

```

/*
 * shortest_cusp_basis.c
 *
 * This file provides the functions
 *
 *      Complex cusp_modulus(          Complex          cusp_shape);
 *
 *      void      shortest_cusp_basis(      Complex          cusp_shape,
 *                                          MatrixInt22      basis_change);
 *
 *      Complex transformed_cusp_shape( Complex          cusp_shape,
 *                                     CONST MatrixInt22 basis_change);
 *
 *      void      install_shortest_bases( Triangulation      *manifold);
 *
 * cusp_modulus() accepts a cusp_shape (longitude/meridian) and returns
 * the cusp modulus. Loosely speaking, the cusp modulus is defined as
 * (second shortest translation)/(shortest translation); it is a
 * complex number z lying in the region  $|\operatorname{Re}(z)| \leq 1/2$  &&  $|z| \geq 1$ .
 * If z lies on the boundary of this region, we choose it so that
 *  $\operatorname{Re}(z) \geq 0$ .
 *
 * shortest_cusp_basis() accepts a cusp_shape (longitude/meridian) and
 * computes the 2 x 2 integer matrix which transforms the old basis
 * (u, v) = (meridian, longitude) to the new basis
 * (u', v') = (shortest, second shortest). That is,
 *
 *      
$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} & \\ & \end{pmatrix} \text{basis\_change} \begin{pmatrix} u \\ v \end{pmatrix}$$

 *
 *      2 x 1      2 x 2      2 x 1
 *      complex    integer    complex
 *      vector     matrix     vector
 *
 * (u', v') is such that v'/u' is the cusp modulus defined above.
 *
 * Occasionally the shortest or second shortest curve won't be
 * unique. In most cases the conventions for the cusp modulus stated
 * above (in particular the convention that  $\operatorname{Re}(z) \geq 0$  when z is on
 * the boundary of the fundamental domain) serve to uniquely specify
 * (u', v') in spite of the nonuniqueness of lengths. The only
 * exceptions are the hexagonal lattice, where three different curves
 * all have minimal length, and the square lattice, where two different
 * curves have minimal length. In these cases the ambiguity is not
 * resolved, and the choice of (u', v') may be machine dependent.
 *
 * transformed_cusp_shape() accepts a cusp_shape and a basis_change,
 * and computes the shape of the cusp relative to the basis (u', v')
 * defined by
 *
 *      
$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} & \\ & \end{pmatrix} \text{basis\_change} \begin{pmatrix} u \\ v \end{pmatrix}$$

 *
 * (u', v') need not be the (shortest, second shortest) basis.
 *
 * install_shortest_bases() installs the (shortest, second shortest)
 * basis on each torus Cusp of manifold. It does not change the bases
 * on Klein bottle cusps. As explained for shortest_cusp_basis()
 * above, the (shortest, second shortest) is not well defined for a
 * hexagonal lattice, and the results may be machine dependent.
 *
 * shortest_cusp_basis() uses the following algorithm. In principle
 * u and v could be any two translations which generate the fundamental
 * group of a torus, although in shortest_cusp_basis(), u is initially 1
 * and v is initially the cusp_shape.
 *
 *      do
 *      {
 *
 *          if ( $|u + v| < |u|$ )
 *              u += v;

```

```

*
*      if ( $|u - v| < |u|$ )
*           $u -= v$ ;
*
*      if ( $|v + u| < |v|$ )
*           $v += u$ ;
*
*      if ( $|v - u| < |v|$ )
*           $v -= u$ ;
*
*      } while (still making progress);
*
*      if ( $|u| > |v|$ )
*          replace (u, v) with (v, -u)
*
*      if ( $\text{Im}(v/u) < 0$ )
*          flag an error -- the original orientation was wrong
*
*      if (v/u is on the boundary of the fundamental domain described above)
*          make sure  $\text{Re}(v/u) \geq 0$ 
*
* Theorem. The above algorithm computes the
* (shortest, second shortest) basis.
*
* Proof. The angle between u and v must be between  $\pi/3$  and  $2\pi/3$ ;
* otherwise the length of the projection of v onto u would be
*  $|v| \cos(\theta) > |u| \cos(\pi/3) = |u|/2$ , and we would have added
*  $+u$  to v, thereby shortening v. This shows that  $|\text{Re}(v/u)| \leq 1/2$ .
* Because we chose  $|u| \leq |v|$ ,  $|v/u| \geq 1$ . Therefore v/u lies within
* the fundamental domain described above. It is also easy to see that
* v is the shortest translation which is linearly independent of u.
* The reason is that the row of lattice points  $2v + nu$  is a distance
* at least  $2|v| \sin(\theta) \geq 2|u| \sin(\pi/3) = \sqrt{3}|u|$  away from
* the row  $0v + nu$ .
*/

#include "kernel.h"

#define EPSILON (1e5 * DBL_EPSILON)

Complex cusp_modulus(
    Complex      cusp_shape)
{
    MatrixInt22 basis_change;

    shortest_cusp_basis(cusp_shape, basis_change);
    return transformed_cusp_shape(cusp_shape, basis_change);
}

void shortest_cusp_basis(
    Complex      cusp_shape,
    MatrixInt22 basis_change) /* basis_change is an output variable here */
{
    Complex u,
            v,
            u_plus_v,
            u_minus_v,
            temp,
            cusp_modulus;
    double  mod_u,          /* These are the complex moduli      */
            mod_v,          /* of the preceding variables.      */
            mod_u_plus_v,
            mod_u_minus_v;
    int     i,
            j,
            temp_int;
    Boolean progress;

    /*
     * For an explanation of this algorithm, see the documentation above.
     */
}

