

```

/*
 * holonomy.c
 *
 * This file provides the functions
 *
 * void compute_holonomies(Triangulation *manifold);
 * void compute_edge_angle_sums(Triangulation *manifold);
 *
 * which the functions compute_complex_equations() and
 * compute_real_equations() in gluing_equations.c call. They store
 * their results directly into Triangulation *manifold, so whenever
 * a hyperbolic structure has been found, you may also assume the
 * holonomies are present. (The edge angle sums will be present too,
 * but since they'll all be  $2\pi i$  they won't be very interesting.)
 *
 * The most accurate holonomies are stored in holonomy[ultimate][M]
 * and holonomy[ultimate][L]. The fields holonomy[penultimate][M]
 * and holonomy[penultimate][L] store the holonomies from the
 * penultimate iteration of Newton's method, for use in estimating
 * the numerical error.
 *
 * The holonomy of a Klein bottle cusp deserves special discussion.
 *
 * (1) The holonomy of the meridian may have a rotational part, but
 * never has a contraction part. I.e. it's log is pure imaginary.
 * Proof: the meridian is freely homotopic to it's own inverse.
 *
 * (2) The holonomy of the longitude doesn't even make sense. As
 * you tilt the angle of the curve at the basepoint, the angle
 * of the next lift down the line rotates in the opposite
 * direction. The rotational part of the holonomy is not an
 * isotopy invariant for an orientation-reversing curve. The
 * contraction is an isotopy invariant, but it's cleaner and
 * simpler to work with the preimage of the longitude in the
 * Klein bottle's orientation double cover, which is a torus.
 * The curve on the double cover must have zero rotational part,
 * because the orientation-reversing covering transformation
 * takes the curve to itself.
 *
 * These observations imply that a Dehn filling on a Klein bottle cusp
 * must be of the form  $(m,0)$ .
 *
 * 96/9/27 I split compute_holonomies() into separate functions
 * copy_holonomies_ultimate_to_penultimate() and compute_the_holonomies(),
 * so that other functions (e.g. cover.c) can call compute_the_holonomies()
 * to compute the penultimate holonomies directly.
 */

#include "kernel.h"

static void copy_holonomies_ultimate_to_penultimate(Triangulation *manifold);

void compute_holonomies(
    Triangulation *manifold)
{
    copy_holonomies_ultimate_to_penultimate(manifold);
    compute_the_holonomies(manifold, ultimate);
}

static void copy_holonomies_ultimate_to_penultimate(
    Triangulation *manifold)
{
    /*
     * Copy holonomy[ultimate][] into holonomy[penultimate][].
     */

    Cusp *cusp;
    int i;

    for (cusp = manifold->cusp_list_begin.next;

```

```

    cusp != &manifold->cusp_list_end;
    cusp = cusp->next)

    for (i = 0; i < 2; i++)      /* i = M, L */

        cusp->holonomy[penultimate][i] = cusp->holonomy[ultimate][i];
}

void compute_the_holonomies(
    Triangulation    *manifold,
    Ultimateness     which_iteration)
{
    Cusp             *cusp;
    Tetrahedron      *tet;
    Complex           log_z[2];
    VertexIndex      v;
    FaceIndex         initial_side,
                      terminal_side;
    int               init[2][2],
                      term[2][2];
    int               i,
                      j;

    /*
     * Initialize holonomy[which_iteration][] to zero.
     */

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

        for (i = 0; i < 2; i++)      /* i = M, L */

            cusp->holonomy[which_iteration][i] = Zero;

    /*
     * Now add the contribution of each tetrahedron.
     *
     * The cross section of the ideal vertex v is the union
     * of two triangles, one with the right_handed orientation
     * and one with the left_handed orientation (please see the
     * documentation at the top of peripheral_curves.c for details).
     *
     * This loop is similar to the loop in compute_complex_derivative()
     * in gluing_equations.c.
     */

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)

        for (v = 0; v < 4; v++)

            for (initial_side = 0; initial_side < 4; initial_side++)
            {
                if (initial_side == v)
                    continue;

                terminal_side = remaining_face[v][initial_side];

                /*
                 * Note the log of the complex edge parameter,
                 * for use with the right_handed triangle.
                 */

                log_z[right_handed] = tet->shape[filled]->cwl[which_iteration][
edge3_between_faces[initial_side][terminal_side] ].log;

                /*
                 * The conjugate of log_z[right_handed] will apply to
                 * the left_handed triangle.
                 */

