

```

/*
 * double_cover.c
 *
 * This file provides the function
 *
 *      Triangulation *double_cover(Triangulation *manifold);
 *
 * which computes the orientable double cover of a nonorientable manifold.
 */

#include "kernel.h"

typedef Tetrahedron *TetPtrPair[2];
typedef int          IntPair[2];

static void allocate_storage(Triangulation *manifold, Triangulation **the_cover, TetPtrPair
    **new_tet);
static void set_neighbors_and_gluings(Triangulation *manifold, TetPtrPair *new_tet);
static void lift_peripheral_curves(Triangulation *manifold, TetPtrPair *new_tet);
static void lift_tet_shapes(Triangulation *manifold, TetPtrPair *new_tet);
static void set_cusp_data(Triangulation *manifold, Triangulation *the_cover, TetPtrPair *
    new_tet);

Triangulation *double_cover(
    Triangulation *manifold)
{
    Triangulation *the_cover;
    TetPtrPair    *new_tet;

    /*
     * Make sure the manifold is nonorientable.
     */
    if (manifold->orientability != nonorientable_manifold)
        uFatalError("double_cover", "double_cover");

    /*
     * Number the manifold's Tetrahedra for easy reference.
     */
    number_the_tetrahedra(manifold);

    /*
     * Allocate space for the_cover.
     *
     * The i-th tetrahedron in manifold will have two preimages in
     * the_cover, which we'll record as new_tet[i][0] and new_tet[i][1].
     */
    allocate_storage(manifold, &the_cover, &new_tet);

    /*
     * Set the neighbor and gluing fields of the Tetrahedra.
     */
    set_neighbors_and_gluings(manifold, new_tet);

    /*
     * Lift the peripheral curves to the double cover.
     * The lifted {meridian, longitude} will adhere to the right hand
     * rule relative to the local orientation on each Cusp's double
     * cover (cf. peripheral_curves.c). The manifold does not yet
     * have a global orientation.
     */
    lift_peripheral_curves(manifold, new_tet);

    /*
     * Copy the TetShapes and shape histories.
     */
    lift_tet_shapes(manifold, new_tet);

    /*
     * Set some global information.
     */
    the_cover->num_tetrahedra = 2 * manifold->num_tetrahedra;
    the_cover->solution_type[complete] = manifold->solution_type[complete];
    the_cover->solution_type[filled]   = manifold->solution_type[filled];
}

```

```

/*
 * Create Cusps for the_cover, make sure they're in the same order
 * as their images in the manifold, copy in the Dehn filling
 * coefficients, and set the CuspTopology to torus_cusp.
 * We must do this before orienting the_cover, or else the
 * vertex indices on the_cover will no longer coincide with
 * those on the manifold.
 * Count the cusps.
 */
create_cusps(the_cover);
set_cusp_data(manifold, the_cover, new_tet);
count_cusps(the_cover);

/*
 * Orient the_cover.
 */
orient(the_cover);
if (the_cover->orientability != oriented_manifold)
    uFatalError("double_cover", "double_cover");

/*
 * orient() moves all peripheral curves to the right_handed sheets
 * of the Cusps local double covers. Thus some {meridian, longitude}
 * pairs may no longer adhere to the right-hand rule. Correct them,
 * and adjust the Dehn filling coefficients accordingly.
 */
fix_peripheral_orientations(the_cover);

/*
 * Create and orient the EdgeClasses.
 */
create_edge_classes(the_cover);
orient_edge_classes(the_cover);

/*
 * In principle we could lift the holonomies and cusp shapes from
 * the manifold to the_cover, but the code would be delicate, difficult
 * to write, and bug-prone (it would be sensitive to the details of
 * which orientation the_cover was given and which meridians or
 * longitudes were reversed, so changing other parts of the algorithm
 * would most likely introduce errors here). A more robust approach
 * is to let polish_hyperbolic_structures() "unnecessarily" refine
 * the hyperbolic structure. It will automatically compute the
 * holonomies and cusp shapes. The price we pay is that in all cases
 * we'll have one "unnecessary" iteration of Newton's method, and
 * in cases where the shape histories are nontrivial, we'll end up
 * computing the hyperbolic structure from scratch.
 */
polish_hyperbolic_structures(the_cover);

/*
 * Free local arrays.
 */
my_free(new_tet);

return the_cover;
}

static void allocate_storage(
    Triangulation *manifold,
    Triangulation **the_cover,
    TetPtrPair **new_tet)
{
    int i,
        j;

    /*
     * Allocate space for the new Triangulation.
     */

    *the_cover = NEW_STRUCT(Triangulation);
    initialize_triangulation(*the_cover);

