

```

/*
 * cusp_cross_sections.c
 *
 * This file provides the high-level functions
 *
 * void allocate_cross_sections(Triangulation *manifold);
 * void free_cross_sections(Triangulation *manifold);
 * void compute_cross_sections(Triangulation *manifold);
 * void compute_tilts(Triangulation *manifold);
 *
 * for use within the kernel, in particular by canonize().
 *
 * It also provides the low-level functions
 *
 * void compute_three_edge_lengths(Tetrahedron *tet, VertexIndex v,
 *                                FaceIndex f, double known_length);
 * void compute_tilts_for_one_tet(Tetrahedron *tet);
 *
 * for its own use, and for the use of two_to_three() and
 * three_to_two() in simplify_triangulation.c (so they can
 * maintain cusp cross sections and tilts correctly).
 * Further documentation of compute_three_edge_lengths()
 * and compute_tilts_for_one_tet() appears in the code itself.
 *
 * The cusp cross section functions, as well as canonize(), use the
 * concepts and terminology of
 *
 * J. Weeks, Convex hulls and isometries of cusped hyperbolic
 * 3-manifolds, Topology Appl. 52 (1993) 127-149.
 *
 * The Tilt Theorem (contained in the above paper) is generalized
 * and given a nicer proof in
 *
 * M. Sakuma and J. Weeks, The generalized tilt formula,
 * Geometriae Dedicata 55 (1995) 115-123.
 *
 * compute_cross_sections() and compute_tilts() set the cross_section
 * and tilt fields, respectively, of the Tetrahedron data structure.
 *
 * The vertex cross section at vertex v of Tetrahedron tet is a
 * triangle. The length of its edge incident to face f of tet is
 * stored as tet->cross_section->edge_length[v][f]. (The edge_length
 * is undefined when v == f.)
 *
 * tet->tilt[f] stores the tilt of the Tetrahedron tet relative to face f.
 *
 * By convention,
 *
 * when no cusp cross sections are in place, the cross_section field
 * of each Tetrahedron is set to NULL, and
 *
 * when cusp cross sections are created, the routine that creates
 * them must allocate the VertexCrossSections structures.
 *
 * Thus, routines which modify a triangulation (e.g. the two_to_three()
 * and three_to_two() moves) know that they must keep track of cusp cross
 * sections if and only if the cross_section fields of the Tetrahedra are
 * not NULL.
 *
 * allocate_cross_sections() and free_cross_sections() allocate and
 * free the VertexCrossSections.
 *
 * compute_cross_sections() sets the (already allocated) VertexCrossSections
 * to correspond to cusp cross sections of area (3/8)sqrt(3). As explained
 * in cusp_neighborhoods.c, such cusp cross sections will always have
 * nonoverlapping interiors.
 *
 * compute_tilts() applies the Tilt Theorem (see "Convex hulls...")
 * to compute the tilts from the VertexCrossSections.
 *
 * The standard way to use these functions is
 *
 * allocate_cross_sections(manifold);
 * compute_cross_sections(manifold);

```

```

*      compute_tilts(manifold);
*      ***      Do stuff with the tilts, possibly including calls to      ***
*      ***      two_to_three() and three_to_two(), which update the      ***
*      ***      cross_sections and tilts correctly whenever the          ***
*      ***      cross_section pointers are not NULL.                      ***
*      free_cross_sections(manifold);
*/

#include "kernel.h"

#define CIRCUMRADIUS_EPSILON      1e-10

typedef struct ideal_vertex
{
    Tetrahedron      *tet;
    VertexIndex      v;
    struct ideal_vertex *next;
} IdealVertex;

static void      initialize_flags(Triangulation *manifold);
static void      cross_section(Triangulation *manifold, Cusp *cusp);
static void      find_starting_point(Triangulation *manifold, Cusp *cusp, Tetrahedron **tet0,
    , VertexIndex *v0);
static double    vertex_area(IdealVertex *ideal_vertex);
static void      normalize_cusp(Triangulation *manifold, Cusp *cusp, double cusp_area);

void allocate_cross_sections(
    Triangulation *manifold)
{
    Tetrahedron *tet;

    for (tet = manifold->tet_list_begin.next;
        tet != &manifold->tet_list_end;
        tet = tet->next)
    {
        /*
         * Just for good measure, make sure no VertexCrossSections
         * are already allocated.
         */
        if (tet->cross_section != NULL)
            uFatalError("allocate_cross_sections", "cusp_cross_sections");

        /*
         * Allocate a VertexCrossSections structure.
         */
        tet->cross_section = NEW_STRUCT(VertexCrossSections);
    }
}

void free_cross_sections(
    Triangulation *manifold)
{
    Tetrahedron *tet;

    for (tet = manifold->tet_list_begin.next;
        tet != &manifold->tet_list_end;
        tet = tet->next)
    {
        /*
         * Just for good measure, make sure the VertexCrossSections
         * really are there.
         */
        if (tet->cross_section == NULL)
            uFatalError("free_cross_sections", "cusp_cross_sections");

        /*
         * Free the VertexCrossSections structure, and set the pointer
         * to NULL.
         */
        my_free(tet->cross_section);
        tet->cross_section = NULL;
    }
}

