

```

/*
 * cusp_shapes.c
 *
 * This file provides the function
 *
 * void compute_cusp_shapes(Triangulation *manifold,
 *                          FillingStatus which_structure);
 *
 * which computes the shape of each unfilled cusp. The shape of an
 * orientable cusp is defined to be the ratio of the complex numbers
 * representing the longitudinal and meridional translations
 * (i.e. longitude/meridian). The shape of a nonorientable cusp is
 * defined to be the shape of the cusp's orientation double cover;
 * it is always pure imaginary. (Another reasonable definition for
 * the shape of a nonorientable cusp would be to divide the shape
 * of the orientation double cover by two, to correspond more closely
 * to the geometry of the nonorientable cusp's fundamental domain,
 * but I decided not to do this.)
 *
 * compute_cusp_shapes() computes the cusp shapes using the which_structure
 * (initial or current) hyperbolic structure, and stores each computed cusp
 * shape in the cusp_shape[which_structure] field of the Cusp data structure.
 *
 * To estimate the computed cusp shape's precision, compute_cusp_shapes()
 * computes the cusp shape using both the ultimate and penultimate values of
 * the Tetrahedron shapes. The number of digits to which the answers agree
 * is stored in the shape_precision[which_structure] field of the Cusp data
 * structure.
 *
 * do_Dehn_filling() calls compute_cusp_shapes() to maintain correct values
 * in the cusp_shape[current] field of each unfilled cusp whenever a
 * hyperbolic structure is present. find_complete_hyperbolic_structure()
 * takes responsibility for copying the original shape of each cusp into
 * the cusp_shape[initial] field.
 *
 * Cusp shapes are computed iff manifold->solution_type[which_structure]
 * is geometric_solution, nongeometric_solution, flat_solution, or
 * (tentatively) other_solution. For other solution types (degenerate_solution,
 * etc.) all cusp shapes are set to zero.
 *
 * Technical note: with the usual orientation conventions for the
 * longitude and meridian, we must view the cusp from the fat part
 * of the manifold looking out towards infinity in order to have the
 * imaginary part of the cusp shape be positive. The code
 * in compute_translation() views the cusp from infinity looking in
 * towards the fat part of the manifold (as usual), and then
 * compute_one_cusp_shape() takes the complex conjugate of the shape
 * at the very end.
 *
 * The function shortest_cusp_basis() in shortest_cusp_basis.c converts
 * the (meridian, longitude) basis to the (shortest, second shortest)
 * basis. cusp_modulus() uses the latter to compute the cusp modulus
 * as (second shortest translation)/(shortest translation).
 *
 * 96/9/27 Added the which_structure parameter to compute_cusp_shapes()
 * so it can compute the cusp shapes for either the initial (complete)
 * or the current (filled) hyperbolic structure. Previously it used only
 * the current structure.
 */

#include "kernel.h"

static void compute_the_cusp_shapes(Triangulation *manifold, FillingStatus which_structure);
static void set_all_shapes_to_zero(Triangulation *manifold, FillingStatus which_structure);
static void compute_one_cusp_shape(Triangulation *manifold, Cusp *cusp, FillingStatus which_structure);
static void compute_translation(PositionedTet *initial_ptet, PeripheralCurve which_curve, TraceDirection which_direction, Complex translation[2], FillingStatus which_structure);
static PositionedTet find_start(Triangulation *manifold, Cusp *cusp);

```

```

void compute_cusp_shapes(
    Triangulation *manifold,
    FillingStatus which_structure)
{
    /*
     * If we have some reasonable (or semi-reasonable) hyperbolic
     * structure, then compute the cusp shapes. Otherwise, fill in
     * all shapes with zeros.
     */

    switch (manifold->solution_type[which_structure])
    {
        case geometric_solution:
        case nongeometric_solution:
        case flat_solution:
        case other_solution:    /* we'll give the cusps of other_solution a try */
                                /* other_solution can be moved below if its */
                                /* cusp shapes can't be computed meaningfully */

            compute_the_cusp_shapes(manifold, which_structure);

            break;

        case not_attempted:
        case degenerate_solution:
        case no_solution:

            set_all_shapes_to_zero(manifold, which_structure);

            break;
    }
}

static void compute_the_cusp_shapes(
    Triangulation *manifold,
    FillingStatus which_structure)
{
    Cusp *cusp;

    /*
     * Call compute_one_cusp_shape() for each unfilled cusp.
     */

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

        if ( which_structure == initial
            || (which_structure == current && cusp->is_complete))

            compute_one_cusp_shape(manifold, cusp, which_structure);

        else
        {
            cusp->cusp_shape[which_structure] = Zero;
            cusp->shape_precision[which_structure] = 0;
        }
}

static void set_all_shapes_to_zero(
    Triangulation *manifold,
    FillingStatus which_structure)
{
    Cusp *cusp;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)
    {
        cusp->cusp_shape[which_structure] = Zero;
        cusp->shape_precision[which_structure] = 0;
    }
}

