

```

/*
 * isometry_cusped.c
 *
 * This file provides the function
 *
 *      FuncResult compute_cusped_isometries(
 *          Triangulation *manifold0,
 *          Triangulation *manifold1,
 *          IsometryList **isometry_list,
 *          IsometryList **isometry_list_of_links);
 *
 * compute_cusped_isometries() computes a list of all Isometries from
 * manifold0 to manifold1 and stores its address in *isometry_list.
 * If isometry_list_of_links is not NULL, it copies those Isometries
 * which extend to Isometries of the associated links (i.e. those
 * which take meridians to plus-or-minus meridians) onto a separate
 * list, and stores its address in *isometry_list_of_links.
 *
 * compute_cusped_isometries() returns func_OK if it can get a canonical
 * decomposition for both manifold0 and manifold1, and func_failed if it
 * can't (because one or both lacks a hyperbolic structure).
 *
 * compute_cusped_isometries() allocates the IsometryList data structures.
 * You should declare isometry_list as IsometryList *isometry_list, and pass
 * &isometry_list to compute_cusped_isometries(), and similarly for
 * isometry_list_of_links.
 */

#include "kernel.h"

static FuncResult attempt_isometry(Triangulation *manifold0, Tetrahedron *tet0,
    Tetrahedron *tet1, Permutation map0);
static Boolean is_isometry_plausible(Tetrahedron *initial_tet0, Tetrahedron *
    initial_tet1, Permutation initial_map);
static void copy_isometry(Triangulation *manifold0, Triangulation *manifold1,
    Isometry **new_isometry);
static void compute_cusp_action(Triangulation *manifold0, Triangulation *manifold1,
    Isometry *isometry);
static void compute_cusp_image(Triangulation *manifold0, Isometry *isometry);
static void compute_cusp_map(Triangulation *manifold0, Triangulation *manifold1,
    Isometry *isometry);
static void copy_images_to_scratch(Triangulation *manifold, int which_set, Boolean
    double_copy_on_tori);
static Boolean does_isometry_extend_to_link(Isometry *isometry);
static void make_isometry_array(IsometryList *isometry_list, Isometry *
    the_linked_list);
static void find_isometries_which_extend(IsometryList *isometry_list, IsometryList
    **isometry_list_of_links);

FuncResult compute_cusped_isometries(
    Triangulation *manifold0,
    Triangulation *manifold1,
    IsometryList **isometry_list,
    IsometryList **isometry_list_of_links)
{
    Triangulation *copy_of_manifold0,
    *copy_of_manifold1;
    Isometry *partial_isometry_list,
    *new_isometry;
    Tetrahedron *tet0,
    *tet1;
    int i;

    /*
     * If manifold0 != manifold1 we are computing the isometries from one
     * manifold to another. We begin by making a copy of each manifold
     * and converting it to the canonical retriangulation of the
     * canonical cell decomposition.
     */
    if (manifold0 != manifold1)
    {
        /*
         * Make copies of the manifolds.

```

```

    */
    copy_triangulation(manifold0, &copy_of_manifold0);
    copy_triangulation(manifold1, &copy_of_manifold1);

    /*
     * Find the canonical triangulations of the copies.
     */
    if (canonize(copy_of_manifold0) == func_failed
        || canonize(copy_of_manifold1) == func_failed)
    {
        free_triangulation(copy_of_manifold0);
        free_triangulation(copy_of_manifold1);
        *isometry_list = NULL;
        *isometry_list_of_links = NULL;
        return func_failed;
    }
}

/*
 * If manifold0 == manifold1 we are computing the symmetries from
 * a manifold to itself. In this case we find the canonical
 * retriangulation of the canonical cell decomposition before making
 * the second copy. This serves two purposes:
 *
 * (1) It reduces the run time, because we avoid a redundant call
 *     to canonize().
 *
 * (2) It insures that the Tetrahedra will be listed in the same
 *     order in the two manifolds. This makes it easy for the
 *     symmetry group program to recognize the identity symmetry.
 *     (Perhaps you are curious as to how the Tetrahedra could
 *     possibly fail to be listed in the same order. In rare
 *     cases canonize() must do some random retriangulation.
 *     If this is done to two identical copies of a Triangulation
 *     the final canonical retriangulations will of course be
 *     combinatorially the same, but the Tetrahedra might be listed
 *     in different orders and numbered differently.)
 */
else { /* manifold0 == manifold1 */

    /*
     * Make one copy of the manifold.
     */
    copy_triangulation(manifold0, &copy_of_manifold0);

    /*
     * Find the canonical retriangulation.
     */
    if (canonize(copy_of_manifold0) == func_failed)
    {
        free_triangulation(copy_of_manifold0);
        *isometry_list = NULL;
        *isometry_list_of_links = NULL;
        return func_failed;
    }

    /*
     * Make a second copy of the canonical retriangulation.
     */
    copy_triangulation(copy_of_manifold0, &copy_of_manifold1);
}

/*
 * Allocate space for the IsometryList and initialize it.
 */
*isometry_list = NEW_STRUCT(IsometryList);
(*isometry_list)->num_isometries = 0;
(*isometry_list)->isometry = NULL;

/*
 * If isometry_list_of_links is not NULL, allocate and
 * initialize *isometry_list_of_links as well.
 */
if (isometry_list_of_links != NULL)

