

```

/*
 * Dirichlet_extras.c
 */

#include "kernel.h"
#include "Dirichlet.h"

/*
 * The distances from the origin to points identified by face pairing
 * isometries must agree to within DIST_EPSILON.
 */
#define DIST_EPSILON 1e-3

/*
 * The length of identified edges must agree to within LENGTH_EPSILON.
 */
#define LENGTH_EPSILON 1e-3

/*
 * A vertex is considered ideal iff o3l_inner_product(vertex->x, vertex->x)
 * is within IDEAL_EPSILON of zero. (Recall that vertex->x[0] is always 1.)
 * The choice of IDEAL_EPSILON as 4e-7 is explained below in the
 * documentation in compute_vertex_distance().
 */
#define IDEAL_EPSILON 4e-7

/*
 * The O(3,1) trace of an elliptic involution must be an integer
 * (-2, 0 or 2) to within TRACE_ERROR_EPSILON.
 */
#define TRACE_ERROR_EPSILON 1e-2

/*
 * A neighborhood of a vertex class will be considered nonsingular iff the
 * vertex class's solid angle is at least 4*pi - PI_EPSILON, and a
 * neighborhood of an edge class will be considered nonsingular iff the
 * edge class's dihedral angle is at least 2*pi - PI_EPSILON. We can
 * afford to make PI_EPSILON large, because the next smallest possible
 * value of the solid angle (resp. dihedral angle) is 2*pi (resp. pi).
 */
#define PI_EPSILON 1e-1

/*
 * solid_angles() sets a vertex class's singularity_order to 0
 * when the total solid angle is less than SOLID_ANGLE_EPSILON.
 */
#define SOLID_ANGLE_EPSILON 1e-4

static void face_classes(WEPolyhedron *polyhedron);
static void edge_classes(WEPolyhedron *polyhedron);
static void initialize_edge_classes(WEPolyhedron *polyhedron);
static void find_edge_mates(WEPolyhedron *polyhedron);
static void match_incident_edges(WFace *face);
static void mI_edge_classes(WEPolyhedron *polyhedron, int *count);
static void make_mI_edge_class(WEPolyhedron *polyhedron, WEdge *edge, WEdgeSide side, int index);
static void Sl_edge_classes(WEPolyhedron *polyhedron, int *count);
static void make_Sl_edge_class(WEPolyhedron *polyhedron, WEdge *edge, int index);
static void vertex_classes(WEPolyhedron *polyhedron);
static void create_vertex_class(WEPolyhedron *polyhedron, WVertex *vertex);
static void subdivide_edges_where_necessary(WEPolyhedron *polyhedron);
static void subdivide_faces_where_necessary(WEPolyhedron *polyhedron);
static void cone_face_to_center(WFace *face, WEPolyhedron *polyhedron);
static void bisect_face(WFace *face, WEPolyhedron *polyhedron);
static void delete_face_classes(WEPolyhedron *polyhedron);
static void delete_edge_classes(WEPolyhedron *polyhedron);
static void delete_vertex_classes(WEPolyhedron *polyhedron);
static void dihedral_angles(WEPolyhedron *polyhedron);
static void solid_angles(WEPolyhedron *polyhedron);
static FuncResult vertex_distances(WEPolyhedron *polyhedron);
static void compute_vertex_distance(WVertex *vertex);
static FuncResult edge_distances(WEPolyhedron *polyhedron);
static void compute_edge_distance(WEdge *edge);

```

```

static void      face_distances(WEPolyhedron *polyhedron);
static FuncResult edge_lengths(WEPolyhedron *polyhedron);
static void      compute_edge_length(WEEdge *edge);
static void      compute_approx_volume(WEPolyhedron *polyhedron);
static void      compute_inradius(WEPolyhedron *polyhedron);
static void      compute_outradius(WEPolyhedron *polyhedron);
static void      compute_spine_radius(WEPolyhedron *polyhedron);
static void      attempt_free_edge_removal(WEPolyhedron *polyhedron);
static void      compute_deviation(WEPolyhedron *polyhedron);
static void      compute_geometric_Euler_characteristic(WEPolyhedron *polyhedron);

