine assumes (1) the link is nonempty,
 * and (2) there are no free loops.
 */
if (internal_link_projection->num_components == 0
    || internal_link_projection->num_free_loops > 0)
    uFatalError("projection_is_connected", "link_complement");

/*
 * Mark all crossings as unvisited.
 */
for (i = 0; i < internal_link_projection->num_crossings; i++)
    internal_link_projection->crossings[i].visited = FALSE;

/*
 * Start a queue to keep track of crossings which have been
 * visited, but whose neighbors have not yet been examined.
 */
queue = NEW_ARRAY(internal_link_projection->num_crossings, LCCrossing *);

/*
 * Put the zeroth crossing onto the queue.
 */
queue[0] = &internal_link_projection->crossings[0];
queue[0]->visited = TRUE;
queue_begin = 0;
queue_end = 0;

/*
 * Process the queue.
 */
while (queue_begin <= queue_end)
{
    /*
     * Pull a pointer off the front of the queue.
     */
    crossing = queue[queue_begin++];

    /*
     * Check its four neighbors.
     */
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
        {
            /*
             * If a neighbor hasn't yet been visited . . .
             */
            if (crossing->neighbor[i][j]->visited == FALSE)
            {
                /*
                 * . . . add it to the queue.
                 */
                crossing->neighbor[i][j]->visited = TRUE;
                queue[++queue_end] = crossing->neighbor[i][j];
            }
        }
}

/*
 * Do a quick "unnecessary" error check.
 */
if (queue_end > internal_link_projection->num_crossings - 1)
    uFatalError("projection_is_connected", "link_complement");

/*
 * Free the queue.
 */

```

```

*/
my_free(queue);

/*
 * The link projection is connected iff we have visited all crossings.
 */
*crossing0 = NULL;
*crossing1 = NULL;
if (queue_end == internal_link_projection->num_crossings - 1)
    is_connected = TRUE;
else
{
    is_connected = FALSE;
    for (i = 0; i < internal_link_projection->num_crossings; i++)
    {
        if (internal_link_projection->crossings[i].visited == TRUE)
            *crossing0 = &internal_link_projection->crossings[i];
        else
            *crossing1 = &internal_link_projection->crossings[i];
    }
}

return is_connected;
}

static void do_Reidemeister_II(
LCPProjection      *internal_link_projection,
LCCrossing         *crossing0,
LCCrossing         *crossing1)
{
    /*
     * Crossing0 and crossing1 are assumed to lie in different
     * connected components of the link projection. Do a type II
     * Reidemeister move so that their forward X-strands pass over
     * one another. Note that the two connected components may always
     * be brought into position to do this (without creating any
     * additional crossings) if one considers them as link projections
     * on the 2-sphere, not just on the plane.
     *
     *
     *          \           /              (crossing2) (crossing3)
     *           |           |             |           |
     *           |           |             |           |
     *          /           \             |           |
     * crossing0 crossing1 becomes crossing0 crossing1
     *                               |           |
     *                               |           |
     *                               |           |
     *                              /           \
     *                          crossingA crossingB
     */

int      crossing_index0,
          crossing_index1;
LCCrossing *crossingA,
            *crossingB,
            *crossing2,
            *crossing3;
KLPStrandType strand2,
               strand3;

    /*
     * Make room for the two new crossings, and give them names.
     * Also revise the pointers to crossing0 and crossing1.
     */
    crossing_index0 = crossing0 - internal_link_projection->crossings;
    crossing_index1 = crossing1 - internal_link_projection->crossings;
    resize_crossing_array(internal_link_projection,
                           internal_link_projection->num_crossings + 2);
    internal_link_projection->num_crossings += 2;
    crossing0 = &internal_link_projection->crossings[crossing_index0];
    crossing1 = &internal_link_projection->crossings[crossing_index1];
    crossingA = &internal_link_projection->crossings[internal_link_projection->
num_crossings - 2];
    crossingB = &internal link projection->crossings[internal link projection->

```

```

    num_crossings - 1];

