

```

/*
 * cover.c
 *
 * construct_cover() constructs the n-sheeted cover of a base_manifold
 * defined by a transitive RepresentationIntoSn. Please see covers.h
 * for details.
 *
 * construct_cover() assumes the base_manifold's generators are present,
 * and correspond to the generators in the representation. The
 * representation is assumed to be transitive, so the cover is connected.
 * construct_cover() also assumes that each Cusp's basepoint_tet,
 * basepoint_vertex and basepoint_orientation are those which were
 * recorded when the fundamental group was computed. All these assumptions
 * will be true if the Triangulation has not been changed since the
 * RepresentationIntoSn was computed (and of course the UI takes
 * responsibility for recomputing the representations whenever the
 * triangulation changes).
 */

#include "kernel.h"

#define NAME_SUFFIX      " cover"

Triangulation *construct_cover(
    Triangulation      *base_manifold,
    RepresentationIntoSn *representation,
    int                 n)
{
    Triangulation      *covering_manifold;
    Tetrahedron        ***covering_tetrahedra,
                        *base_tetrahedron,
                        **lifts;
    int                 i,
                        j,
                        count,
                        face,
                        nbr_index,
                        gen_index,
                        sheet,
                        nbr_sheet,
                        covering_cusp_index,
                        the_gcd,
                        base_singularity,
                        covering_singularity,
                        index,
                        index0,
                        *primitive_permutation;
    VertexIndex         v;
    FaceIndex            f;
    MatrixInt22          *change_matrices;
    Orientation          handedness;
    Cusp                 *base_cusp,
                        *covering_cusp;

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

    /*
     * Fill in some of the global information.
     */
    if (base_manifold->name == NULL)
        uFatalError("construct_cover", "cover");
    covering_manifold->name = NEW_ARRAY(strlen(base_manifold->name) +
    strlen(NAME_SUFFIX) + 1, char);
    strcpy(covering_manifold->name, base_manifold->name);
    strcat(covering_manifold->name, NAME_SUFFIX);
    covering_manifold->num_tetrahedra = n * base_manifold->num_tetrahedra;
    covering_manifold->solution_type[complete] = base_manifold->solution_type[complete];
    covering_manifold->solution_type[ filled ] = base_manifold->solution_type[ filled ];

```

```

covering_manifold->orientability          = unknown_orientability;

/*
 * Make sure the base_manifold's Tetrahedra have indices.
 */
number_the_tetrahedra(base_manifold);

/*
 * Allocate a 2-dimensional array of Tetrahedra for the covering_manifold.
 */
covering_tetrahedra = NEW_ARRAY(base_manifold->num_tetrahedra, Tetrahedron **);
for (i = 0; i < base_manifold->num_tetrahedra; i++)
{
    covering_tetrahedra[i] = NEW_ARRAY(n, Tetrahedron *);
    for (j = 0; j < n; j++)
    {
        covering_tetrahedra[i][j] = NEW_STRUCT(Tetrahedron);
        initialize_tetrahedron(covering_tetrahedra[i][j]);
        INSERT_BEFORE(covering_tetrahedra[i][j], &covering_manifold->tet_list_end);
    }
}

/*
 * Copy information from the Tetrahedra of the base_manifold to
 * the Tetrahedra of the covering_manifold.
 */
for (base_tetrahedron = base_manifold->tet_list_begin.next, count = 0;
     base_tetrahedron != &base_manifold->tet_list_end;
     base_tetrahedron = base_tetrahedron->next, count++)
{
    if (base_tetrahedron->index != count)
        uFatalError("construct_cover", "cover");

    lifts = covering_tetrahedra[base_tetrahedron->index];

    /*
     * First figure out the neighbor fields.
     * This is the only part that uses the RepresentationIntoSn.
     */
    for (face = 0; face < 4; face++)
    {
        nbr_index = base_tetrahedron->neighbor[face]->index;
        gen_index = base_tetrahedron->generator_index[face];

        switch (base_tetrahedron->generator_status[face])
        {
            case not_a_generator:
                for (sheet = 0; sheet < n; sheet++)
                    lifts[sheet]->neighbor[face] =
                        covering_tetrahedra[nbr_index][sheet];
                break;

            case outbound_generator:
                for (sheet = 0; sheet < n; sheet++)
                    lifts[sheet]->neighbor[face] =
                        covering_tetrahedra[nbr_index][representation->image[gen_index]
[sheet]];
                break;

            case inbound_generator:
                for (nbr_sheet = 0; nbr_sheet < n; nbr_sheet++)
                    lifts[representation->image[gen_index][nbr_sheet]]->neighbor[face] =
                        covering_tetrahedra[nbr_index][nbr_sheet];
                break;

            default:
                uFatalError("construct_cover", "cover");
        }
    }

    /*
     * Lift the gluings straight from the base tetrahedron.