```

```

FuncResult Dirichlet_bells_and_whistles(
    WEPolyhedron *polyhedron)
{
    /*
     * Compute supplementary information about the Dirichlet domain.
     *
     * Some of the following functions use the results of the others,
     * so please avoid changing their order.
     */

    face_classes(polyhedron);
    edge_classes(polyhedron);
    vertex_classes(polyhedron);

    /*
     * An orbifold's singular set will always be on the Dirichlet
     * domain's boundary. But it may or may not be a subcomplex
     * of the Dirichlet domain's 2-skeleton. It can happen that
     * a 0-cell of the singular set lies at the midpoint of an edge
     * of the Dirichlet domain, or in the interior of a face of the
     * Dirichlet. A 1-cell of the singular set may bisect a face
     * of the Dirichlet domain. We subdivide the Dirichlet domain
     * to contain the singular set as a subcomplex, not just a subspace.
     * If changes are made, we recompute the face_classes(),
     * edge_classes() and vertex_classes().
     *
     * 94/10/4 JRW
     */
    subdivide_edges_where_necessary(polyhedron);
    subdivide_faces_where_necessary(polyhedron);

    dihedral_angles(polyhedron);
    solid_angles(polyhedron);

    if (vertex_distances(polyhedron) == func_failed)
        return func_failed;

    if (edge_distances(polyhedron) == func_failed)
        return func_failed;

    face_distances(polyhedron);

    if (edge_lengths(polyhedron) == func_failed)
        return func_failed;

    compute_approx_volume(polyhedron);
    compute_inradius(polyhedron);
    compute_outradius(polyhedron);

    compute_spine_radius(polyhedron);

    compute_deviation(polyhedron);

    compute_geometric_Euler_characteristic(polyhedron);

    return func_OK;
}

static void face_classes(
    WEPolyhedron *polyhedron)
{

```

```

/*
 * Set the index and hue fields for each face.
 */

WEFace *face;
int count;

/*
 * Initialize all f_class fields to NULL to show they haven't been set.
 */

for (face = polyhedron->face_list_begin.next;
     face != &polyhedron->face_list_end;
     face = face->next)

    face->f_class = NULL;

/*
 * Now go through the list again, and for each face whose f_class has
 * not yet been set, set both it and its mate. (Faces will typically
 * be found consecutively with their mates, but not always, because
 * if two or more pairs of faces are the same distance from the
 * origin, they will be sorted by roundoff error.)
 */

count = 0;

for (face = polyhedron->face_list_begin.next;
     face != &polyhedron->face_list_end;
     face = face->next)

    if (face->f_class == NULL)
    {
        face->f_class = NEW_STRUCT(WEFaceClass);
        face->mate->f_class = face->f_class;

        face->f_class->index = count++;
        face->f_class->hue = index_to_hue(face->f_class->index);

        face->f_class->num_elements = (face->mate == face) ? 1 : 2;

        face->f_class->parity = gl4R_determinant(*face->group_element) > 0.0 ?
                                orientation_preserving :
                                orientation_reversing;

        INSERT_BEFORE(face->f_class, &polyhedron->face_class_end);
    }

/*
 * Set the num_face_classes field.
 */
polyhedron->num_face_classes = count;
}

static void edge_classes(
    WEPolyhedron *polyhedron)
{
    int count;

    /*
     * Initialize all e_class fields to NULL to show they
     * have not yet been set.
     */
    initialize_edge_classes(polyhedron);

    /*
     * Initialize the edge count.
     * We'll pass its address to mI_edge_classes() and Sl_edge_classes()
     * so they can assign indices consistently.
     */
    count = 0;

    /*