```

```

    }
}

static void compute_one_cusp_shape(
    Triangulation *manifold,
    Cusp *cusp,
    FillingStatus which_structure)
{
    PositionedTet initial_ptet;
    TraceDirection direction[2]; /* direction[M/L] */
    Complex translation[2][2], /* translation[M/L][ultimate/penultimate] */
    shape[2]; /* shape[ultimate/penultimate] */
    int i;

    /*
     * Compute the longitudinal and meridional translations, and
     * divide them to get the cusp shape.
     *
     * Do parallel computations for the ultimate and penultimate shapes,
     * to estimate the accuracy of the final answer.
     */

    /*
     * Find and position a tetrahedron so that the near edge of the top
     * vertex intersects both the meridian and the longitude.
     */
    initial_ptet = find_start(manifold, cusp);

    for (i = 0; i < 2; i++) /* which curve */
    {
        /*
         * Decide whether the meridian and longitude cross the near edge of the
         * top vertex in a forwards or backwards direction.
         */
        direction[i] =
            (initial_ptet.tet->curve[i][initial_ptet.orientation] [initial_ptet.
bottom_face] [initial_ptet.near_face] > 0) ?
            trace_forwards:
            trace_backwards;

        /*
         * Compute the translation.
         */
        compute_translation(&initial_ptet, i, direction[i], translation[i],
which_structure);
    }

    /*
     * Compute the cusp shape.
     */
    for (i = 0; i < 2; i++) /* i = ultimate, penultimate */
        shape[i] = complex_div(translation[L][i], translation[M][i]); /* will handle
division by Zero correctly */

    /*
     * Record the cusp shape and its accuracy.
     */
    cusp->cusp_shape[which_structure] = shape[ultimate];
    cusp->shape_precision[which_structure] = complex_decimal_places_of_accuracy(shape
[ultimate], shape[penultimate]);

    /*
     * Adjust for the fact that the meridian and/or the longitude may have
     * been traced backwards.
     */
    if (direction[M] != direction[L])
    {
        cusp->cusp_shape[which_structure].real = - cusp->cusp_shape[which_structure].real;
        cusp->cusp_shape[which_structure].imag = - cusp->cusp_shape[which_structure].imag;
    }

    /*
     * As explained at the top of this file, the usual convention for the

```

```

    * cusp shape requires viewing the cusp from the fat part of
    * the manifold looking out, rather than from the cusp looking in, as
    * in done in the rest of SnapPea. For this reason, we must take the
    * complex conjugate of the final cusp shape.
    */
    cusp->cusp_shape[which_structure].imag = - cusp->cusp_shape[which_structure].imag;
}

static void compute_translation(
    PositionedTet    *initial_ptet,
    PeripheralCurve  which_curve,
    TraceDirection   which_direction,
    Complex          translation[2], /* returns translations based on ultimate    */
                                   /* and penultimate shapes                    */
    FillingStatus    which_structure)
{
    PositionedTet    ptet;
    int              i,
                    initial_strand,
                    strand,
                    *this_vertex,
                    near_strands,
                    left_strands;
    Complex          left_endpoint[2], /* left_endpoint[ultimate/penultimate] */
                    right_endpoint[2], /* right_endpoint[ultimate/penultimate] */
                    old_diff,
                    new_diff,
                    rotation;

    /*
    * Place the near edge of the top vertex of the initial_ptet in the
    * complex plane with its left endpoint at zero and its right endpoint at one.
    * Trace the curve which_curve in the direction which_direction, using the
    * shapes of the ideal tetrahedra to compute the position of endpoints of
    * each edge we cross. When we return to our starting point in the manifold,
    * the position of the left endpoint (or the position of the right endpoint
    * minus one) will tell us the translation.
    *
    * Note that we are working in the orientation double cover of the cusp.
    *
    * Here's how we keep track of where we are. At each step, we are always
    * at the near edge of the top vertex (i.e. the truncated vertex opposite
    * the bottom face) of the PositionedTet ptet. The curve (i.e. the
    * meridian or longitude) may cross that edge several times. The variable
    * "strand" keeps track of which intersection we are at; 0 means we're at
    * the strand on the far left, 1 means we're at the next strand, etc.
    */

    ptet          = *initial_ptet;
    initial_strand = 0;
    strand         = initial_strand;
    for (i = 0; i < 2; i++) /* i = ultimate, penultimate */
    {
        left_endpoint[i] = Zero;
        right_endpoint[i] = One;
    }

    do
    {
        /*
        * Note the curve's intersection numbers with the near side and left side.
        */
        this_vertex = ptet.tet->curve[which_curve][ptet.orientation][ptet.bottom_face];
        near_strands = this_vertex[ptet.near_face];
        left_strands = this_vertex[ptet.left_face];

        /*
        * If we are tracing the curve backwards, negate the intersection numbers
        * so the rest of compute_translation() can enjoy the illusion that we
        * are tracing the curve forwards.
        */
        if (which_direction == trace_backwards)
        {