```

```

{
    *isometry_list_of_links = NEW_STRUCT(IsometryList);
    (*isometry_list_of_links)->num_isometries = 0;
    (*isometry_list_of_links)->isometry = NULL;
}

/*
 * If the two manifolds don't have the same number of
 * Tetrahedra, then there can't possibly be any isometries
 * between the two. We just initialized the IsometryList(s)
 * to be empty, so if there aren't any isometries, we're
 * ready free the copies of the manifolds and return.
 */
if (copy_of_manifold0->num_tetrahedra != copy_of_manifold1->num_tetrahedra)
{
    free_triangulation(copy_of_manifold0);
    free_triangulation(copy_of_manifold1);
    return func_OK;
}

/*
 * partial_isometry_list will keep a linked list of the
 * Isometries we discover. When we're done we'll allocate
 * the array (*isometry_list)->isometry and copy the
 * addresses of the Isometries into it. For now, we
 * initialize partial_isometry_list to NULL to show that
 * the linked list is empty.
 */
partial_isometry_list = NULL;

/*
 * Assign indices to the Tetrahedra in each manifold.
 */
number_the_tetrahedra(copy_of_manifold0);
number_the_tetrahedra(copy_of_manifold1);

/*
 * Try mapping an arbitrary but fixed Tetrahedron of
 * manifold0 ("tet0") to each Tetrahedron of manifold1
 * in turn, with each possible Permutation. See which
 * of these maps extends to a global isometry. We're
 * guaranteed to find all possible isometries this way
 * (and typically quite a lot of other garbage as well).
 */

/*
 * Let tet0 be the first Tetrahedron on manifold0's list.
 */
tet0 = copy_of_manifold0->tet_list_begin.next;

/*
 * Consider each Tetrahedron in manifold1.
 */
for (tet1 = copy_of_manifold1->tet_list_begin.next;
     tet1 != &copy_of_manifold1->tet_list_end;
     tet1 = tet1->next)

    /*
     * Consider each of the 24 possible ways to map tet0 to tet1.
     */
    for (i = 0; i < 24; i++)

        /*
         * Does mapping tet0 to tet1 via permutation_by_index[i]
         * define an isometry?
         */
        if (attempt_isometry(copy_of_manifold0, tet0, tet1, permutation_by_index[i]) ==
func_OK)
        {
            /*
             * Copy the isometry to an Isometry data structure . . .
             */
            copy_isometry(copy_of_manifold0, copy_of_manifold1, &new_isometry);