```

```

    }
}

void compute_cross_sections(
    Triangulation *manifold)
{
    Cusp *cusp;

    /*
     * Initialize cross_section->has_been_set flags to FALSE.
     */

    initialize_flags(manifold);

    /*
     * Compute a cross section of area  $(3/8)\sqrt{3}$  for each cusp.
     */

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

        cross_section(manifold, cusp);
}

static void initialize_flags(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    VertexIndex v;

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

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

            tet->cross_section->has_been_set[v] = FALSE;
}

static void cross_section(
    Triangulation *manifold,
    Cusp *cusp)
{
    double cusp_area;
    Tetrahedron *tet0,
    *nbr_tet;
    VertexIndex v0,
    nbr_v;
    FaceIndex f,
    nbr_f;
    IdealVertex *vertex_stack,
    *initial_vertex,
    *this_vertex,
    *nbr_vertex;
    Permutation gluing;

    /*
     * The plan is to compute an arbitrary cross section of the
     * cusp, and then normalize it to have area  $(3/8)\sqrt{3}$ .
     */

    /*
     * The variable cusp_area will keep track of the area of the
     * cusp cross section. Initialize it to zero.
     */

    cusp_area = 0.0;

    /*
     * Find an ideal vertex belonging to this cusp.

```

```

    */

    find_starting_point(manifold, cusp, &tet0, &v0);

    /*
     * Set the edge_length of some edge of the initial vertex cross section
     * to some arbitrary value, say 1.0, and compute the other two
     * edge_lengths at the initial vertex in terms of it.
     * Set the has_been_set flag to TRUE.
     */

    compute_three_edge_lengths(tet0, v0, !v0, 1.0);

    /*
     * At this point the simplest thing would be to write a
     * recursive function to set the edge_lengths of the remaining
     * vertices. However, recursive functions can cause trouble
     * (e.g. stack/heap collisions) if the recursion is exceptionally
     * deep, so I'll create my own stack explicitly. The stack will
     * contain vertices whose edge_lengths are set, but whose neighbors
     * have not yet been checked. Each ideal vertex experiences the
     * following operations in the following order:
     *
     * (1) edge_lengths are computed
     * (2) has_been_set flag is set to TRUE
     * (3) IdealVertex is put on stack
     * (4) IdealVertex comes off stack
     * (5) area of vertex cross section is added to cusp_area
     * (6) neighboring ideal vertices are checked, and
     *     added to stack as necessary
     * (7) IdealVertex data structure is destroyed
     *
     * Proposition. Each ideal vertex goes onto the stack exactly once.
     * Proof. No ideal vertex can go onto the stack more than once,
     * because once its has_been_set flag is TRUE it is excluded from
     * further consideration. When a vertex comes off the
     * stack its neighbors are considered for addition to the stack,
     * therefore because the cusp is connected all its ideal vertices
     * will eventually go onto the stack.
     */

    initial_vertex = NEW_STRUCT(IdealVertex);
    initial_vertex->tet = tet0;
    initial_vertex->v = v0;
    initial_vertex->next = NULL;

    vertex_stack = initial_vertex;

    while (vertex_stack != NULL)
    {
        /*
         * Pull an IdealVertex off the vertex_stack.
         */
        this_vertex = vertex_stack;
        vertex_stack = vertex_stack->next;

        /*
         * Add the area of the vertex cross section to cusp_area.
         */
        cusp_area += vertex_area(this_vertex);

        /*
         * Check the three neighbors of this IdealVertex.
         */
        for (f = 0; f < 4; f++)
        {
            if (f == this_vertex->v)
                continue;

            /*
             * Locate this_vertex's neighbor by face f.
             */
            gluing = this_vertex->tet->gluing[f];
            nbr_tet = this_vertex->tet->neighbor[f];