/*
 * Give names to the crossing which originally follow
 * crossings 0 and 1, and note which of their strands we're using.
 */
crossing2 = crossing0->neighbor[KLPStrandX][KLPForward];
strand2   = crossing0->strand  [KLPStrandX][KLPForward];
crossing3 = crossing1->neighbor[KLPStrandX][KLPForward];
strand3   = crossing1->strand  [KLPStrandX][KLPForward];

/*
 * Work the two new crossings into the link projection,
 * as illustrated above.
 */

crossingA->neighbor[KLPStrandX][KLPBackward] = crossing0;
crossingA->neighbor[KLPStrandX][KLPForward ] = crossingB;
crossingA->neighbor[KLPStrandY][KLPBackward] = crossing1;
crossingA->neighbor[KLPStrandY][KLPForward ] = crossingB;

crossingA->strand[KLPStrandX][KLPBackward] = KLPStrandX;
crossingA->strand[KLPStrandX][KLPForward ] = KLPStrandY;
crossingA->strand[KLPStrandY][KLPBackward] = KLPStrandX;
crossingA->strand[KLPStrandY][KLPForward ] = KLPStrandX;

crossingA->handedness = KLPHalfTwistCL;

crossingA->component[KLPStrandX] = crossing0->component[KLPStrandX];
crossingA->component[KLPStrandY] = crossing1->component[KLPStrandX];

crossingB->neighbor[KLPStrandX][KLPBackward] = crossingA;
crossingB->neighbor[KLPStrandX][KLPForward ] = crossing3;
crossingB->neighbor[KLPStrandY][KLPBackward] = crossingA;
crossingB->neighbor[KLPStrandY][KLPForward ] = crossing2;

crossingB->strand[KLPStrandX][KLPBackward] = KLPStrandY;
crossingB->strand[KLPStrandX][KLPForward ] = strand3;
crossingB->strand[KLPStrandY][KLPBackward] = KLPStrandX;
crossingB->strand[KLPStrandY][KLPForward ] = strand2;

crossingB->handedness = KLPHalfTwistCCL;

crossingB->component[KLPStrandX] = crossing1->component[KLPStrandX];
crossingB->component[KLPStrandY] = crossing0->component[KLPStrandX];

crossing0->neighbor[KLPStrandX][KLPForward] = crossingA;
crossing0->strand  [KLPStrandX][KLPForward] = KLPStrandX;

crossing1->neighbor[KLPStrandX][KLPForward] = crossingA;
crossing1->strand  [KLPStrandX][KLPForward] = KLPStrandY;

crossing2->neighbor[strand2][KLPBackward] = crossingB;
crossing2->strand  [strand2][KLPBackward] = KLPStrandY;

crossing3->neighbor[strand3][KLPBackward] = crossingB;
crossing3->strand  [strand3][KLPBackward] = KLPStrandX;
}

static void make_all_components_have_crossings(
    LCPProjection *internal_link_projection)
{
    /*
     * Add nugatory crossings if necessary to ensure that each
     * link component has both overcrossings and undercrossings.
     */

    Boolean *undercrossing_flags,
            *overcrossing_flags;
    int i;

    /*
     * The caller should have already checked that there are no free loops.

```

```

    */
    if (internal_link_projection->num_free_loops != 0)
        uFatalError("make_all_components_have_crossings", "link_complement");

    undercrossing_flags = NEW_ARRAY(internal_link_projection->num_components, Boolean);
    overcrossing_flags = NEW_ARRAY(internal_link_projection->num_components, Boolean);

    for (i = 0; i < internal_link_projection->num_components; i++)
    {
        undercrossing_flags[i] = FALSE;
        overcrossing_flags[i] = FALSE;
    }

    for (i = 0; i < internal_link_projection->num_crossings; i++)
    {
        switch (internal_link_projection->crossings[i].handedness)
        {
            case KLPHalfTwistCL:
                undercrossing_flags[internal_link_projection->crossings[i].component
[KLPStrandY]] = TRUE;
                overcrossing_flags[internal_link_projection->crossings[i].component
[KLPStrandX]] = TRUE;
                break;