```

```

        /*
         * . . . add it to the partial_isometry_list . . .
         */
        new_isometry->next = partial_isometry_list;
        partial_isometry_list = new_isometry;

        /*
         * . . . and increment the count.
         */
        (*isometry_list)->num_isometries++;
    }

    /*
     * If some Isometries were found, allocate space for the
     * array (*isometry_list)->isometry and write the addresses
     * of the Isometries into it.
     */
    make_isometry_array(*isometry_list, partial_isometry_list);

    /*
     * If isometry_list_of_links is not NULL, make a copy of
     * those Isometries which extend to the associated link
     * (i.e. those which take meridians to meridians).
     */
    find_isometries_which_extend(*isometry_list, isometry_list_of_links);

    /*
     * Discard the copies of the manifolds.
     */
    free_triangulation(copy_of_manifold0);
    free_triangulation(copy_of_manifold1);

    return func_OK;
}

/*
 * attempt_isometry() checks whether sending tet0 (of manifold0)
 * to tet1 (of manifold1, which may or may equal manifold0)
 * defines an isometry between the two manifolds. If it does,
 * it leaves the isometry in the "image" and "map" fields of
 * the Tetrahedra of manifold0 (see the documentation at the top
 * of isometry.h for details), and returns func_OK. If it doesn't
 * define an isometry, it leaves garbage in the "image" and "map"
 * fields and returns func_failed.
 *
 * Technical note: attempt_isometry() could be written using a simple
 * recursive function to set the image and map fields, but I'm trying
 * to avoid recursions, because on some machines (e.g. older Macs)
 * a stack-heap collision is a possibility if one has a deep recursion
 * with a lot of nonstatic local variables, whereas in the future static
 * local variables are likely to cause problems in a multithreaded
 * environment. So . . . better to avoid the recursion. The actual
 * algorithm is documented in the following code.
 */

static FuncResult attempt_isometry(
    Triangulation *manifold0,
    Tetrahedron *initial_tet0,
    Tetrahedron *initial_tet1,
    Permutation initial_map)
{
    Tetrahedron *tet0,
                *tet1,
                *nbr0,
                *nbr1,
                **queue;
    int first,
        last;
    FaceIndex face0,
             face1;
    Permutation gluing0,
               gluing1,
               nbr0_map;

```

```

/*
 * initial_tet1 and initial_map are arbitrary, so
 * the vast majority of calls to attempt_isometry()
 * will fail. Therefore it's worth including a quick
 * plausibility check at the beginning, to make the
 * algorithm run faster.
 */
if (is_isometry_plausible(initial_tet0, initial_tet1, initial_map) == FALSE)
    return func_failed;

/*
 * Initialize all the image fields of manifold0
 * to NULL to show they haven't been set.
 */
for (tet0 = manifold0->tet_list_begin.next;
     tet0 != &manifold0->tet_list_end;
     tet0 = tet0->next)
    tet0->image = NULL;

/*
 * Allocate space for a queue which is large enough
 * to hold pointers to all the Tetrahedra in manifold0.
 */
queue = NEW_ARRAY(manifold0->num_tetrahedra, Tetrahedron *);

/*
 * At all times, the Tetrahedra on the queue will be those which
 *
 *      (1) have set their image and map fields, but
 *
 *      (2) have not checked their neighbors.
 */

/*
 * Set the image and map fields for initial_tet0.
 */
initial_tet0->image = initial_tet1;
initial_tet0->map = initial_map;

/*
 * Put initial_tet0 on the queue.
 */
first = 0;
last = 0;
queue[first] = initial_tet0;

/*
 * While there are Tetrahedra on the queue . . .
 */
while (last >= first)
{
    /*
     * Pull the first Tetrahedron off the queue and call it tet0.
     */
    tet0 = queue[first++];

    /*
     * tet0 maps to some Tetrahedron tet1 in manifold1.
     */
    tet1 = tet0->image;

    /*
     * For each face of tet0 . . .
     */
    for (face0 = 0; face0 < 4; face0++)
    {
        /*
         * Let nbr0 be the Tetrahedron which meets tet0 at face0.
         */
        nbr0 = tet0->neighbor[face0];

        /*
         * tet0->map takes face0 of tet0 to facel of tet1.