```

```

    * Determine which edges are identified to which under the action
    * of the face pairings.
    */
    find_edge_mates(polyhedron);

    /*
    * The link of an edge (in the manifold or orbifold obtained by gluing
    * the Dirichlet domain's matching faces) may be either a circle or
    * an interval mI with mirror endpoints. First find the all mI edge
    * classes. The remaining edge classes must then be circular.
    */
    mI_edge_classes(polyhedron, &count);
    SI_edge_classes(polyhedron, &count);

    /*
    * Record the number of edge classes.
    */
    polyhedron->num_edge_classes = count;
}

static void initialize_edge_classes(
    WEPolyhedron *polyhedron)
{
    WEEdge *edge;
    int i;

    /*
    * Initialize each edge->e_class to NULL to show it hasn't been set.
    * While we're at it, we might as well initialize the neighbor,
    * preserves_direction and preserves_orientation fields as a guard
    * against programmer error.
    */

    for (edge = polyhedron->edge_list_begin.next;
        edge != &polyhedron->edge_list_end;
        edge = edge->next)
    {
        edge->e_class = NULL;

        for (i = 0; i < 2; i++)
        {
            edge->neighbor[i] = NULL;
            edge->preserves_direction[i] = -1;
            edge->preserves_orientation[i] = -1;
        }
    }
}

static void find_edge_mates(
    WEPolyhedron *polyhedron)
{
    WEFace *face;

    /*
    * Initialize the face->matched flags to FALSE.
    */

    for (face = polyhedron->face_list_begin.next;
        face != &polyhedron->face_list_end;
        face = face->next)
    {
        face->matched = FALSE;
    }

    /*
    * For each face which hasn't yet been matched, match it with its
    * mate and fill in the incident WEEdges' neighbor, preserves_direction
    * and preserves_orientation fields.
    */

    for (face = polyhedron->face_list_begin.next;
        face != &polyhedron->face_list_end;
        face = face->next)

```

```

    {
        match_incident_edges(face);

        face->matched      = TRUE;
        face->mate->matched = TRUE;
    }
}

static void match_incident_edges(
    WEFace *face)
{
    O3lVector    *face_vertices,
                 *mate_vertices;
    WEEdge       *edge,
                 *face_edge,
                 *mate_edge;
    WEVertex     *vertex;
    int           count,
                 i,
                 j,
                 offset,
                 best_offset;
    double        min_error,
                 error,
                 diff;
    WEEdgeSide    face_side,
                 mate_side;
    Boolean        traverse_clockwise,
                 sides_preserved,
                 orientation_preserved,
                 direction_preserved;

    /*
     * verify_faces() in Dirichlet_construction.c has already checked
     * that matching faces have the same number of sides. But it's
     * cheap and easy to check again.
     */

    if (face->num_sides != face->mate->num_sides)
        uFatalError("match_incident_edges", "Dirichlet_extras");

    /*
     * Allocate space for the coordinates of this face's vertices,
     * and for the images of face->mate's vertices.
     */

    face_vertices = NEW_ARRAY(face->num_sides, O3lVector);
    mate_vertices = NEW_ARRAY(face->num_sides, O3lVector);

    /*
     * Copy the coordinates of this face's vertices, beginning at the
     * clockwise-most vertex of face->some_edge, and proceeding
     * counterclockwise around the face.
     */

    edge = face->some_edge;
    count = 0;

    do
    {
        vertex = (edge->f[left] == face) ?
                  edge->v[tail] :
                  edge->v[tip];

        o3l_copy_vector(face_vertices[count++], vertex->x);

        edge = (edge->f[left] == face) ?
                edge->e[tip][left] :
                edge->e[tail][right];
    } while (edge != face->some_edge);

    if (count != face->num_sides)

```

```

    uFatalError("match_incident_edges", "Dirichlet_extras");

/*
 * If face->group_element is orientation-preserving, we'll traverse
 * face->mate beginning at the counterclockwise-most vertex of
 * face->mate->some_edge and proceeding clockwise.
 *
 * If face->group_element is orientation-reversing, we'll traverse
 * face->mate beginning at the clockwise-most vertex of
 * face->mate->some_edge and proceeding counterclockwise.
 *
 * To decide whether face->group_element preserves or reverses
 * orientation, check its determinant. The determinant will be
 * +1 or -1, so we needn't worry about roundoff errors.
 *
 * Rather than copy face->mate's vertex coordinates directly,
 * we'll apply the face pairing isometry to them, so they can be
 * compared directly to the coordinates of face's vertices.
 * Note that a vertex's coordinates don't lie on the hyperboloid
 * itself; instead they follow the convention that x[0] == 1.0.
 */

    traverse_clockwise = (gl4R_determinant(*face->group_element) > 0.0);

    edge = face->mate->some_edge;
    count = 0;

do
{
    vertex = (edge->f[traverse_clockwise ? right : left] == face->mate) ?
              edge->v[tail] :
              edge->v[tip];

    o3l_matrix_times_vector(*face->group_element, vertex->x, mate_vertices[count]);
    for (i = 1; i < 4; i++)
        mate_vertices[count][i] /= mate_vertices[count][0];
    mate_vertices[count][0] = 1.0;
    count++;

    edge = traverse_clockwise ?
        (
            edge->f[right] == face->mate ?
            edge->e[tip][right] :
            edge->e[tail][left]
        ) :
        (
            edge->f[left] == face->mate ?
            edge->e[tip][left] :
            edge->e[tail][right]
        );
} while (edge != face->mate->some_edge);

if (count != face->mate->num_sides)
    uFatalError("match_incident_edges", "Dirichlet_extras");

/*
 * face_vertices[] will coincide with mate_vertices[] as sets, but
 * there'll be some offset in the ordering. For example, if the
 * offset is 3 and face->num_sides == 5, then
 *
 *         face_vertices[0] == mate_vertices[3]
 *         face_vertices[1] == mate_vertices[4]
 *         face_vertices[2] == mate_vertices[0]
 *         face_vertices[3] == mate_vertices[1]
 *         face_vertices[4] == mate_vertices[2]
 *
 * Of course the coordinates won't match precisely because of roundoff
 * error, but we don't know just how big the roundoff error will be.
 * So we try all possible values for the offset, and see which one
 * produces the least error. (We compute the error as the sum of the
 * squares of the Euclidean distances from face_vertices[i] to
 * mate_vertices[i + offset] in the projective model.)
 */

