

```

/*
 * isometry.c
 *
 * This file provides the functions
 *
 *      FuncResult compute_isometries(
 *          Triangulation    *manifold0,
 *          Triangulation    *manifold1,
 *          Boolean          *are_isometric,
 *          IsometryList     **isometry_list,
 *          IsometryList     **isometry_list_of_links);
 *
 *      Checks whether manifold0 and manifold1 are isometric
 *      (taking into account the Dehn fillings) and sets
 *      the flag *are_isometric accordingly. If manifold0 and
 *      manifold1 are cusped manifolds, sets *isometry_list and
 *      *isometry_list_of_links as in compute_cusped_isometries()
 *      in isometry_cusped.c. You may pass NULL for isometry_list
 *      and isometry_list_of_links if you aren't interested in the
 *      IsometryLists. But if you do take them, be sure to free
 *      them with free_isometry_list() when you're done.
 *
 *      compute_isometries() results are 100% rigorous.
 *      It reports an isometry only when it has found identical
 *      Triangulations (with identical Dehn fillings in the case
 *      of closed manifolds). It reports nonisometry only when
 *      some discrete invariant (number of cusps or first homology
 *      group) distinguishes the manifolds. [96/12/6 This doesn't
 *      seem to be the case. It looks like the code is willing
 *      to report a nonisometry when canonical cell decompositions
 *      fail to match. That's plenty rigorous, but the canonical
 *      decomposition does rely on the accuracy of the hyperbolic
 *      structure.]
 *
 *      Returns
 *
 *          func_OK          if all goes well,
 *
 *          func_bad_input   if some Dehn filling coefficients are not
 *                          relatively prime integers,
 *
 *          func_failed      if it can't decide.
 *
 *      int isometry_list_size(IsometryList *isometry_list);
 *
 *      Returns the number of Isometries in the IsometryList.
 *
 *      int isometry_list_num_cusps(IsometryList *isometry_list);
 *
 *      Returns the number of cusps in each of the underlying
 *      manifolds. If the IsometryList is empty (as would be the
 *      case when the underlying manifolds have different numbers
 *      of cusps), then isometry_list_num_cusps()'s return value
 *      is undefined.
 *
 *      void isometry_list_cusp_action( IsometryList    *isometry_list,
 *                                      int              anIsometryIndex,
 *                                      int              aCusp,
 *                                      int              *cusp_image,
 *                                      int              cusp_map[2][2]);
 *
 *      Fills in the cusp_image and cusp_map[2][2] to describe the
 *      action of the given Isometry on the given Cusp.
 *
 *      Boolean isometry_extends_to_link( IsometryList    *isometry_list,
 *                                       int              i);
 *
 *      Returns TRUE if Isometry i extends to the associated links
 *      (i.e. if it takes meridians to meridians), FALSE if it doesn't.
 *
 *      void isometry_list_orientations(
 *          IsometryList    *isometry_list,
 *          Boolean          *contains_orientation_preserving_isometries,
 *          Boolean          *contains_orientation_reversing_isometries);

```

```

*
*       Says whether the IsometryList contains orientation-preserving
*       and/or orientation-reversing elements. Assumes the underlying
*       Triangulations are oriented.
*
*       void free_isometry_list(IsometryList *isometry_list);
*
*       Frees the IsometryList.
*
* The UI never explicitly looks at an IsometryList. It only passes
* them to the above kernel functions to obtain information.
*/

#include "kernel.h"

#define CRUDE_VOLUME_EPSILON    0.01

static int      count_unfilled_cusps(Triangulation *manifold);
static Boolean   same_homology(Triangulation *manifold0, Triangulation *manifold1);
static void      free_isometry(Isometry *isometry);

FuncResult compute_isometries(
    Triangulation *manifold0,
    Triangulation *manifold1,
    Boolean *are_isometric,
    IsometryList **isometry_list,
    IsometryList **isometry_list_of_links)
{
    Triangulation *simplified_manifold0,
    *simplified_manifold1;
    IsometryList *the_isometry_list,
    *the_isometry_list_of_links;
    FuncResult result;

    /*
     * Make sure the variables used to pass back our results
     * are initially empty.
     */

    if ((isometry_list != NULL && *isometry_list != NULL)
        || (isometry_list_of_links != NULL && *isometry_list_of_links != NULL))
        uFatalError("compute_isometries", "isometry");

    /*
     * If one of the spaces isn't a manifold, return func_bad_input.
     */

    if (all_Deahn_coefficients_are_relatively_prime_integers(manifold0) == FALSE
        || all_Deahn_coefficients_are_relatively_prime_integers(manifold1) == FALSE)
        return func_bad_input;

    /*
     * Check whether the manifolds are obviously nonhomeomorphic.
     *
     * (In the interest of a robust, beyond-a-shadow-of-a-doubt algorithm,
     * stick to discrete invariants like the number of cusps and the
     * first homology group, and avoid real-valued invariants like
     * the volume which require judging when two floating point numbers
     * are equal.) [96/12/6 Comparing canonical triangulations
     * relies on having at least a vaguely correct hyperbolic structure,
     * so it should be safe to reject manifolds whose volumes differ
     * by more than, say, 0.01.]
     */

    if (count_unfilled_cusps(manifold0) != count_unfilled_cusps(manifold1)
        || same_homology(manifold0, manifold1) == FALSE
        || ( manifold0->solution_type[filled] == geometric_solution
            && manifold1->solution_type[filled] == geometric_solution
            && fabs(volume(manifold0, NULL) - volume(manifold1, NULL)) >
            CRUDE_VOLUME_EPSILON))
    {
        *are_isometric = FALSE;
        return func_OK;
    }
}