```

```

    */
    facel = EVALUATE(tet0->map, face0);

    /*
     * Let nbr1 be the Tetrahedron which meets tet1 at facel.
     */
    nbr1 = tet1->neighbor[facel];

    /*
     * Let gluing0 be the gluing which identifies face0 of
     * tet0 to nbr0, and similarly for gluing1.
     */
    gluing0 = tet0->gluing[face0];
    gluing1 = tet1->gluing[facel];

    /*
     *
     *          gluing0
     *          ----->
     *          tet0      nbr0
     *          |         |
     *  tet0->map |         | nbr0->map
     *          |         |
     *          V   gluing1   V
     *          tet1 -----> nbr1
     *
     * We want to ensure that tet1 and nbr1 enjoy the same
     * relationship to each other in manifold1 that tet0 and
     * nbr0 do in manifold0. The conditions
     *
     *          nbr0->image == nbr1
     * and
     *          nbr0->map == gluing1 o tet0->map o gluing0^-1
     *
     * are necessary and sufficient to insure that we have a
     * combinatorial equivalence between the Triangulations.
     * (The proof relies on the fact that we've already checked
     * (near the beginning of compute_cusped_isometries() above)
     * that the Triangulations have the same number of Tetrahedra;
     * otherwise one Triangulation could be a (possibly branched)
     * covering of the other.)
     */

    /*
     * Compute the required value for nbr0->map.
     */
    nbr0_map = compose_permutations(
        compose_permutations(
            gluing1,
            tet0->map
        ),
        inverse_permutation[gluing0]
    );

    /*
     * If nbr0->image and nbr0->map have already been set,
     * check that they satisfy the above conditions.
     */
    if (nbr0->image != NULL)
    {
        if (nbr0->image != nbr1 || nbr0->map != nbr0_map)
        {
            /*
             * This isn't an isometry.
             */
            my_free(queue);
            return func_failed;
        }
    }
    /*
     * else . . .
     * nbr0->image and nbr0->map haven't been set.
     * Set them, and put nbr0 on the queue.
     */
    else
    {

```

```

        nbr0->image = nbr1;
        nbr0->map    = nbr0_map;
        queue[++last] = nbr0;
    }
}

/*
 * A quick error check.
 * Is it plausible that each Tetrahedron
 * has been on the queue exactly once?
 */
if (first != manifold0->num_tetrahedra
    || last != manifold0->num_tetrahedra - 1)
    uFatalError("attempt_isometry", "isometry");

/*
 * Free the queue, and report success.
 */
my_free(queue);
return func_OK;
}

static Boolean is_isometry_plausible(
    Tetrahedron    *initial_tet0,
    Tetrahedron    *initial_tet1,
    Permutation     initial_map)
{
    /*
     * To check whether an Isometry taking initial_tet0 to
     * initial_tet1 via initial_map is even plausible, let's
     * check whether their EdgeClass orders match up.
     */

    int i,
        j;

    for (i = 0; i < 4; i++)
        for (j = i + 1; j < 4; j++)
            if (initial_tet0->edge_class[edge_between_vertices[i][j]]->order
                != initial_tet1->edge_class[edge_between_vertices[EVALUATE(initial_map, i)]
                    [EVALUATE(initial_map, j)]]->order)
                return FALSE;

    return TRUE;
}

/*
 * copy_isometry() assumes the "image" and "map" fields of
 * manifold0 describe an isometry to a (not necessarily
 * distinct) manifold1. It also assumes that the Tetrahedra
 * in both manifolds have been numbered as described in
 * symmetry.h. It allocates an Isometry data structure,
 * writes the isometry into it, and sets *new_isometry to
 * point to it.
 */

static void copy_isometry(
    Triangulation    *manifold0,
    Triangulation    *manifold1,
    Isometry         **new_isometry)
{
    Tetrahedron *tet0;
    int i;

    /*
     * Allocate the Isometry.
     */