```

```

    log_z[left_handed] = complex_conjugate(log_z[right_handed]);

    /*
     * Note the intersection numbers of the meridian and
     * longitude with the initial and terminal sides.
     */

    for (i = 0; i < 2; i++) {          /* which curve */
        for (j = 0; j < 2; j++) {      /* which sheet */
            init[i][j] = tet->curve[i][j][v][initial_side];
            term[i][j] = tet->curve[i][j][v][terminal_side];
        }
    }

    /*
     * holonomy[which_iteration][i] +=
     *     FLOW(init[i][right_handed], term[i][right_handed]) * log_z
[right_handed]
     *     + FLOW(init[i][left_handed ], term[i][left_handed ]) * log_z
[left_handed ];
     */
    for (i = 0; i < 2; i++)          /* which curve */
#endif
The stupid Symantec C compiler is choking on the following expression.
So I am breaking it into two parts to make its work easier.

original version:
    tet->cuspid[v]->holonomy[which_iteration][i] = complex_plus(
        tet->cuspid[v]->holonomy[which_iteration][i],
        complex_plus(
            complex_real_mult(
                FLOW(init[i][right_handed], term[i][right_handed]),
                log_z[right_handed]
            ),
            complex_real_mult(
                FLOW(init[i][left_handed], term[i][left_handed]),
                log_z[left_handed]
            )
        )
    );

modified version:
#else
    {
        Complex temp;

        temp = complex_plus(
            tet->cuspid[v]->holonomy[which_iteration][i],
            complex_plus(
                complex_real_mult(
                    FLOW(init[i][right_handed], term[i][right_handed]),
                    log_z[right_handed]
                ),
                complex_real_mult(
                    FLOW(init[i][left_handed], term[i][left_handed]),
                    log_z[left_handed]
                )
            )
        );

        tet->cuspid[v]->holonomy[which_iteration][i] = temp;
    }

#endif
}

void compute_edge_angle_sums(
    Triangulation *manifold)
{
    EdgeClass *edge;
    Tetrahedron *tet;
    EdgeIndex e;

```

```
/*
 * Initialize all edge_angle_sums to zero.
 */

for ( edge = manifold->edge_list_begin.next;
      edge != &manifold->edge_list_end;
      edge = edge->next)

    edge->edge_angle_sum = Zero;

/*
 * Add in the contribution of each edge of each tetrahedron.
 *
 * If the EdgeClass sees the Tetrahedron as right_handed,
 * add in the log of the edge parameter directly.  If it sees
 * it as left_handed, add in the log of the conjugate-inverse
 * (i.e. add the imaginary part of the log as usual, but subtract
 * the real part).
 */

for (tet = manifold->tet_list_begin.next;
      tet != &manifold->tet_list_end;
      tet = tet->next)

    for (e = 0; e < 6; e++)
    {
        tet->edge_class[e]->edge_angle_sum.imag
            += tet->shape[filled]->cwl[ultimate][edge3[e]].log.imag;

        if (tet->edge_orientation[e] == right_handed)

            tet->edge_class[e]->edge_angle_sum.real
                += tet->shape[filled]->cwl[ultimate][edge3[e]].log.real;

        else

            tet->edge_class[e]->edge_angle_sum.real
                -= tet->shape[filled]->cwl[ultimate][edge3[e]].log.real;
    }
}
```