            case KLPHalfTwistCCL:
                undercrossing_flags[internal_link_projection->crossings[i].component
[KLPStrandX]] = TRUE;
                overcrossing_flags[internal_link_projection->crossings[i].component
[KLPStrandY]] = TRUE;
                break;

            default:
                uFatalError("make_all_components_have_crossings", "link_complement");
        }
    }

    for (i = 0; i < internal_link_projection->num_components; i++)
        if (undercrossing_flags[i] == FALSE
            || overcrossing_flags[i] == FALSE)
            add_nugatory_crossing(internal_link_projection, i);

    my_free(undercrossing_flags);
    my_free(overcrossing_flags);
}

static void add_nugatory_crossing(
    LCPProjection *internal_link_projection,
    int component_index)
{
    /*
     * The component of the given component_index has only overcrossings
     * or only undercrossings. Add a nugatory crossing so that the
     * component will have both.
     */

    LCCrossing *crossingA,
                *crossingB,
                *crossingC;
    KLPStrandType strandC;
    int i;

    /*
     * The caller should have already checked that there are no free loops.
     */
    if (internal_link_projection->num_free_loops != 0)
        uFatalError("add_nugatory_crossing", "link_complement");

    /*
     * Make room for the new crossing.
     * (Note that we must resize the array before find the pointer
     * to crossingA.)
     */
    resize_crossing_array( internal_link_projection,
                           internal_link_projection->num_crossings + 1);

```

```

/*
 * Lemma. If a component has at least one crossing, then it must
 * appear at least once as an X strand and once as a Y strand.
 *
 * Proof. If the component crosses itself, the result is obvious.
 * If it doesn't cross itself, then it's unknotted and bounds
 * a disk in the plane of the link projection. If some other
 * link component enters the disk as, say, an X strand, it must
 * eventually leave the disk as a Y strand. Q.E.D.
 *
 * Find a crossing where the link component occurs as an X strand.
 */
crossingA = NULL;
for (i = 0; i < internal_link_projection->num_crossings; i++)
    if (internal_link_projection->crossings[i].component[KLPStrandX] ==
        component_index)
        crossingA = &internal_link_projection->crossings[i];
if (crossingA == NULL)
    uFatalError("add_nugatory_crossing", "link_complement");

/*
 * Let crossingC be the crossing which follows crossingA
 * in the forward X direction.
 */
crossingC = crossingA->neighbor[KLPStrandX][KLPForward];
strandC = crossingA->strand [KLPStrandX][KLPForward];

/*
 * Create crossingB, which will be a nugatory crossing lying
 * between crossingA and crossingC.
 *
 *
 *
 *      \
 *       \
 *        \ crossingB
 *       /
 *      /
 * crossingA      crossingC
 *
 */

internal_link_projection->num_crossings++;
crossingB = &internal_link_projection->crossings[internal_link_projection->
num_crossings - 1];

crossingA->neighbor[KLPStrandX][KLPForward] = crossingB;
crossingA->strand [KLPStrandX][KLPForward] = KLPStrandY;

crossingB->neighbor[KLPStrandX][KLPBackward] = crossingB;
crossingB->neighbor[KLPStrandX][KLPForward ] = crossingC;
crossingB->neighbor[KLPStrandY][KLPBackward] = crossingA;
crossingB->neighbor[KLPStrandY][KLPForward ] = crossingB;

crossingB->strand[KLPStrandX][KLPBackward] = KLPStrandY;
crossingB->strand[KLPStrandX][KLPForward ] = strandC;
crossingB->strand[KLPStrandY][KLPBackward] = KLPStrandX;
crossingB->strand[KLPStrandY][KLPForward ] = KLPStrandX;

crossingB->handedness = KLPHalfTwistCCL;

crossingB->component[KLPStrandX] = component_index;
crossingB->component[KLPStrandY] = component_index;

crossingC->neighbor[strandC][KLPBackward] = crossingB;
crossingC->strand [strandC][KLPBackward] = KLPStrandX;
}

static Triangulation *create_basic_triangulation(
    LCPProjection      *internal_link_projection)
{
    Triangulation      *manifold;
    LCCrossing          *theCrossing,
                        *theNbrCrossing;
    KLPStrandType       theNbrStrand;

```