```

```

    *new_isometry          = NEW_STRUCT(Isometry);
    (*new_isometry)->tet_image = NEW_ARRAY(manifold0->num_tetrahedra, int);
    (*new_isometry)->tet_map   = NEW_ARRAY(manifold0->num_tetrahedra, Permutation);
    (*new_isometry)->cuspid_image = NEW_ARRAY(manifold0->num_cusps, int);
    (*new_isometry)->cuspid_map  = NEW_ARRAY(manifold0->num_cusps, MatrixInt22);

    /*
     * Set the num_tetrahedra and num_cusps fields.
     */

    (*new_isometry)->num_tetrahedra = manifold0->num_tetrahedra;
    (*new_isometry)->num_cusps      = manifold0->num_cusps;

    /*
     * Copy the isometry from the Triangulation
     * to the Isometry data structure.
     */

    for (tet0 = manifold0->tet_list_begin.next, i = 0;
         tet0 != &manifold0->tet_list_end;
         tet0 = tet0->next, i++)
    {
        (*new_isometry)->tet_image[i] = tet0->image->index;
        (*new_isometry)->tet_map[i]   = tet0->map;
    }

    /*
     * How does this Isometry act on the Cusps?
     */

    compute_cusp_action(manifold0, manifold1, *new_isometry);

    /*
     * We don't use the "next" field.
     */

    (*new_isometry)->next = NULL;
}

/*
 * Given a Triangulation *manifold0 whose "image" and "map" fields
 * describe an isometry to some (not necessarily distinct) manifold,
 * compute_cusp_action() computes the action on the Cusps. Only real
 * cusps are considered -- finite vertices are ignored.
 */

static void compute_cusp_action(
    Triangulation *manifold0,
    Triangulation *manifold1,
    Isometry *isometry)
{
    compute_cusp_image(manifold0, isometry);
    compute_cusp_map(manifold0, manifold1, isometry);
    isometry->extends_to_link = does_isometry_extend_to_link(isometry);
}

static void compute_cusp_image(
    Triangulation *manifold0,
    Isometry *isometry)
{
    Tetrahedron *tet;
    VertexIndex v;

    /*
     * Examine each vertex of each Tetrahedron to see which
     * cusp is being taken to which.
     *
     * There's a tremendous amount of redundancy here -- each
     * cusp image is set over and over -- but it hardly matters.
     *
     * Ignore negatively indexed Cusps -- they're finite vertices.
     */

```



```

for (tet = manifold0->tet_list_begin.next;
    tet != &manifold0->tet_list_end;
    tet = tet->next)

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

        if (tet->cuspid[v]->index >= 0)

            isometry->cuspid_image[tet->cuspid[v]->index]
                = tet->image->cuspid[EVALUATE(tet->map, v)]->index;
}

static void compute_cuspid_map(
    Triangulation *manifold0,
    Triangulation *manifold1,
    Isometry *isometry)
{
    Tetrahedron *tet;
    VertexIndex v;
    int i;

    /*
     * Copy the manifold1's peripheral curves into
     * scratch_curves[0], and copy the images of manifold0's
     * peripheral curves into scratch_curves[1].
     *
     * When the manifold is orientable and the Isometry is
     * orientation-reversing, and sometimes when the manifold
     * is nonorientable, the images of the peripheral curves
     * of a torus will lie on the "wrong" sheet of the Cuspid's
     * orientation double cover. (See peripheral_curves.c for
     * background material.) Therefore we copy the images of
     * the peripheral curves of torus cusps to both sheets of
     * the orientation double cover, to guarantee that the
     * intersection numbers come out right.
     */

    copy_curves_to_scratch(manifold1, 0, FALSE);
    copy_images_to_scratch(manifold0, 1, TRUE);

    /*
     * Compute the intersection numbers of the images of manifold0's
     * peripheral curves with manifold1's peripheral curves..
     */

    compute_intersection_numbers(manifold1);

    /*
     * Now extract the cuspid_maps from the linking numbers.
     *
     * There's a lot of redundancy in this loop, but a trivial
     * computation so the redundancy hardly matters.
     *
     * Ignore negatively indexed Cusps -- they're finite vertices.
     */

    for (tet = manifold0->tet_list_begin.next;
        tet != &manifold0->tet_list_end;
        tet = tet->next)

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

            if (tet->cuspid[v]->index >= 0)

                for (i = 0; i < 2; i++) /* i = M, L */
                {
                    isometry->cuspid_map[tet->cuspid[v]->index][M][i]
                        = + tet->image->cuspid[EVALUATE(tet->map, v)]->intersection_number[L]
[i];

                    isometry->cuspid_map[tet->cuspid[v]->index][L][i]
                        = - tet->image->cuspid[EVALUATE(tet->map, v)]->intersection_number[M]