```

```

}

/*
 * Whether the actual manifolds (after taking into account Dehn
 * fillings) have cusps or not, we want to begin by getting rid
 * of "unnecessary" cusps.
 */

simplified_manifold0 = fill_reasonable_cusps(manifold0);
if (simplified_manifold0 == NULL)
    return func_failed;

simplified_manifold1 = fill_reasonable_cusps(manifold1);
if (simplified_manifold1 == NULL)
    return func_failed;

/*
 * Split into cases according to whether the manifolds are
 * closed or cusped (i.e. whether all cusps are filled or not).
 * The above tests insure that either both are closed or both
 * are cusped.
 */

if (all_cusps_are_filled(simplified_manifold0) == TRUE)

    result = compute_closed_isometry(    simplified_manifold0,
                                         simplified_manifold1,
                                         are_isometric);

else
{
    result = compute_cusped_isometries( simplified_manifold0,
                                         simplified_manifold1,
                                         &the_isometry_list,
                                         &the_isometry_list_of_links);

    if (result == func_OK)
    {
        *are_isometric = the_isometry_list->num_isometries > 0;

        if (isometry_list != NULL)
            *isometry_list = the_isometry_list;
        else
            free_isometry_list(the_isometry_list);

        if (isometry_list_of_links != NULL)
            *isometry_list_of_links = the_isometry_list_of_links;
        else
            free_isometry_list(the_isometry_list_of_links);
    }
}

/*
 * We no longer need the simplified manifolds.
 */
free_triangulation(simplified_manifold0);
free_triangulation(simplified_manifold1);

return result;
}

static int count_unfilled_cusps(
    Triangulation *manifold)
{
    int    num_unfilled_cusps;
    Cusp   *cusp;

    num_unfilled_cusps = 0;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)

        if (cusp->is_complete == TRUE)

```

```

        num_unfilled_cusps++;

    return num_unfilled_cusps;
}

static Boolean same_homology(
    Triangulation *manifold0,
    Triangulation *manifold1)
{
    AbelianGroup *g0,
                *g1;
    Boolean groups_are_isomorphic;
    int i;

    /*
     * Compute the homology groups.
     */
    g0 = homology(manifold0);
    g1 = homology(manifold1);

    /*
     * compute_isometries() has already checked that both manifolds
     * really are manifolds, so neither g0 nor g1 should be NULL.
     */
    if (g0 == NULL || g1 == NULL)
        uFatalError("same_homology", "isometry");

    /*
     * Put the homology groups into a canonical form.
     */
    compress_abelian_group(g0);
    compress_abelian_group(g1);

    /*
     * Compare the groups.
     */

    if (g0->num_torsion_coefficients != g1->num_torsion_coefficients)
        groups_are_isomorphic = FALSE;

    else
    {
        groups_are_isomorphic = TRUE; /* innocent until proven guilty */

        for (i = 0; i < g0->num_torsion_coefficients; i++)
            if (g0->torsion_coefficients[i] != g1->torsion_coefficients[i])
                groups_are_isomorphic = FALSE;
    }

    /*
     * Free the Abelian groups.
     */
    free_abelian_group(g0);
    free_abelian_group(g1);

    return groups_are_isomorphic;
}

int isometry_list_size(
    IsometryList *isometry_list)
{
    return isometry_list->num_isometries;
}

int isometry_list_num_cusps(
    IsometryList *isometry_list)
{
    return isometry_list->isometry[0]->num_cusps;
}

```

```

void isometry_list_cusp_action(
    IsometryList *isometry_list,
    int          anIsometryIndex,
    int          aCusp,
    int          *cusp_image,
    int          cusp_map[2][2])
{
    Isometry *the_isometry;
    int      i,
            j;

    the_isometry = isometry_list->isometry[anIsometryIndex];

    *cusp_image = the_isometry->cusp_image[aCusp];

    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            cusp_map[i][j] = the_isometry->cusp_map[aCusp][i][j];
}

```

```

Boolean isometry_extends_to_link(
    IsometryList *isometry_list,
    int          i)
{
    return isometry_list->isometry[i]->extends_to_link;
}

```

```

void isometry_list_orientations(
    IsometryList *isometry_list,
    Boolean      *contains_orientation_preserving_isometries,
    Boolean      *contains_orientation_reversing_isometries)
{
    /*
     * This function assumes the underlying Triangulations
     * are oriented.
     */

    int i;

    *contains_orientation_preserving_isometries = FALSE;
    *contains_orientation_reversing_isometries  = FALSE;

    for (i = 0; i < isometry_list->num_isometries; i++)
        if (parity[isometry_list->isometry[i]->tet_map[0]] == 0)
            *contains_orientation_preserving_isometries = TRUE;
        else
            *contains_orientation_reversing_isometries  = TRUE;
}

```

```

void free_isometry_list(
    IsometryList *isometry_list)
{
    int i;

    if (isometry_list != NULL)
    {
        for (i = 0; i < isometry_list->num_isometries; i++)
            free_isometry(isometry_list->isometry[i]);

        if (isometry_list->num_isometries != 0)
            my_free(isometry_list->isometry);

        my_free(isometry_list);
    }
}

```

```

static void free_isometry(
    Isometry *isometry)
{

```

```
    my_free(isometry->tet_image);
    my_free(isometry->tet_map);
    my_free(isometry->cusp_image);
    my_free(isometry->cusp_map);

    my_free(isometry);
}
```