```

```

/*
 * Make sure cusp_shape is nondegenerate.
 */
if (fabs(cusp_shape.imag) < EPSILON)
{
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            basis_change[i][j] = 0;
    return;
}

u = One;
v = cusp_shape;

mod_u = complex_modulus(u);
mod_v = complex_modulus(v);

for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
        basis_change[i][j] = (i == j);

do
{
    progress = FALSE;

    u_plus_v = complex_plus(u,v);
    mod_u_plus_v = complex_modulus(u_plus_v);
    if (mod_u - mod_u_plus_v > EPSILON)
    {
        u = u_plus_v;
        mod_u = mod_u_plus_v;
        for (j = 0; j < 2; j++)
            basis_change[0][j] += basis_change[1][j];
        progress = TRUE;
    }
    else if (mod_v - mod_u_plus_v > EPSILON)
    {
        v = u_plus_v;
        mod_v = mod_u_plus_v;
        for (j = 0; j < 2; j++)
            basis_change[1][j] += basis_change[0][j];
        progress = TRUE;
    }

    u_minus_v = complex_minus(u,v);
    mod_u_minus_v = complex_modulus(u_minus_v);
    if (mod_u - mod_u_minus_v > EPSILON)
    {
        u = u_minus_v;
        mod_u = mod_u_minus_v;
        for (j = 0; j < 2; j++)
            basis_change[0][j] -= basis_change[1][j];
        progress = TRUE;
    }
    else if (mod_v - mod_u_minus_v > EPSILON)
    {
        v = complex_negate(u_minus_v);
        mod_v = mod_u_minus_v;
        for (j = 0; j < 2; j++)
            basis_change[1][j] -= basis_change[0][j];
        progress = TRUE;
    }
} while (progress);

if (mod_u > mod_v + EPSILON)
{
    temp = u;
    u = v;
    v = complex_negate(temp);

    for (j = 0; j < 2; j++)
    {
        temp_int = basis_change[0][j];

```

```

        basis_change[0][j] = basis_change[1][j];
        basis_change[1][j] = - temp_int;
    }
}

cusp_modulus = complex_div(v,u);

if (cusp_modulus.imag < 0)
    uFatalError("cusp_modulus", "cusp_modulus");

if (cusp_modulus.real < -0.5 + EPSILON)
{
    /*
     * Do an extra v += u.
     */

    cusp_modulus.real = 0.5;

    for (j = 0; j < 2; j++)
        basis_change[1][j] += basis_change[0][j];
}

if (complex_modulus(cusp_modulus) < 1.0 + EPSILON)
{
    if (cusp_modulus.real < -EPSILON)
    {
        /*
         * Replace (u,v) with (v,-u).
         */
        cusp_modulus.real = - cusp_modulus.real;
        for (j = 0; j < 2; j++)
        {
            temp_int
                = basis_change[0][j];
            basis_change[0][j] = basis_change[1][j];
            basis_change[1][j] = - temp_int;
        }
    }
}
}

Complex transformed_cusp_shape(
    Complex      cusp_shape,
    CONST MatrixInt22  basis_change) /* basis_change is an input variable here */
{
    Complex u,
            v;

    u = complex_plus(
        complex_real_mult(
            basis_change[0][0],
            One
        ),
        complex_real_mult(
            basis_change[0][1],
            cusp_shape
        )
    );

    v = complex_plus(
        complex_real_mult(
            basis_change[1][0],
            One
        ),
        complex_real_mult(
            basis_change[1][1],
            cusp_shape
        )
    );

    if (complex_modulus(u) < EPSILON)
        return Infinity;
    else
        return complex_div(v,u);
}

```

```
}

void install_shortest_bases(
    Triangulation *manifold)
{
    Cusp *cusp;
    MatrixInt22 *change_matrices;
    int i,
        j;

    /*
     * 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->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)

        if (cusp->topology == torus_cusp)

            shortest_cusp_basis( cusp->cusp_shape[initial],
                                change_matrices[cusp->index]);

        else

            for (i = 0; i < 2; i++)
                for (j = 0; j < 2; j++)
                    change_matrices[cusp->index][i][j] = (i == j);

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

    if (change_peripheral_curves(manifold, change_matrices) != func_OK)

        uFatalError("install_shortest_bases", "shortest_cusp_basis");

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

    my_free(change_matrices);
}
```