     */
}

```

```

    for (sheet = 0; sheet < n; sheet++)
        for (face = 0; face < 4; face++)
            lifts[sheet]->gluing[face] = base_tetrahedron->gluing[face];

    for (sheet = 0; sheet < n; sheet++)
        for (i = 0; i < 2; i++) /* complete, filled */
        {
            lifts[sheet]->shape[i] = NEW_STRUCT(TetShape);
            *lifts[sheet]->shape[i] = *base_tetrahedron->shape[i];
            copy_shape_history(base_tetrahedron->shape_history[i], &lifts[sheet]->
shape_history[i]);
        }
    }

/*
 * Create and orient the EdgeClasses.
 *
 * Note: These functions require only the tet->neighbor and
 *       tet->gluing fields.
 * Note: orient_edge_classes() assigns an arbitrary orientation
 *       to each edge class. If the manifold is orientable,
 *       orient() (cf. below) will replace each arbitrary orientation
 *       with the right_handed one.
 */
create_edge_classes(covering_manifold);
orient_edge_classes(covering_manifold);

/*
 * Create the covering_manifold's Cusps in the "natural" order. That is,
 * Cusps which project to the base_manifold's Cusp 0 should come first,
 * then Cusps which project to the base_manifold's Cusp 1, etc.
 */
error_check_for_create_cusps(covering_manifold); /* unnecessary */
covering_cusp_index = 0; /* running count */
for (base_cusp = base_manifold->cusp_list_begin.next;
    base_cusp != &base_manifold->cusp_list_end;
    base_cusp = base_cusp->next)
{
    /*
     * We don't know a priori how many Cusps in the covering_manifold
     * will project down to the given Cusp in the base_manifold.
     * So we examine each of the n lifts of the given ideal vertex,
     * and assign Cusps to those which haven't already been assigned
     * a Cusp at an earlier iteration of the loop. Let each new
     * Cusp's matching_cusp field store a pointer to the corresponding
     * cusp in the base_manifold. Note that we are working with
     * the basepoint_tets and basepoint_vertices relative to which
     * the fundamental_group() computed the meridians and longitudes;
     * we'll need to rely on this fact later on.
     */
    lifts = covering_tetrahedra[base_cusp->basepoint_tet->index];
    for (sheet = 0; sheet < n; sheet++)
        if (lifts[sheet]->cusp[base_cusp->basepoint_vertex] == NULL)
        {
            create_one_cusp(    covering_manifold,
                                lifts[sheet],
                                FALSE,
                                base_cusp->basepoint_vertex,
                                covering_cusp_index++);
            lifts[sheet]->cusp[base_cusp->basepoint_vertex]->matching_cusp = base_cusp;
        }
    }

/*
 * Install an arbitrary set of peripheral curves.
 *
 * Notes:
 *
 * (1) After the manifold is oriented we'll replace these
 *     arbitrary curves with a set in which the Dehn filling
 *     curve (if any) is the meridian, and the {meridian,
 *     longitude} pair adhere to the right hand rule.
 *
 * (2) peripheral_curves() will determine the CuspTopology of

```

```

    *      each Cusp, and write it into the cusp->topology field.
    */
peripheral_curves(covering_manifold);

/*
 * Count the total number of Cusps, and also the number
 * with torus and Klein bottle CuspTopology.
 */
count_cusps(covering_manifold);

/*
 * The Dehn filling curve on each cusp in the covering_manifold
 * should be a lift of the Dehn filling curve from the corresponding
 * cusp in the base_manifold. Note that this algorithm works
 * correctly even for orbifolds: for example, a singular circle
 * of order 6 in the base_manifold might be covered by singular
 * circles of order 2 and 3 in a 5-fold covering_manifold, and our
 * algorithm will provide the correct Dehn filling curve on each one.
 *
 * Conceptually we want to trace each Dehn filling curve in the
 * base_manifold, lifting it to the covering_manifold as we go.
 * Computationally it's simpler to compute the entire preimage
 * of the Dehn filling curve in the covering_manifold -- which gives
 * some number of parallel copies of the true Dehn filling curve --
 * and then use the RepresentationIntoSn to deduce the correct multiple.