```

int            i,
               j,
               k;

/*
 * Allocate and initialize the Triangulation structure itself.
 */
manifold = NEW_STRUCT(Triangulation);
initialize_triangulation(manifold);

/*
 * Allocate and initialize the four Tetrahedra incident to each crossing.
 */
for (i = 0; i < internal_link_projection->num_crossings; i++)
    for (j = 0; j < 4; j++)
    {
        internal_link_projection->crossings[i].tet[j] = NEW_STRUCT(Tetrahedron);
        initialize_tetrahedron(internal_link_projection->crossings[i].tet[j]);
        INSERT_BEFORE(internal_link_projection->crossings[i].tet[j], &manifold->
tet_list_end);
        manifold->num_tetrahedra++;
    }

/*
 * Interpret the vertex indices of each Tetrahedron as follows:
 *
 * Vertex 0 is at the south pole.
 * Vertex 1 is at the north pole.
 * Vertices 2 and 3 are chosen so that the Tetrahedron is right_handed
 * according to the definition of Orientation in kernel_typedefs.h.
 */

/*
 * Set the neighbor fields.
 *
 * To make sense of this, please refer to the sketch you made while
 * reading the documentation at the top of this file.
 */
for (i = 0; i < internal_link_projection->num_crossings; i++)
{
    /*
     * Consider the group of four Tetrahedra surrounding crossing i.
     */
    theCrossing = &internal_link_projection->crossings[i];

    /*
     * Set the neighbor fields within the group.
     */
    if (theCrossing->handedness == KLPHalfTwistCL)
    {
        theCrossing->tet[0]->neighbor[0] = theCrossing->tet[1];
        theCrossing->tet[0]->neighbor[1] = theCrossing->tet[3];

        theCrossing->tet[1]->neighbor[0] = theCrossing->tet[0];
        theCrossing->tet[1]->neighbor[1] = theCrossing->tet[2];

        theCrossing->tet[2]->neighbor[0] = theCrossing->tet[3];
        theCrossing->tet[2]->neighbor[1] = theCrossing->tet[1];

        theCrossing->tet[3]->neighbor[0] = theCrossing->tet[2];
        theCrossing->tet[3]->neighbor[1] = theCrossing->tet[0];
    }
    else
    {
        theCrossing->tet[0]->neighbor[0] = theCrossing->tet[3];
        theCrossing->tet[0]->neighbor[1] = theCrossing->tet[1];

        theCrossing->tet[1]->neighbor[0] = theCrossing->tet[2];
        theCrossing->tet[1]->neighbor[1] = theCrossing->tet[0];

        theCrossing->tet[2]->neighbor[0] = theCrossing->tet[1];
        theCrossing->tet[2]->neighbor[1] = theCrossing->tet[3];

        theCrossing->tet[3]->neighbor[0] = theCrossing->tet[0];
    }
}

```

```

        theCrossing->tet[3]->neighbor[1] = theCrossing->tet[2];
    }

    /*
     * Set the neighbor fields connecting this group to other groups.
     */

    /*
     * backward x-direction
     */
    theNbrCrossing = theCrossing->neighbor[KLPStrandX][KLPBackward];
    theNbrStrand = theCrossing->strand [KLPStrandX][KLPBackward];
    if (theNbrStrand == KLPStrandX)
    {
        theCrossing->tet[2]->neighbor[3] = theNbrCrossing->tet[3];
        theCrossing->tet[1]->neighbor[2] = theNbrCrossing->tet[0];
    }
    else
    {
        theCrossing->tet[2]->neighbor[3] = theNbrCrossing->tet[0];
        theCrossing->tet[1]->neighbor[2] = theNbrCrossing->tet[1];
    }