```

```

/*
 * Allocate space for the new Tetrahedra, and the array
 * which organizes them.
 */

new_tet = NEW_ARRAY(manifold->num_tetrahedra, TetPtrPair);

for (i = 0; i < manifold->num_tetrahedra; i++)
    for (j = 0; j < 2; j++)
    {
        (*new_tet)[i][j] = NEW_STRUCT(Tetrahedron);
        initialize_tetrahedron((*new_tet)[i][j]);
        INSERT_BEFORE((*new_tet)[i][j], &(*the_cover)->tet_list_end);
    }
}

static void set_neighbors_and_gluings(
    Triangulation *manifold,
    TetPtrPair *new_tet)
{
    int i, j;
    Tetrahedron *old_tet;
    VertexIndex v;

    for (old_tet = manifold->tet_list_begin.next, i = 0;
        old_tet != &manifold->tet_list_end;
        old_tet = old_tet->next, i++)
    {
        if (old_tet->index != i)
            uFatalError("set_neighbors_and_gluings", "double_cover");

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

            for (v = 0; v < 4; v++)
            {
                new_tet[i][j]->neighbor[v] = new_tet
                    [old_tet->neighbor[v]->index]
                    [parity[old_tet->gluing[v]] == orientation_preserving ? j : !j];

                new_tet[i][j]->gluing[v] = old_tet->gluing[v];
            }
    }
}

static void lift_peripheral_curves(
    Triangulation *manifold,
    TetPtrPair *new_tet)
{
    int i;
    Tetrahedron *old_tet;
    PeripheralCurve c;
    VertexIndex v;
    FaceIndex f;

    /*
     * Lift the peripheral curves from manifold to the_cover.
     *
     * The curves on a torus cusp lift to both preimages in the_cover.
     * In the original nonorientable manifold, the curves are stored on
     * one sheet of the cusp's double cover (cf. the top-of-file
     * documentation in peripheral_curves.c). We don't know a priori
     * which sheet it is, but the other sheet will be empty, so we can
     * safely lift both sheets to the_cover. We lift the right_handed
     * (resp. left_handed) sheet at each vertex in the manifold to the
     * right_handed (resp. left_handed) sheet at the corresponding vertex
     * in each of the corresponding tetrahedra in the_cover to insure that
     * the peripheral curves obey the right hand rule relative to the
     * orientation of each cusp's double cover (cf. the top-of-file
     * documentation in peripheral_curves.c).
     */
}