```

```

    [i];
    }
}

static void copy_images_to_scratch(
    Triangulation *manifold0,
    int which_set,
    Boolean double_copy_on_tori)
{
    Tetrahedron *tet;
    int i,
        j,
        jj,
        k,
        kk,
        l,
        ll;

    /*
     * This function is modelled on copy_curves_to_scratch()
     * in intersection_numbers.c.
     */

    /*
     * Note that even though we are passed manifold0 as an
     * explicit function parameter, we are ultimately writing
     * the image curves in manifold1.
     */

    for (tet = manifold0->tet_list_begin.next;
         tet != &manifold0->tet_list_end;
         tet = tet->next)

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

            for (k = 0; k < 4; k++)
            {
                kk = EVALUATE(tet->map, k);

                for (l = 0; l < 4; l++)
                {
                    ll = EVALUATE(tet->map, l);

                    if (tet->cuspid[k]->topology == torus_cuspid
                        && double_copy_on_tori == TRUE)

                        tet->image->scratch_curve[which_set][i][right_handed][kk][ll] =
                        tet->image->scratch_curve[which_set][i][left_handed][kk][ll] =
                        tet->curve[i][right_handed][k][l]
                        + tet->curve[i][left_handed][k][l];

                    else
                        /*
                         * tet->cuspid[k]->topology == Klein_cuspid
                         * || double_copy_on_tori == FALSE
                         */

                        for (j = 0; j < 2; j++)
                        {
                            /*
                             * parities can be tricky.
                             *
                             * When discussing gluings from a face of one Tetrahedron
                             * to a face of another, an even parity corresponds to an
                             * orientation-reversing gluing.
                             *
                             * When discussing a map from one Tetrahedron onto
                             * another, an even parity is an orientation-preserving
                             * map.
                             */

                            /*
                             * If tet->map has even parity (i.e. if it's an

```

```

        * orientation-preserving map) it will send the
        * right-handed vertex cross section to a right-
        * handed image.
        *
        * If tet->map has odd parity (i.e. if it's an
        * orientation-reversing map) it will send the
        * right-handed vertex cross section to a left-
        * handed image.
        */
        jj = (parity[tet->map] == 0) ? j : !j;

        tet->image->scratch_curve[which_set][i][jj][kk][ll]
            = tet->curve[i][j][k][l];
    }
}

static Boolean does_isometry_extend_to_link(
    Isometry *isometry)
{
    /*
     * This function assumes the cusp_maps have been
     * computed, and checks whether they all take
     * meridians to meridians.
     */

    int i;

    for (i = 0; i < isometry->num_cusps; i++)
        if (isometry->cusp_map[i][L][M] != 0)
            return FALSE;

    return TRUE;
}

static void make_isometry_array(
    IsometryList *isometry_list,
    Isometry *the_linked_list)
{
    int i;
    Isometry *an_isometry;

    if (isometry_list->num_isometries == 0)
        isometry_list->isometry = NULL;

    else
    {
        isometry_list->isometry = NEW_ARRAY(isometry_list->num_isometries, Isometry *);

        for (an_isometry = the_linked_list, i = 0;
             an_isometry != NULL && i < isometry_list->num_isometries;
             an_isometry = an_isometry->next, i++)

            isometry_list->isometry[i] = an_isometry;

        /*
         * A quick error check.
         */
        if (an_isometry != NULL || i != isometry_list->num_isometries)
            uFatalError("make_isometry_array", "isometry");
    }
}

static void find_isometries_which_extend(
    IsometryList *isometry_list,
    IsometryList **isometry_list_of_links)
{