    /*
     * forward x-direction
     */
    theNbrCrossing = theCrossing->neighbor[KLPStrandX][KLPForward];
    theNbrStrand = theCrossing->strand [KLPStrandX][KLPForward];
    if (theNbrStrand == KLPStrandX)
    {
        theCrossing->tet[0]->neighbor[3] = theNbrCrossing->tet[1];
        theCrossing->tet[3]->neighbor[2] = theNbrCrossing->tet[2];
    }
    else
    {
        theCrossing->tet[0]->neighbor[3] = theNbrCrossing->tet[2];
        theCrossing->tet[3]->neighbor[2] = theNbrCrossing->tet[3];
    }

    /*
     * backward y-direction
     */
    theNbrCrossing = theCrossing->neighbor[KLPStrandY][KLPBackward];
    theNbrStrand = theCrossing->strand [KLPStrandY][KLPBackward];
    if (theNbrStrand == KLPStrandX)
    {
        theCrossing->tet[3]->neighbor[3] = theNbrCrossing->tet[3];
        theCrossing->tet[2]->neighbor[2] = theNbrCrossing->tet[0];
    }
    else
    {
        theCrossing->tet[3]->neighbor[3] = theNbrCrossing->tet[0];
        theCrossing->tet[2]->neighbor[2] = theNbrCrossing->tet[1];
    }

    /*
     * forward y-direction
     */
    theNbrCrossing = theCrossing->neighbor[KLPStrandY][KLPForward];
    theNbrStrand = theCrossing->strand [KLPStrandY][KLPForward];
    if (theNbrStrand == KLPStrandX)
    {
        theCrossing->tet[1]->neighbor[3] = theNbrCrossing->tet[1];
        theCrossing->tet[0]->neighbor[2] = theNbrCrossing->tet[2];
    }
    else
    {
        theCrossing->tet[1]->neighbor[3] = theNbrCrossing->tet[2];
        theCrossing->tet[0]->neighbor[2] = theNbrCrossing->tet[3];
    }
}

/*
 * Set the gluing fields.

```

```

    *
    * A very pleasant consequence of our indexing scheme is that
    * every gluing in the whole triangulation is 2310!
    */
    for (i = 0; i < internal_link_projection->num_crossings; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                internal_link_projection->crossings[i].tet[j]->gluing[k] = PERMUTATION2310;

    return manifold;
}

static void create_real_cusps(
    LCProjection *internal_link_projection,
    Triangulation *manifold)
{
    Cusp **theCusps,
        *theXCusp,
        *theYCusp;
    LCCrossing *theCrossing;
    int i;

    /*
     * Use a temporary array to keep the freshly created Cusps organized.
     */
    theCusps = NEW_ARRAY(internal_link_projection->num_components, Cusp *);

    /*
     * Create and initialize the Cusps.
     */

    manifold->num_cusps = 0;
    manifold->num_or_cusps = 0;
    manifold->num_nonor_cusps = 0;

    for (i = 0; i < internal_link_projection->num_components; i++)
    {
        theCusps[i] = NEW_STRUCT(Cusp);
        initialize_cusp(theCusps[i]);
        theCusps[i]->topology = torus_cusp;
        theCusps[i]->index = i;
        theCusps[i]->is_finite = FALSE;
        INSERT_BEFORE(theCusps[i], &manifold->cusp_list_end);
        manifold->num_cusps++;
        manifold->num_or_cusps++;
    }

    /*
     * Assign the Cusps to the ideal vertices of the Tetrahedra.
     * Don't worry about the finite vertices for now.
     */

    for (i = 0; i < internal_link_projection->num_crossings; i++)
    {
        theCrossing = &internal_link_projection->crossings[i];

        theXCusp = theCusps[theCrossing->component[KLPStrandX]];
        theYCusp = theCusps[theCrossing->component[KLPStrandY]];

        theCrossing->tet[0]->cusp[2] = theXCusp;
        theCrossing->tet[0]->cusp[3] = theYCusp;

        theCrossing->tet[1]->cusp[2] = theYCusp;
        theCrossing->tet[1]->cusp[3] = theXCusp;

        theCrossing->tet[2]->cusp[2] = theXCusp;
        theCrossing->tet[2]->cusp[3] = theYCusp;

        theCrossing->tet[3]->cusp[2] = theYCusp;
        theCrossing->tet[3]->cusp[3] = theXCusp;
    }

