

```

/*
 * current_curve_basis.c
 *
 * This file provides the functions
 *
 * void current_curve_basis(      Triangulation  *manifold,
 *                               int             cusp_index,
 *                               MatrixInt22     basis_change);
 *
 * void install_current_curve_bases(  Triangulation  *manifold);
 *
 * current_curve_basis() accepts a Triangulation and a cusp index,
 * and computes a 2 x 2 integer matrix basis_change with the property that
 *
 * if the Cusp of index cusp_index is filled, and has
 *   integer Dehn filling coefficients,
 *
 * the first row of basis_change is set to the current
 *   Dehn filling coefficients (divided by their gcd), and
 * the second row of basis_change is set to the shortest
 *   curve which completes a basis.
 *
 * else
 *
 *   basis_change is set to the identity
 *
 * Note that for nonorientable cusps, the only possible Dehn
 * filling coefficients are +/- (m,0), and +/- the longitude will be the
 * shortest curve which completes the basis.
 *
 * install_current_curve_bases() installs the current curve basis
 *   on each Cusp of manifold.
 *
 * 96/9/28 Modified to accept non relatively prime integer coefficients.
 * For example, (15, 20) is treated the same as (3,4). Thus in
 * the new basis the surgery coefficients will be of the form (m,0),
 * where m is the gcd of the original coefficients.
 *
 * 99/11/05 Added install_current_curve_bases().
 */

#include "kernel.h"

#define EPSILON      1e-5
#define BIG_MODULUS  1e+5

static void current_curve_basis_on_cusp(Cusp *cusp, MatrixInt22 basis_change);

void current_curve_basis(
    Triangulation  *manifold,
    int            cusp_index,
    MatrixInt22     basis_change)
{
    current_curve_basis_on_cusp(    find_cusp(manifold, cusp_index),
                                   basis_change);
}

static void current_curve_basis_on_cusp(
    Cusp            *cusp,
    MatrixInt22     basis_change)
{
    int             m_int,
                   l_int,
                   the_gcd;
    long            a,
                   b;
    Complex          new_shape;
    int             multiple;
    int             i,
                   j;

```

```

m_int = (int) cusp->m;
l_int = (int) cusp->l;

if (cusp->is_complete == FALSE /* cusp is filled and */
    && m_int == cusp->m /* coefficients are integers */
    && l_int == cusp->l)
{
    /*
     * Find a and b such that am + bl = gcd(m, l).
     */
    the_gcd = euclidean_algorithm(m_int, l_int, &a, &b);

    /*
     * Divide through by the g.c.d.
     */
    m_int /= the_gcd;
    l_int /= the_gcd;

    /*
     * Set basis_change to
     *
     *      m  l
     *     -b  a
     */
    basis_change[0][0] = m_int;
    basis_change[0][1] = l_int;
    basis_change[1][0] = -b;
    basis_change[1][1] = a;

    /*
     * Make sure the new longitude is as short as possible.
     * The ratio (new longitude)/(new meridian) should have a
     * real part in the interval (-1/2, +1/2].
     */

    /*
     * Compute the new_shape, using the tentative longitude.
     */
    new_shape = transformed_cusp_shape( cusp->cusp_shape[initial],
                                         basis_change);

    /*
     * 96/10/1 There is a danger that for nonhyperbolic solutions
     * the cusp shape will be ill-defined (either very large or NaN).
     * However for some nonhyperbolic solutions (flat solutions
     * for example) it may make good sense. So we attempt to
     * make the longitude short iff the new_shape is defined and
     * not outrageously large; otherwise we're content with an
     * arbitrary longitude.
     */
    if (complex_modulus(new_shape) < BIG_MODULUS)
    {
        /*
         * Figure out how many meridians we need to subtract
         * from the longitude.
         */
        multiple = (int) floor(new_shape.real - (-0.5 + EPSILON));

        /*
         * longitude -= multiple * meridian
         */
        for (j = 0; j < 2; j++)
            basis_change[1][j] -= multiple * basis_change[0][j];
    }
}
else
{
    /*
     * Set basis_change to the identity.
     */
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            basis_change[i][j] = (i == j);
}
}

```

```
}

void install_current_curve_bases(
    Triangulation *manifold)
{
    Cusp *cusp;
    MatrixInt22 *change_matrices;

    /*
     * Allocate an array to store the change of basis matrices.
     */

    change_matrices = NEW_ARRAY(manifold->num_cusps, MatrixInt22);

    /*
     * Compute the change of basis matrices.
     */

    for (cusp = manifold->cusplink_begin;
         cusp != &manifold->cusplink_end;
         cusp = cusp->next)
    {
        if (cusp->index < 0
            || cusp->index >= manifold->num_cusps)
            uFatalError("install_current_curve_bases", "current_curve_basis");

        current_curve_basis_on_cusp(cusp, change_matrices[cusp->index]);
    }

    /*
     * Install the change of basis matrices.
     */

    if (change_peripheral_curves(manifold, change_matrices) != func_OK)
        uFatalError("install_current_curve_bases", "current_curve_basis");

    /*
     * Free the array used to store the change of basis matrices.
     */

    my_free(change_matrices);
}
```