```

```

    nbr_v    = EVALUATE(gluing, this_vertex->v);

    /*
     * If the neighbor's edge_lengths have not yet been computed,
     * compute them and add the neighbor to the stack.
     */

    if (nbr_tet->cross_section->has_been_set[nbr_v] == FALSE)
    {
        /*
         * Find the face of nbr_tet which glues to
         * face f of this_vertex->tet.
         */
        nbr_f = EVALUATE(gluing, f);

        /*
         * Set the edge_lengths of vertex nbr_v of Tetrahedron
         * nbr_tet, and set its has_been_set flag to TRUE.
         */
        compute_three_edge_lengths(
            nbr_tet,
            nbr_v,
            nbr_f,
            this_vertex->tet->cross_section->edge_length[this_vertex->v][f]);

        /*
         * Add the neighbor to the stack.
         */
        nbr_vertex = NEW_STRUCT(IdealVertex);
        nbr_vertex->tet    = nbr_tet;
        nbr_vertex->v      = nbr_v;
        nbr_vertex->next   = vertex_stack;
        vertex_stack      = nbr_vertex;
    }

    /*
     * Free this IdealVertex.
     */
    my_free(this_vertex);
}

/*
 * We have constructed a cusp cross section of area cusp_area.
 * To normalize it to have area  $(3/8)\sqrt{3}$ , we must multiply all
 * edge_lengths by  $\sqrt{(3/8)\sqrt{3} / \text{cusp\_area}}$ .
 */

normalize_cusp(manifold, cusp, cusp_area);
}

static void find_starting_point(
    Triangulation *manifold,
    Cusp *cusp,
    Tetrahedron **tet0,
    VertexIndex *v0)
{
    for (*tet0 = manifold->tet_list_begin.next;
        *tet0 != &manifold->tet_list_end;
        *tet0 = (*tet0)->next)

        for (*v0 = 0; *v0 < 4; (*v0)++)

            if ((*tet0)->cusp[*v0] == cusp)

                return;

    /*
     * We should never get to this point.
     */
    uFatalError("find_starting_point", "cusp_cross_sections");
}

```

```

/*
 * compute_three_edge_lengths() sets tet->cross_section->edge_length[v][f]
 * to known_length, computes the remaining two edge_lengths at vertex v
 * in terms of it, and sets the has_been_set flag to TRUE.
 */

void compute_three_edge_lengths(
    Tetrahedron *tet,
    VertexIndex v,
    FaceIndex f,
    double known_length)
{
    double *this_triangle;
    FaceIndex left_face,
              right_face;

    /*
     * For convenience, note which triangle we're working with.
     */

    this_triangle = tet->cross_section->edge_length[v];

    /*
     * Set the given edge_length.
     */

    this_triangle[f] = known_length;

    /*
     * Find the left and right edges of the triangle, corresponding
     * to the left_face and right_face of the Tetrahedron, in the
     * imagery of positioned_tet.h. Work relative to the right_handed
     * Orientation of the Tetrahedron, since that's how the TetShapes
     * are defined.
     */

    left_face = remaining_face[v][f];
    right_face = remaining_face[f][v];

    /*
     * The real part of the logarithmic form of the angle between the
     * near and left faces gives us the log of the ratio of the lengths
     * of the near and left sides of this_triangle, and similarly for
     * the right side.
     */

    this_triangle[left_face] = known_length *
        exp(tet->shape[complete]->cwl[ultimate][edge3_between_faces[f][left_face]].log.
        real);

    this_triangle[right_face] = known_length /
        exp(tet->shape[complete]->cwl[ultimate][edge3_between_faces[f][right_face]].log.
        real);

    /*
     * Set the has_been_set flag to TRUE.
     */

    tet->cross_section->has_been_set[v] = TRUE;
}

static double vertex_area(
    IdealVertex *ideal_vertex)
{
    /*
     * We compute the area of a triangular vertex cross section
     * using Heron's formula
     *
     * 
$$\text{area} = \sqrt{s * (s - a) * (s - b) * (s - c)}$$

     *
     * where a, b and c are the length of the triangle's sides,
     */