```

```

* The curves on a Klein bottle cusp are already described as a single
* lift to the double cover (cf. the top-of-file ocumentation in
* peripheral_curves.c). So we copy the contents of the
*
*   old_tet->curve[][right_handed][][]
*   to new_tet[][0]->curve[][right_handed][][],
* and
*   old_tet->curve[][ left_handed][][]
*   to new_tet[][1]->curve[][ left_handed][][].
*
* The fields
*   new_tet[][1]->curve[][right_handed][][]
* and
*   new_tet[][0]->curve[][ left_handed][][]
*
* are left zero.
*
* Note that the curves match up across gluings (compare the convention
* for matching curves in peripheral_curves.c with the convention for
* assigning neighbors in set_neighbors_and_gluings() above).
*/

for (old_tet = manifold->tet_list_begin.next, i = 0;
    old_tet != &manifold->tet_list_end;
    old_tet = old_tet->next, i++)

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

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

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

                if (old_tet->cusp[v]->topology == torus_cusp)
                {
                    new_tet[i][0]->curve[c][right_handed][v][f] =
                    new_tet[i][1]->curve[c][right_handed][v][f] =
                    old_tet->curve[c][right_handed][v][f];

                    new_tet[i][0]->curve[c][ left_handed][v][f] =
                    new_tet[i][1]->curve[c][ left_handed][v][f] =
                    old_tet->curve[c][ left_handed][v][f];
                }
                else
                {
                    new_tet[i][0]->curve[c][right_handed][v][f] =
                    old_tet->curve[c][right_handed][v][f];

                    new_tet[i][1]->curve[c][ left_handed][v][f] =
                    old_tet->curve[c][ left_handed][v][f];
                }
            }
        }

static void lift_tet_shapes(
    Triangulation *manifold,
    TetPtrPair *new_tet)
{
    int i,
        j,
        k;
    Tetrahedron *old_tet;

    for (old_tet = manifold->tet_list_begin.next, i = 0;
        old_tet != &manifold->tet_list_end;
        old_tet = old_tet->next, i++)

        for (k = 0; k < 2; k++) /* k = complete, filled */

            if (old_tet->shape[k] != NULL)

                for (j = 0; j < 2; j++) /* j = which lift */
                {
                    new_tet[i][j]->shape[k] = NEW_STRUCT(TetShape);
                    *new_tet[i][j]->shape[k] = *old_tet->shape[k];
                }
            }
        }
}

```

```

        copy_shape_history(old_tet->shape_history[k], &new_tet[i][j]->
shape_history[k]);
    }
}

```

```

static void set_cusp_data(
    Triangulation *manifold,
    Triangulation *the_cover,
    TetPtrPair *new_tet)
{
    int i,
        j,
        cusp_count;
    IntPair *old_to_new;
    CuspTopology topology;
    Boolean is_complete;
    double m,
        l;
    Tetrahedron *old_tet;
    VertexIndex v;

    /*
     * Set up the correspondence between the old indices and the new ones.
     * Old tori will lift to two new tori, while old Klein bottles each
     * lift to a single new one.
     */

    old_to_new = NEW_ARRAY(manifold->num_cusps, IntPair);

    cusp_count = 0;

    for (i = 0; i < manifold->num_cusps; i++)
    {
        get_cusp_info(manifold, i, &topology,
            NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);

        if (topology == torus_cusp)
        {
            old_to_new[i][0] = cusp_count++;
            old_to_new[i][1] = cusp_count++;
        }
        else
        {
            old_to_new[i][0] = cusp_count++;
            old_to_new[i][1] = old_to_new[i][0];
        }
    }

    /*
     * Now assign the indices to the newly computed cusps.
     * This algorithm is highly redundant -- each index will be set
     * once for every ideal vertex incident to the cusp -- but the
     * inefficiency is insignificant. A cusp and its mate (i.e.
     * the two cusps in the double cover which project down to a given
     * cusp in the original manifold) may swap indices many times during
     * this redundant procedure, as their indices get overwritten.
     *
     * While we're here, let's set the topology of each Cusp to torus_cusp.
     */

    for (old_tet = manifold->tet_list_begin.next, i = 0;
        old_tet != &manifold->tet_list_end;
        old_tet = old_tet->next, i++)

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

            for (j = 0; j < 2; j++)
            {
                new_tet[i][j]->cusp[v]->index = old_to_new
                    [old_tet->cusp[v]->index] [j];

                new_tet[i][j]->cusp[v]->topology = torus_cusp;
            }
    }
}

```

```
    }

    /*
     * Set the is_complete, m and l fields of the new cusps.
     */

    for (i = 0; i < manifold->num_cusps; i++)
    {
        get_cusp_info(manifold, i, &topology, &is_complete, &m, &l,
                      NULL, NULL, NULL, NULL, NULL, NULL);

        if (topology == torus_cusp)
        {
            set_cusp_info(the_cover, old_to_new[i][0], is_complete, m, l);
            set_cusp_info(the_cover, old_to_new[i][1], is_complete, m, l);
        }
        else
        {
            set_cusp_info(the_cover, old_to_new[i][0], is_complete, m, l);
        }
    }

    /*
     * Free the local storage.
     */

    my_free(old_to_new);
}
```