 */

/*
 * Copy the covering_manifold's Dehn filling curves into
 * scratch_curve[0]. Note that the double_copy_on_tori option
 * is TRUE, so we'll get copies of the Dehn filling curves on
 * both the left_ and right_handed sheets of a torus cusp.
 */
copy_curves_to_scratch(covering_manifold, 0, TRUE);

/*
 * Copy the complete preimage of all the base_manifold's Dehn filling
 * curves into the scratch_curve[1][M] fields in the covering_manifold.
 * The scratch_curve[1][L] curves won't be used, so set them to zero.
 */
for ( base_tetrahedron = base_manifold->tet_list_begin.next, count = 0;
      base_tetrahedron != &base_manifold->tet_list_end;
      base_tetrahedron = base_tetrahedron->next, count++)
{
    if (base_tetrahedron->index != count)
        uFatalError("construct_cover", "cover");

    lifts = covering_tetrahedra[base_tetrahedron->index];

    for (sheet = 0; sheet < n; sheet++)
        for (handedness = 0; handedness < 2; handedness++) /* right_handed, left_handed */
            for (v = 0; v < 4; v++)
                for (f = 0; f < 4; f++)
                {
                    lifts[sheet]->scratch_curve[1][M][handedness][v][f] =
                        base_tetrahedron->cusp[v]->is_complete ?
                        0 :
                        /* representations.c has already checked that */
                        /* all Dehn filling coefficients are integers. */
                        ( (int)base_tetrahedron->cusp[v]->m * base_tetrahedron->curve
[M][handedness][v][f]
                        + (int)base_tetrahedron->cusp[v]->l * base_tetrahedron->curve
[L][handedness][v][f] );

                    lifts[sheet]->scratch_curve[1][L][handedness][v][f] = 0;
                }
}

/*
 * Compute the intersection numbers of the preimages of the Dehn
 * filling curves (in scratch[1]) with the covering_manifold's
 * peripheral curves (in scratch[0]).
 */

```

```
compute_intersection_numbers(covering_manifold);

/*
 * For each cusp in the covering_manifold, use the intersection
 * numbers to get the Dehn filling coefficients (relative to the
 * covering_manifold's peripheral curves) of the complete preimage
 * of the base_manifold's Dehn filling curve. Divide through by
 * the great common divisor to get the coefficients for a "primitive"
 * Dehn filling curve. The paragraph after this one will figure out
 * what multiple of the primitive curve is correct.
 */
for (covering_cusp = covering_manifold->cusp_list_begin.next;
     covering_cusp != &covering_manifold->cusp_list_end;
     covering_cusp = covering_cusp->next)
{
    covering_cusp->is_complete = covering_cusp->matching_cusp->is_complete;

    if (covering_cusp->is_complete == FALSE)
    {
        covering_cusp->m = covering_cusp->intersection_number[L][M];
        covering_cusp->l = -covering_cusp->intersection_number[M][M];

        the_gcd = gcd((long int)covering_cusp->m, (long int)covering_cusp->l);

        covering_cusp->m /= the_gcd;
        covering_cusp->l /= the_gcd;
    }
}

/*
 * Now's lets figure out the correct multiple of the primitive Dehn
 * filling curve. The following illustration shows a cross-section
 * of the universal cover of a tubular neighborhood of one of the
 * base_manifold's singular circles. In this example the singularity
 * has order 6.
 *
 *      \   d   /
 *       \   /
 *      e   /   c
 * -----/-----
 *       \   b
 *      f   \   a
 *       \   \
 *
 * The six labelled points are preimages of a single point in the
 * base_manifold. They may, however, be preimages of several different
 * points in the covering_manifold. The primitive_Dehn_image will
 * tell us which images are distinct. For example, if the
 * primitive_Dehn_image assigns a permutation which is a cycle of
 * order 3, then points a, b, and c will be preimages of distinct
 * points in the covering_manifold, while d, e and f are additional
 * preimages of those same three points. Note that the nature of
 * the covering may vary from one sheet of the covering_manifold
 * to another. For example, the permutation (01)(234) would define
 * a covering_manifold with two singular circles, one of which has
 * order 3 and maps nearby points 2-to-1 onto the base_manifold,
 * and the other of which has order 2 and maps nearby points 3-to-1
 * onto the base_manifold.
 *
 * In practical terms, we have two questions to answer:
 *
 * (1) What is the order of the singularity in the base_manifold?
 *
 * (2) What is the smallest power of the primitive Dehn permutation
 * which takes a given sheet of the covering back to itself?
 *
 * The answer to question (1) divided by the answer to question (2)
 * will tell us the order of the singularity in the covering_manifold.