```

```

    * and s is the semiperimeter (a + b + c)/2.
    */

double      *this_triangle,
            a,
            b,
            c,
            s,
            area;
VertexIndex v;
FaceIndex   face_a,
            face_b,
            face_c;

v          = ideal_vertex->v;
face_a     = ! v;
face_b     = remaining_face[v][face_a];
face_c     = remaining_face[face_a][v];

this_triangle = ideal_vertex->tet->cross_section->edge_length[v];

a = this_triangle[face_a];
b = this_triangle[face_b];
c = this_triangle[face_c];

s = 0.5 * (a + b + c);

area = safe_sqrt( s * (s - a) * (s - b) * (s - c) );

return area;
}

static void normalize_cusp(
    Triangulation *manifold,
    Cusp          *cusp,
    double        cusp_area)
{
    double        factor;
    Tetrahedron *tet;
    VertexIndex v;
    FaceIndex    f;

    /*
     * The given cusp has area cusp_area.
     * Multiply all the edge_lengths by sqrt( (3/8)sqrt(3) / cusp_area )
     * to normalize the area to (3/8)sqrt(3).
     */

    factor = safe_sqrt(0.375 * ROOT_3 / cusp_area);

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

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

            if (tet->cusp[v] == cusp)

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

                    if (f != v)

                        tet->cross_section->edge_length[v][f] *= factor;
}

void compute_tilts(
    Triangulation *manifold)
{
    Tetrahedron *tet;

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

```

```

        tet = tet->next)

    compute_tilts_for_one_tet(tet);
}

void compute_tilts_for_one_tet(
    Tetrahedron *tet)
{
    double    factor,
              R[4];
    int       i,
              j;

    /*
     * Theorem 2 of "Convex hulls..." gives the tilts in terms
     * of the circumradii. A generalization of the theorem and
     * a cleaner proof appear in "Canonical cell decompositions...".
     *
     * We may compute the circumradius of a triangle in terms
     * of the length of any side c and its opposite angle C,
     * according to the formula
     *
     * 
$$R = c / (2 \sin(C))$$

     *
     * We must be careful in the case of flat (or almost flat)
     * ideal Tetrahedra. As sin(C) goes to zero, the circumradii
     * and the tilts go to infinity. We must take care that the
     * numerical values computed for the circumradii are in
     * proportion to the linear dimensions of the four vertex
     * cross sections. That way even though the numerical values
     * of the tilts will be very large numbers, they will have
     * the correct signs, and the canonization algorithm will proceed
     * correctly. To insure that the circumradii are computed
     * correctly, we use a fixed value for sin(C) (rather than reading
     * the sines of different angles at different vertex cross sections),
     * and we make sure its value exceeds some small epsilon (in
     * particular, we don't want it to be zero).
     */

    /*
     * Compute the circumradii.
     */

    /*
     * Let factor = 2 sin(C), where C is the angle at edge 0.
     * Make sure factor is at least CIRCUMRADIUS_EPSILON.
     */
    factor = 2 * sin(tet->shape[complete]->cwl[ultimate][0].log.imag);
    if (factor < CIRCUMRADIUS_EPSILON)
        factor = CIRCUMRADIUS_EPSILON;

    /*
     * Use the relationship  $R = c / \text{factor}$  (cf. above) to compute
     * the circumradii.
     */
    R[0] = tet->cross_section->edge_length[0][1] / factor;
    R[1] = tet->cross_section->edge_length[1][0] / factor;
    R[2] = tet->cross_section->edge_length[2][3] / factor;
    R[3] = tet->cross_section->edge_length[3][2] / factor;

    /*
     * 95/9/19 JRW
     * Scale the circumradii according to the cusps' displacements.
     * As explained in cusp_neighborhoods.c, a cusp's linear
     * dimensions vary as the exponential of the displacement.
     */
    for (i = 0; i < 4; i++)
        R[i] *= tet->cusp[i]->displacement_exp;

    /*
     * Apply the Tilt Theorem to compute the tilts in terms
     * of the circumradii.
     */
}

```



```
for (i = 0; i < 4; i++)
{
    tet->tilt[i] = 0.0;

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

        if (j == i)

            tet->tilt[i] += R[j];

        else

            tet->tilt[i] -= R[j] *
                cos(tet->shape[complete]->cwl[ultimate][edge3_between_vertices[i][j]]).  ✎
log.imag);
}
}
```