

```

/*
 * rehydrate_census.c
 *
 * The function
 *
 * void rehydrate_census_manifold(
 *     TersestTriangulation   tersest,
 *     int                    which_census,
 *     int                    which_manifold,
 *     Triangulation          **manifold);
 *
 * calls tersest_to_tri() to rehydrate the manifold, then call
 * resolve_ambiguous_bases() [described below] to insure that the
 * peripheral curves are correct.
 *
 * For certain census manifolds, the "canonical" peripheral curves
 * installed by tersest_to_tri() are not well defined. (The problem
 * is that the cusps do not have unique shortest geodesics, so the
 * geometry of the cusp alone does not provide any way to select
 * a preferred meridian.) The function
 *
 * void resolve_ambiguous_bases(   Triangulation   *theTriangulation,
 *                                 int              aCensus,
 *                                 int              anIndex);
 *
 * resolves this problem for the census manifolds by choosing a
 * set of peripheral curves based on the homology of the manifold as
 * a whole.
 *
 * Comments:
 *
 * (1) Full documentation appears in the files "Read Me" and
 *     "ambiguous examples.symmetries" in the folder "cusped census 2.1".
 *
 * (2) The choices rely on a fixed orientation for the manifold.
 *
 * (3) For most of the 2-cusp manifolds, all (shortest) choices for
 *     the meridian of a single cusp are equivalent, but once you've
 *     chosen it the choices for the remaining cusp are *not* equivalent.
 *     This observation lets us treat the 1-cusp and 2-cusp cases using
 *     the same usual_algorithm() code, which rotates the coordinates
 *     on cusp 0 until the homology is right, regardless of whether the
 *     manifold has a second cusp or not.
 */

#include "kernel.h"

static void      resolve_ambiguous_bases(Triangulation *theTriangulation, int aCensus, int anIndex) ✓
static void      usual_algorithm(Triangulation *aTriangulation, int anM, int anL, CONST ✓
MatrixInt22 *aChangeMatrixArray, int aNumCoefficients, int aFirstCoefficient, int ✓
aSecondCoefficient, int aThirdCoefficient);
static void      algorithm_s596(Triangulation *aTriangulation);
static Boolean   check_homology(Triangulation *aTriangulation, AbelianGroup *anAbelianGroup) ✓
;

/*
 * The documentation at the top of the file change_peripheral_curves.c
 * explains the interpretation of the following "change matrices".
 */

CONST static MatrixInt22
rotate6[2] =
{
    {
        { 0, 1 },
        { -1, 1 }
    },
    {
        { 1, 0 },
        { 0, 1 }
    }
},

```

```

rotate6a[2] = {
    {
        { 1, 0 },
        { 0, 1 }
    },
    {
        { 0, 1 },
        { -1, 1 }
    }
},
rotate4[2] = {
    {
        { 0, 1 },
        { -1, 0 }
    },
    {
        { 1, 0 },
        { 0, 1 }
    }
};

```

```

void rehydrate_census_manifold(
    TersestTriangulation  tersest,
    int                   which_census,
    int                   which_manifold,
    Triangulation         **manifold)
{
    /*
     * Rehydrate the manifold.
     */
    tersest_to_tri(tersest, manifold);

    /*
     * If the manifold happens to be one of the census manifolds
     * with square or hexagonal cusps, make sure the peripheral
     * curves are the standard ones.
     */
    resolve_ambiguous_bases(*manifold, which_census, which_manifold);
}

```

```

static void resolve_ambiguous_bases(
    Triangulation  *theTriangulation,
    int            aCensus,
    int            anIndex)
{
    switch (aCensus)
    {
        case 5:
            switch (anIndex)
            {
                case 003:
                    usual_algorithm(theTriangulation, 1, 0, rotate6, 1, 10, -1, -1);
                    break;

                case 125:
                    usual_algorithm(theTriangulation, 1, 0, rotate4, 1, 3, -1, -1);
                    break;

                case 130:
                    usual_algorithm(theTriangulation, 1, 1, rotate4, 2, 2, 16, -1);
                    break;

                case 135:
                    usual_algorithm(theTriangulation, 1, 1, rotate4, 3, 2, 2, 4);
                    break;

                case 139:
                    usual_algorithm(theTriangulation, 1, 0, rotate4, 1, 24, -1, -1);
                    break;

                case 202:
                    usual_algorithm(theTriangulation, 1, 0, rotate6, 1, 3, -1, -1);