    /*

```



```

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

static void create_finite_vertices(
    LCPProjection    *internal_link_projection,
    Triangulation    *manifold)
{
    Cusp    *thePoles[2];
    int      i,
             j,
             k;

    /*
     *   Create finite vertices for the north and south poles.
     */
    for (i = 0; i < 2; i++)
    {
        thePoles[i] = NEW_STRUCT(Cusp);
        initialize_cusp(thePoles[i]);
        thePoles[i]->index      = i - 2;    /* indices are -1 and -2 */
        thePoles[i]->is_finite  = TRUE;
        INSERT_BEFORE(thePoles[i], &manifold->cusp_list_end);
    }

    /*
     *   Assign the finite vertices to the Tetrahedra.
     */
    for (i = 0; i < internal_link_projection->num_crossings; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 2; k++)
                internal_link_projection->crossings[i].tet[j]->cusp[k] = thePoles[k];
}

static void add_peripheral_curves(
    LCPProjection    *internal_link_projection)
{
    clear_peripheral_curves (internal_link_projection);
    add_longitudes          (internal_link_projection);
    add_meridians           (internal_link_projection);
    adjust_longitudes       (internal_link_projection);
}

static void clear_peripheral_curves(
    LCPProjection    *internal_link_projection)
{
    int      i,
             j,
             c,
             h,
             v,
             f;

    /*
     *   initialize_tetrahedron() has already initialized the peripheral
     *   curves to zero, but we reinitialize them just to be safe.
     */
    for (i = 0; i < internal_link_projection->num_crossings; i++)
        for (j = 0; j < 4; j++)
            for (c = 0; c < 2; c++)
                for (h = 0; h < 2; h++)
                    for (v = 0; v < 4; v++)
                        for (f = 0; f < 4; f++)
                            internal_link_projection->crossings[i].tet[j]->curve[c][h][v] = 0;

    [f] = 0;
}

static void add_longitudes(
    LCPProjection    *internal_link_projection)

```

```

{
    /*
     * Construct longitudes which run along the right side of each link
     * component in the forward direction. Eventually add_peripheral_curves()
     * will call adjust_longitudes() to add in some number of meridians
     * to obtain the homologically trivial longitudes.
     */

    int i;
    LCCrossing *theCrossing;

    for (i = 0; i < internal_link_projection->num_crossings; i++)
    {
        theCrossing = &internal_link_projection->crossings[i];

        /*
         * x strand
         */

        theCrossing->tet[2]->curve[L][right_handed][2][3] = +1;
        theCrossing->tet[3]->curve[L][right_handed][3][2] = -1;

        if (theCrossing->handedness == KLPHalfTwistCL)
        {
            theCrossing->tet[2]->curve[L][right_handed][2][0] = -1;
            theCrossing->tet[3]->curve[L][right_handed][3][0] = +1;
        }
        else
        {
            theCrossing->tet[2]->curve[L][right_handed][2][1] = -1;
            theCrossing->tet[3]->curve[L][right_handed][3][1] = +1;
        }

        /*
         * y strand
         */

        theCrossing->tet[3]->curve[L][right_handed][2][3] = +1;
        theCrossing->tet[0]->curve[L][right_handed][3][2] = -1;

        if (theCrossing->handedness == KLPHalfTwistCL)
        {
            theCrossing->tet[3]->curve[L][right_handed][2][1] = -1;
            theCrossing->tet[0]->curve[L][right_handed][3][1] = +1;
        }
        else
        {
            theCrossing->tet[3]->curve[L][right_handed][2][0] = -1;
            theCrossing->tet[0]->curve[L][right_handed][3][0] = +1;
        }
    }
}

static void add_meridians(
    LCPProjection *internal_link_projection)
{
    Boolean *theArray;
    int i,
        theComponent;
    LCCrossing *theCrossing,
        *theNextCrossing;
    KLPStrandType theStrand,
        theNextStrand;
    Boolean theStrandGoesOver,
        theNextStrandGoesOver;