```

```

    near_strands = - near_strands;
    left_strands = - left_strands;
}

/*
 * Does the current strand bend to the left or to the right?
 */

if (strand < FLOW(near_strands, left_strands))
{
    /*
     * The current strand bends to the left.
     */

    /*
     * The left_endpoint remains fixed.
     * Update the right_endpoint.
     *
     * The plan is to compute the vector old_diff which runs
     * from left_endpoint to right_endpoint, multiply it by the
     * complex edge parameter to get the vector new_diff which
     * runs from left_endpoint to the new value of right_endpoint,
     * and then add new_diff to left_endpoint to get the new
     * value of right_endpoint itself.
     *
     * Note that the complex edge parameters are always expressed
     * relative to the right_handed Orientation, so if we are
     * viewing this Tetrahedron relative to the left_handed
     * Orientation, we must take the conjugate-inverse of the
     * edge parameter.
     */
    for (i = 0; i < 2; i++) /* i = ultimate, penultimate */
    {
        old_diff = complex_minus(right_endpoint[i], left_endpoint[i]);
        rotation = ptet.tet->shape[which_structure]->cwl[i][edge3_between_faces
[ptet.near_face][ptet.left_face]].rect;
        if (ptet.orientation == left_handed)
        {
            rotation = complex_div(One, rotation); /* invert . . .
*/
            rotation.imag = - rotation.imag; /* . . . and conjugate
*/
        }
        new_diff = complex_mult(old_diff, rotation);
        right_endpoint[i] = complex_plus(left_endpoint[i], new_diff);
    }

    /*
     * strand remains unchanged.
     */

    /*
     * Move the PositionedTet onward, following the curve.
     */
    veer_left(&ptet);
}
else
{
    /*
     * The current strand bends to the right.
     *
     * Proceed as above, but note that
     *
     * (1) We now divide by the complex edge parameter
     *     instead of multiplying by it.
     *
     * (2) We must adjust the variable "strand". Some of the strands
     *     from the near edge may be peeling off to the left (in which
     *     case left_strands is negative), or some strands from the left
     *     edge may be joining those from the near edge in passing to
     *     the right edge (in which case left_strands is positive).
     *     Either way, the code "strand += left_strands" is correct.
     */

```



```
        {
            ptet.left_face  = remaining_face[ptet.near_face][ptet.
bottom_face];
            ptet.right_face = remaining_face[ptet.bottom_face][ptet.
near_face];
        }
        ptet.orientation   = orientation;

        /*
         * . . . and return it.
         */
        return ptet;
    }

    /*
     * The program should never get to this point.
     */
    uFatalError("find_start", "cusp_shapes");

    /*
     * The compiler would like a return value, even though
     * we never return from the uFatalError() call.
     * The Metrowerks compiler would, in addition, like
     * the data to be initialized before use.
     */
    ptet.tet          = NULL;
    ptet.near_face    = 0;
    ptet.left_face    = 0;
    ptet.right_face   = 0;
    ptet.bottom_face  = 0;
    ptet.orientation  = unknown_orientation;
    return ptet;
}
```