```

```
        break;

    case 208:
        usual_algorithm(theTriangulation, 1, 0, rotate6, 1, 20, -1, -1);
        break;

    default:
        /*
         * Peripheral curves are already well defined by the
         * geometry of the cusp. Don't change them.
         */
        break;
}
break;

case 6:
    switch (anIndex)
    {
        case 594:
            usual_algorithm(theTriangulation, 1, 0, rotate6, 3, 2, 2, 0);
            break;

        case 596:
            algorithm_s596(theTriangulation);
            break;

        case 859:
            usual_algorithm(theTriangulation, 1, 0, rotate4, 1, 6, -1, -1);
            break;

        case 913:
            usual_algorithm(theTriangulation, 1, 0, rotate4, 1, 5, -1, -1);
            break;

        case 955:
            usual_algorithm(theTriangulation, 1, 0, rotate6, 2, 4, 20, -1);
            break;

        case 957:
            usual_algorithm(theTriangulation, 1, 0, rotate6, 2, 4, 4, -1);
            break;

        case 959:
            usual_algorithm(theTriangulation, 1, 0, rotate6, 1, 9, -1, -1);
            break;

        case 960:
            usual_algorithm(theTriangulation, 1, 0, rotate6, 3, 2, 10, 0);
            break;

        default:
            /*
             * Peripheral curves are already well defined by the
             * geometry of the cusp. Don't change them.
             */
            break;
    }
    break;

case 8:    /* really the nonorientable 6-tetrahedron census */
    /*
     * There are no square or hexagonal orientable cusps,
     * so there are no special cases to deal with.
     */
    break;

case 7:
    switch (anIndex)
    {
        case 1859:
            usual_algorithm(theTriangulation, 1, 0, rotate4, 3, 2, 2, 2);
            break;

        case 3318:
```

```

        usual_algorithm(theTriangulation, 1, 0, rotate4, 2, 2, 2, -1);
        break;

    case 3319:
        usual_algorithm(theTriangulation, 1, 0, rotate4, 1, 3, -1, -1);
        break;

    case 3461:
        usual_algorithm(theTriangulation, 1, 0, rotate6, 1, 5, -1, -1);
        break;

    case 3551:
        usual_algorithm(theTriangulation, 1, 0, rotate6, 1, 14, -1, -1);
        break;

    default:
        /*
         * Peripheral curves are already well defined by the
         * geometry of the cusp. Don't change them.
         */
        break;
}
break;

case 9:      /* really the nonorientable 7-tetrahedron census */
    /*
     * There are no square or hexagonal orientable cusps,
     * so there are no special cases to deal with.
     */
    break;

default:
    uFatalError("resolve_ambiguous_bases", "ambiguous_bases");
}
}

static void usual_algorithm(
    Triangulation      *aTriangulation,
    int                anM,
    int                anL,
    CONST MatrixInt22  *aChangeMatrixArray,
    int                aNumCoefficients,
    int                aFirstCoefficient,
    int                aSecondCoefficient,
    int                aThirdCoefficient)
{
    int                i,
                      theRotationCount;
    long               theCoefficientArray[3];
    AbelianGroup       theAbelianGroup;

    /*
     * Set up theAbelianGroup.
     */
    theCoefficientArray[0] = aFirstCoefficient;
    theCoefficientArray[1] = aSecondCoefficient;
    theCoefficientArray[2] = aThirdCoefficient;
    theAbelianGroup.num_torsion_coefficients = aNumCoefficients;
    theAbelianGroup.torsion_coefficients = theCoefficientArray;

    /*
     * Set up an (m,l) Dehn filling on each cusp, relative to the initial
     * (arbitrary) coordinate system. Don't actually compute the
     * hyperbolic structure -- the computation would be slow (compared
     * to what we're doing here) and we don't need the hyperbolic
     * structure to check the homology anyhow.
     */
    for (i = 0; i < get_num_cusps(aTriangulation); i++)
        set_cusp_info(aTriangulation, i, FALSE, anM, anL);

    /*
     * We'll keep track of how many times we've been through the following
     * while() loop. If something goes wrong we should display an error