    /*
     * Construct one meridian for each link component.
     * Define it using the right hand rule (grasp the link component
     * with your right hand -- if your thumb points in the direction
     * of the positive longitude, your fingers will wrap around in the
     * direction of the positive meridian).
     */

```

```

/*
 * The following proposition allows us to examine only x-strands.
 *
 * Proposition. Each link component appears as the x-strand
 * at some crossing.
 *
 * Proof. If a component C crosses itself, it's both the x- and
 * y-strand at that crossing. So assume it doesn't cross itself,
 * in which case it bounds a disk D in the link projection. By the
 * Essential Assumption in the documentation at the top of this file,
 * the component C has crossings. Trace some other component C' which
 * crosses C. If C is an x-strand where C' enters the disk D, it's
 * a y-strand where C' leaves D, and vice versa. Q.E.D.
 */

/*
 * Use a temporary array to record which meridians have been constructed.
 */

theArray = NEW_ARRAY(internal_link_projection->num_components, Boolean);

for (i = 0; i < internal_link_projection->num_components; i++)
    theArray[i] = FALSE;

for (i = 0; i < internal_link_projection->num_crossings; i++)
{
    theComponent = internal_link_projection->crossings[i].component[KLPStrandX];

    if (theArray[theComponent] == FALSE)
    {
        /*
         * We've found the desired component, but drawing the
         * meridian isn't as simple as one would hope, because
         * either the top or the bottom of the x-strand is
         * incident to "shrinking bigons", as described in the
         * documentation at the top of this file.
         *
         * To simplify matters, we move along theComponent until
         * we reach an overcrossing immediately followed by an
         * undercrossing. (According to the Essential Assumption
         * each link component has both undercrossings and
         * overcrossings, so we are sure to find such a point.)
         *
         * Note: An "overcrossing" (resp. an "undercrossing") is
         * a crossing at which theComponent passes over (resp. under)
         * another strand.
         */

        theCrossing      = &internal_link_projection->crossings[i];
        theStrand         = KLPStrandX;
        theStrandGoesOver = (theCrossing->handedness == KLPHalfTwistCL) ?
                            (theStrand == KLPStrandX) :
                            (theStrand == KLPStrandY);

        theNextCrossing  = theCrossing->neighbor[theStrand][KLPForward];
        theNextStrand     = theCrossing->strand [theStrand][KLPForward];
        theNextStrandGoesOver = (theNextCrossing->handedness == KLPHalfTwistCL) ?
                                (theNextStrand == KLPStrandX) :
                                (theNextStrand == KLPStrandY);

        while (theStrandGoesOver == FALSE || theNextStrandGoesOver == TRUE)
        {
            theCrossing      = theNextCrossing;
            theStrand         = theNextStrand;
            theStrandGoesOver = theNextStrandGoesOver;

            theNextCrossing  = theCrossing->neighbor[theStrand][KLPForward];
            theNextStrand     = theCrossing->strand [theStrand][KLPForward];
            theNextStrandGoesOver = (theNextCrossing->handedness == KLPHalfTwistCL) ?
                                    (theNextStrand == KLPStrandX) :
                                    (theNextStrand == KLPStrandY);
        }
    }
}

```

```

    /*
     * We've reach a point where theStrandGoesOver == TRUE and
     * theNextStrandGoesOver == FALSE, so we can draw the meridian.
     */

    if (theStrand == KLPStrandX) /* => KLPHalfTwistCL */
    {
        theCrossing->tet[3]->curve[M][right_handed][3][1] = -1;
        theCrossing->tet[3]->curve[M][right_handed][3][2] = +1;

        theCrossing->tet[0]->curve[M][right_handed][2][1] = +1;
        theCrossing->tet[0]->curve[M][right_handed][2][3] = -1;
    }
    else /* theStrand == KLPStrandY => KLPHalfTwistCCL */
    {
        theCrossing->tet[0]->curve[M][right_handed][3][1] = -1;
        theCrossing->tet[0]->curve[M][right_handed][3][2] = +1;

        theCrossing->tet[1]->curve[M][right_handed][2][1] = +1;
        theCrossing->tet[1]->curve[M][right_handed][2][3] = -1;
    }