```

```

min_error = DBL_MAX;

for (offset = 0; offset < face->num_sides; offset++)
{
    error = 0.0;

    for (i = 0; i < face->num_sides; i++)
        for (j = 1; j < 4; j++)
        {
            diff = face_vertices[i][j]
                - mate_vertices[(i + offset)%face->num_sides][j];
            error += diff * diff;
        }

    if (error < min_error)
    {
        best_offset = offset;
        min_error = error;
    }
}

/*
 * We now know the relative orientation of face and face->mate, and
 * the offset needed to get them to match up. So we can tell their
 * incident edges about each other by setting their neighbor,
 * preserves_direction and preserves_orientation fields. We'll
 * traverse face and face->mate simultaneously, with face_edge and
 * mate_edge recording the edges currently being matched.
 */

/*
 * Set face_edge and mate_edge to the default starting edges.
 */

face_edge = face->some_edge;
mate_edge = face->mate->some_edge;

/*
 * Advance mate_edge to account for the offset.
 */

for (i = 0; i < best_offset; i++)
    mate_edge = traverse_clockwise ?
        (
            mate_edge->f[right] == face->mate ?
            mate_edge->e[tip][right] :
            mate_edge->e[tail][left]
        ) :
        (
            mate_edge->f[left] == face->mate ?
            mate_edge->e[tip][left] :
            mate_edge->e[tail][right]
        );

/*
 * Traverse face and face->mate simultaneously, matching up the
 * corresponding edges.
 */

do
{
    /*
     * Which side of face_edge (left or right) lies on face?
     * Which side of mate_edge (left or right) lies on face->mate?
     */
    face_side = (face_edge->f[left] == face) ? left : right;
    mate_side = (mate_edge->f[left] == face->mate) ? left : right;

    /*
     * Does face_side == mate_side?
     */
    sides_preserved = (face_side == mate_side);
}

```

```

/*
 * When we set traverse_clockwise above, we checked whether the
 * gluing preserves or reverses orientation.
 */
orientation_preserved = traverse_clockwise;

/*
 * In the orientation preserving case, face_edge and mate_edge
 * point in the same direction iff (face_side != mate_side).
 *
 * In the orientation reversing case, face_edge and mate_edge
 * point in the same direction iff (face_side == mate_side).
 */
direction_preserved = (orientation_preserved ^ sides_preserved);

/*
 * Tell face_edge and mate_edge about each other.
 */

face_edge->neighbor[face_side] = mate_edge;
mate_edge->neighbor[mate_side] = face_edge;

face_edge->preserves_sides[face_side] = sides_preserved;
mate_edge->preserves_sides[mate_side] = sides_preserved;

face_edge->preserves_direction[face_side] = direction_preserved;
mate_edge->preserves_direction[mate_side] = direction_preserved;

face_edge->preserves_orientation[face_side] = orientation_preserved;
mate_edge->preserves_orientation[mate_side] = orientation_preserved;

/*
 * Advance face_edge to the next position.
 */
face_edge = (face_edge->f[left] == face) ?
    face_edge->e[tip][left] :
    face_edge->e[tail][right];

/*
 * Advance mate_edge to the next position.
 */
mate_edge = traverse_clockwise ?
    (
        mate_edge->f[right] == face->mate ?
        mate_edge->e[tip][right] :
        mate_edge->e[tail][left]
    ) :
    (
        mate_edge->f[left] == face->mate ?
        mate_edge->e[tip][left] :
        mate_edge->e[tail][right]
    );

} while (face_edge != face->some_edge);

/*
 * Free the local arrays.
 */
my_free(face_vertices);
my_free(mate_vertices);
}

static void mI_edge_classes(
    WEPolyhedron *polyhedron,
    int *count)
{
    WEdge *edge;
    WEdgeSide side;

    /*
     * Look for edges which have not been assigned to edge classes,
     * and which glue to themselves on a single side. Such edges