```

```

    * message instead of looping forever.
    */
    theRotationCount = 0;

/*
 * If the homology isn't what we want, rotate the coordinate system
 * a sixth or quarter turn, according to aChangeMatrixArray.
 * After at most two such rotations we should find the meridian
 * we're looking for. See the file "ambiguous examples.symmetries"
 * for an explanation of how the desired meridians were chosen.
 */
while (check_homology(aTriangulation, &theAbelianGroup) == FALSE)
{
    /*
     * The call to change_peripheral_curves() will adjust the Dehn
     * filling coefficients to compensate for the changed coordinate
     * system, thereby preserving the original Dehn filling.
     * But we want to move on to a new Dehn filling, which is (m,l)
     * in the *new* coordinate system.
     */
    change_peripheral_curves(aTriangulation, aChangeMatrixArray);
    set_cusp_info(aTriangulation, 0, FALSE, anM, anL);

    /*
     * We shouldn't have to rotate more than twice to find
     * the desired meridian.
     */
    if (++theRotationCount > 2)
        uFatalError("usual_algorithm", "ambiguous_bases");
}

/*
 * We've found the correct peripheral curves. Restore the
 * Dehn filling coefficients to their original, unfilled state.
 */
for (i = 0; i < get_num_cusps(aTriangulation); i++)
    set_cusp_info(aTriangulation, i, TRUE, 0.0, 0.0);
}

static void algorithm_s596(
    Triangulation *aTriangulation)
{
    /*
     * Please see the file "ambiguous examples.symmetries"
     * for an explanation of why s596 needs special treatment.
     */

    int theRotationCount;
    long theCoefficientArray[2];
    AbelianGroup theAbelianGroup;

    theAbelianGroup.num_torsion_coefficients = 2;
    theAbelianGroup.torsion_coefficients = theCoefficientArray;
    theAbelianGroup.torsion_coefficients[0] = 2;
    theAbelianGroup.torsion_coefficients[1] = 2;

    set_cusp_info(aTriangulation, 0, FALSE, 1.0, 0.0);
    set_cusp_info(aTriangulation, 1, FALSE, 1.0, 0.0);

    theRotationCount = 0;

    while (check_homology(aTriangulation, &theAbelianGroup) == FALSE)
    {
        /*
         * Cycle through all possible combinations
         * of meridians for cusps 0 and 1.
         */
        if (theRotationCount % 3 == 0)
        {
            change_peripheral_curves(aTriangulation, rotate6);
            set_cusp_info(aTriangulation, 0, FALSE, 1.0, 0.0);
        }
        else

```

```
{
    change_peripheral_curves(aTriangulation, rotate6a);
    set_cusp_info(aTriangulation, 1, FALSE, 1.0, 0.0);
}

if (++theRotationCount > 8)
    uFatalError("algorithm_s596", "ambiguous_bases");
}

set_cusp_info(aTriangulation, 0, TRUE, 0.0, 0.0);
set_cusp_info(aTriangulation, 1, TRUE, 0.0, 0.0);
}

static Boolean check_homology(
    Triangulation    *aTriangulation,
    AbelianGroup     *anAbelianGroup)
{
    AbelianGroup     *theHomology;
    Boolean          theGroupsAreIsomorphic;
    int              i;

    theHomology = homology(aTriangulation);

    if (theHomology == NULL)
        uFatalError("check_homology", "rehydrate_census");

    compress_abelian_group(theHomology);

    if (theHomology->num_torsion_coefficients
        != anAbelianGroup->num_torsion_coefficients)
        theGroupsAreIsomorphic = FALSE;
    else
    {
        theGroupsAreIsomorphic = TRUE;
        for (i = 0; i < theHomology->num_torsion_coefficients; i++)
            if (theHomology->torsion_coefficients[i]
                != anAbelianGroup->torsion_coefficients[i])
                theGroupsAreIsomorphic = FALSE;
    }

    free_abelian_group(theHomology);

    return theGroupsAreIsomorphic;
}
```