```

```

Isometry    *original,
            *copy;
int          i,
            j,
            k,
            l,
            count;

/*
 * If the isometry_list_of_links isn't needed, don't compute it.
 */

if (isometry_list_of_links == NULL)
    return;

/*
 * The function compute_cusped_isometries() (which calls this function)
 * has already allocated space for the new IsometryList.
 * (It has also initialized the num_isometries and isometry fields,
 * but we reinitialize them here just for good form.)
 */

/*
 * How many Isometries extend to the link?
 */

(*isometry_list_of_links)->num_isometries = 0; /* intentionally redundant */

for (i = 0; i < isometry_list->num_isometries; i++)
    if (isometry_list->isometry[i]->extends_to_link == TRUE)
        (*isometry_list_of_links)->num_isometries++;

/*
 * If there are no Isometries which extend, set
 * (*isometry_list_of_links)->isometry to NULL and return.
 */

if ((*isometry_list_of_links)->num_isometries == 0)
{
    (*isometry_list_of_links)->isometry = NULL; /* intentionally redundant */

    return;
}

/*
 * Allocate space for the Isometries which extend to
 * Isometries of the associated links, and copy them in.
 */

(*isometry_list_of_links)->isometry = NEW_ARRAY((*isometry_list_of_links)->
num_isometries, Isometry *);

for (i = 0, count = 0; i < isometry_list->num_isometries; i++)
    if (isometry_list->isometry[i]->extends_to_link == TRUE)
    {
        (*isometry_list_of_links)->isometry[count] = NEW_STRUCT(Isometry);

        original    = isometry_list->isometry[i];
        copy        = (*isometry_list_of_links)->isometry[count];

        copy->num_tetrahedra    = original->num_tetrahedra;
        copy->num_cusps        = original->num_cusps;

        copy->tet_image = NEW_ARRAY(copy->num_tetrahedra, int);
        copy->tet_map    = NEW_ARRAY(copy->num_tetrahedra, Permutation);
        for (j = 0; j < copy->num_tetrahedra; j++)
        {
            copy->tet_image[j] = original->tet_image[j];
            copy->tet_map [j] = original->tet_map [j];
        }
    }

```

```

        copy->cuspid_image    = NEW_ARRAY(copy->num_cusps, int);
        copy->cuspid_map      = NEW_ARRAY(copy->num_cusps, MatrixInt22);
        for (j = 0; j < copy->num_cusps; j++)
        {
            copy->cuspid_image[j] = original->cuspid_image[j];
            for (k = 0; k < 2; k++)
                for (l = 0; l < 2; l++)
                    copy->cuspid_map[j][k][l] = original->cuspid_map[j][k][l];
        }

        copy->extends_to_link = original->extends_to_link;

        count++;
    }
}

```

```

Boolean same_triangulation(
    Triangulation *manifold0,
    Triangulation *manifold1)
{
    /*
     * Check whether manifold0 and manifold1 have combinatorially
     * equivalent triangulations (ignoring Dehn fillings).
     *
     * This function is most useful when manifold0 and manifold1 are
     * canonical retriangulations of the canonical cell decomposition,
     * but it can be applied to any Triangulations. For normal use,
     * compute_isometries(manifold0, manifold1, &are_isometric, NULL, NULL)
     * is more flexible and easier to use. The present function is
     * intended for batch use (in particular for processing Jim Hoste's
     * knot tabulation), when it's important to avoid the overhead of
     * recomputing canonical triangulations.
     */

    /*
     * Imitate the algorithm in compute_cusped_isometries().
     */

    Tetrahedron *tet0,
                *tet1;
    int          i;

    /*
     * If the triangulations contain different numbers of tetrahedra,
     * they can't possibly be equivalent.
     */
    if (manifold0->num_tetrahedra != manifold1->num_tetrahedra)
        return FALSE;

    /*
     * Try mapping an arbitrary but fixed Tetrahedron of manifold0
     * ("tet0") to each Tetrahedron of manifold1 in turn, with each
     * possible Permutation. See whether any of these maps extends
     * to a global isometry.
     */

    /*
     * Let tet0 be the first Tetrahedron on manifold0's list.
     */
    tet0 = manifold0->tet_list_begin.next;

    /*
     * Consider each Tetrahedron in manifold1.
     */
    for (tet1 = manifold1->tet_list_begin.next;
         tet1 != &manifold1->tet_list_end;
         tet1 = tet1->next)
    /*
     * Consider each of the 24 possible ways to map tet0 to tet1.
     */
    for (i = 0; i < 24; i++)
        /*
         * Does mapping tet0 to tet1 via permutation_by_index[i]

```

```
        *   define an isometry?
        */
    if (attempt_isometry(manifold0, tet0, tet1, permutation_by_index[i]) ==
func_OK)
        /*
         *   The triangulations are equivalent.
         */
        return TRUE;

    /*
     *   The triangulations are different.
     */
    return FALSE;
}
```