 */
for (covering_cusp = covering_manifold->cusp_list_begin.next;
     covering_cusp != &covering_manifold->cusp_list_end;
     covering_cusp = covering_cusp->next)
```

```

if (covering_cusp->is_complete == FALSE)
{
    /*
     * Recall that we stashed away a pointer to the base_cusp.
     */
    base_cusp = covering_cusp->matching_cusp;

    /*
     * The answer to question (1) is easy.
     */
    base_singularity = gcd((long int)base_cusp->m, (long int)base_cusp->l);

    /*
     * The basepoint used for defining the meridian and longitude
     * (in the base_manifold's cusp) has n lifts to the covering
     * manifold, implicitly labelled by the indices 0, 1, ..., n-1.
     * Find the index of a lift which belongs to the present
     * covering_cusp.
     */
    lifts = covering_tetrahedra[base_cusp->basepoint_tet->index];
    v = base_cusp->basepoint_vertex;
    for (index = 0; index < n; index++)
        if (lifts[index]->cusp[v] == covering_cusp)
            break;
    if (index == n)
        uFatalError("construct_cover", "cover");

    /*
     * How many times must we apply the primitive Dehn permutation
     * to the index before it returns to its original value?
     * This answers question (2).
     */
    primitive_permutation = representation->primitive_Dehn_image[base_cusp->index];
    index0 = index;
    count = 0;
    do
    {
        index = primitive_permutation[index];
        count++;
    }
    while (index != index0);

    /*
     * Divide answer (1) by answer (2) to get the order of
     * the singularity in the covering_manifold.
     */
    if (base_singularity % count != 0)
        uFatalError("construct_cover", "cover");
    covering_singularity = base_singularity / count;

    /*
     * Multiply the primitive Dehn filling curve
     * by the covering_singularity.
     */
    covering_cusp->m *= covering_singularity;
    covering_cusp->l *= covering_singularity;
}

/*
 * Free the 2-dimensional array used to keep track of the Tetrahedra.
 */
for (i = 0; i < base_manifold->num_tetrahedra; i++)
    my_free(covering_tetrahedra[i]);
my_free(covering_tetrahedra);

/*
 * Attempt to orient the manifold. Note that orient() may change
 * the vertex/face indexing, so we call it after we've lifted all
 * the information we need from the base_manifold.
 */
orient(covering_manifold);

/*

```

```

    * If the covering_manifold is oriented, orient() moves all
    * peripheral curves to the right_handed sheets. Thus some
    * {meridian, longitude} pairs may no longer adhere to the
    * right-hand rule. Correct them, and adjust the Dehn filling
    * coefficients accordingly.
    */
if (covering_manifold->orientability == oriented_manifold)
    fix_peripheral_orientations(covering_manifold);

/*
 * Normally the holonomies and cusp shapes are computed as part of
 * the computation of the hyperbolic structure. But we've lifted
 * the hyperbolic structure directly from the base_manifold. So
 * we compute the holonomies and cusp shapes explicitly.
 */
compute_the_holonomies(covering_manifold, ultimate);
compute_the_holonomies(covering_manifold, penultimate);
compute_cusp_shapes(covering_manifold, initial);
compute_cusp_shapes(covering_manifold, current);

/*
 * Lift the Chern-Simons value (if any) from the base manifold,
 * and use it to compute the fudge factor.
 */
covering_manifold->CS_value_is_known = base_manifold->CS_value_is_known;
if (base_manifold->CS_value_is_known)
{
    covering_manifold->CS_value[ultimate] = n * base_manifold->CS_value[ultimate];
    covering_manifold->CS_value[penultimate] = n * base_manifold->CS_value[penultimate]
;
}
compute_CS_fudge_from_value(covering_manifold);

/*
 * If the covering_manifold is hyperbolic, install a set of shortest
 * basis curves on all cusps. (The shortest curves on filled cusps
 * will be replaced below by curves for which the Dehn filling curve
 * is a multiple of the meridian.)
 */
switch (covering_manifold->solution_type[complete])
{
    case geometric_solution:
    case nongeometric_solution:
        install_shortest_bases(covering_manifold);
        break;
}

/*
 * On filled cusps, install a basis in which the Dehn filling curves
 * are (perhaps multiples of) a meridian.
 */
change_matrices = NEW_ARRAY(covering_manifold->num_cusps, MatrixInt22);
for (i = 0; i < covering_manifold->num_cusps; i++)
    current_curve_basis(covering_manifold, i, change_matrices[i]);
change_peripheral_curves(covering_manifold, change_matrices);
my_free(change_matrices);

return covering_manifold;
}

```