    if (theNextStrand == KLPStrandX) /* => KLPHalfTwistCCL */
    {
        theNextCrossing->tet[1]->curve[M][right_handed][3][0] = -1;
        theNextCrossing->tet[1]->curve[M][right_handed][3][2] = +1;

        theNextCrossing->tet[2]->curve[M][right_handed][2][0] = +1;
        theNextCrossing->tet[2]->curve[M][right_handed][2][3] = -1;
    }
    else /* theNextStrand == KLPStrandY => KLPHalfTwistCL */
    {
        theNextCrossing->tet[2]->curve[M][right_handed][3][0] = -1;
        theNextCrossing->tet[2]->curve[M][right_handed][3][2] = +1;

        theNextCrossing->tet[3]->curve[M][right_handed][2][0] = +1;
        theNextCrossing->tet[3]->curve[M][right_handed][2][3] = -1;
    }

    /*
     * Note that we've added the peripheral curves to this component.
     */
    theArray[theComponent] = TRUE;
}

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

```

```

static void adjust_longitudes(
    LCPProjection *internal_link_projection)
{
    /*
     * To define the canonical longitude on a link component, consider
     * the link component alone, ignoring all other link components,
     * and let the canonical longitude be the one which is homologically
     * trivial in the knot complement. (The direction of the longitude
     * is of course the direction of the link component.) The canonical
     * longitude is well defined up to isotopy.
     *
     * If a link component never crosses itself, then the longitude
     * which runs along the right side of the (thickened) link component
     * is a canonical longitude. Proof: cone to the north pole of  $S^3$ 
     * to see that the longitude bounds a disk.
     *
     * If the link component does cross itself, then when you try coning
     * to the north pole the "disk" will intersect the (thickened) link
     * component in some number of meridians. To obtain the canonical
     * longitude, we must subtract off that number of meridians.
     */
}

```

```

    * Each counterclockwise crossing generates a negative meridian,
    * and each clockwise crossing generates a positive meridian, so
    * we must compute the signed sum of the crossings, and subtract
    * that number of meridians from the longitude.
    */

int      *theSignedSum,
        i,
        j,
        v,
        f,
        theXComponent,
        theYComponent,
        theXSignedSum,
        theYSignedSum;
Tetrahedron *theTet;

/*
 * Allocate a temporary array to hold the signed sum for each component.
 */
theSignedSum = NEW_ARRAY(internal_link_projection->num_components, int);

/*
 * Initialize the signed sums to zero.
 */
for (i = 0; i < internal_link_projection->num_components; i++)
    theSignedSum[i] = 0;

/*
 * Add in -1 (resp. +1) for each counterclockwise (resp. clockwise)
 * crossing where a component crosses itself.
 */
for (i = 0; i < internal_link_projection->num_crossings; i++)
{
    theXComponent = internal_link_projection->crossings[i].component[KLPStrandX];
    theYComponent = internal_link_projection->crossings[i].component[KLPStrandY];

    if (theXComponent == theYComponent)
    {
        if (internal_link_projection->crossings[i].handedness == KLPHalfTwistCL)
            theSignedSum[theXComponent]++;
        else
            theSignedSum[theXComponent]--;
    }
}

/*
 * Subtract the appropriate multiples of the meridians from the
 * longitudes. Note that only vertices 2 and 3 carry peripheral curves.
 */
for (i = 0; i < internal_link_projection->num_crossings; i++)
{
    theXSignedSum = theSignedSum[internal_link_projection->crossings[i].component
[KLPStrandX]];
    theYSignedSum = theSignedSum[internal_link_projection->crossings[i].component
[KLPStrandY]];

    for (j = 0; j < 4; j++)
    {
        theTet = internal_link_projection->crossings[i].tet[j];

        for (v = 2; v < 4; v++)
            for (f = 0; f < 4; f++)
                theTet->curve[L][right_handed][v][f]
                    -= (((j & 0x01) == (v & 0x01)) ? theXSignedSum : theYSignedSum)
                    * theTet->curve[M][right_handed][v][f];
    }
}

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

```