```



```

    * occur when the link of the edge's midpoint is one of the orbifolds
    * *nn, 2*n or 22n.
    */

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

    for (side = 0; side < 2; side++)      /* side = left, right */

        if (edge->e_class == NULL
            && edge->neighbor[side] == edge
            && edge->preserves_sides[side] == TRUE)

            make_mI_edge_class(polyhedron, edge, side, (*count)++);
}

static void make_mI_edge_class(
    WEPolyhedron    *polyhedron,
    WEEdge          *edge,
    WEEdgeSide      side,
    int             index)
{
    WEEdgeClass *new_class;
    WEEdge      *this_edge,
               *next_edge;
    WEEdgeSide  leading_side;

    /*
     * Allocate and initialize the new WEEdgeClass.
     */
    new_class = NEW_STRUCT(WEEdgeClass);
    new_class->index      = index;
    new_class->hue         = index_to_hue(index);
    new_class->num_elements = 0;
    INSERT_BEFORE(new_class, &polyhedron->edge_class_end);

    /*
     * Start with "edge" and work our way around the edge class.
     *
     * We need to exit the loop at a different point from where we enter,
     * so we must use a "while (TRUE) {}" loop and break from the middle.
     *
     * At each step, this_edge will be the edge currently under
     * consideration, and leading_side will be the side (left or right)
     * where the next_edge is attached.
     */

    this_edge      = edge;
    leading_side    = ! side;

    while (TRUE)
    {
        /*
         * Assign the edge class.
         */
        this_edge->e_class = new_class;

        /*
         * Increment the count.
         */
        new_class->num_elements++;

        /*
         * Which edge is next?
         */
        next_edge = this_edge->neighbor[leading_side];

        /*
         * If next_edge == this_edge, we note the topology of the edge
         * class and then break from the while (TRUE) {} loop.
         */

        if (next_edge == this_edge)

```

```

    {
        /*
         * Check the topology.
         */
        if (edge->preserves_direction[side] == TRUE)
        {
            if (this_edge->preserves_direction[leading_side] == TRUE)
                new_class->link = orbifold_xnn;
            else
                new_class->link = orbifold_2xn;
        }
        else /* edge->preserves_direction[side] == FALSE */
        {
            if (this_edge->preserves_direction[leading_side] == TRUE)
                new_class->link = orbifold_2xn;
            else
                new_class->link = orbifold_22n;
        }

        /*
         * Exit the "while (TRUE) {}" loop.
         */
        break;
    }

    /*
     * We want the edge directions to be consistent whenever possible.
     * So if this_edge and next_edge aren't consistently directed,
     * reverse the direction of next_edge.
     */
    if (this_edge->preserves_direction[leading_side] == FALSE)
        redirect_edge(next_edge, TRUE);

    /*
     * We now know that preserves_direction is TRUE, so
     *
     *     leading_side will change
     *     iff preserves_orientation is FALSE
     *     iff preserves_sides is TRUE
     */
    if (this_edge->preserves_orientation[leading_side] == FALSE)
        leading_side = ! leading_side;

    /*
     * Move on to the next_edge, and continue with the loop.
     */
    this_edge = next_edge;
}

static void S1_edge_classes(
    WEPolyhedron *polyhedron,
    int *count)
{
    WEEdge *edge;

    /*
     * Look for edges which have not been assigned to edge classes.
     */

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

        if (edge->e_class == NULL)

            make_S1_edge_class(polyhedron, edge, (*count)++);
}

static void make_S1_edge_class(
    WEPolyhedron *polyhedron,
    WEEdge *edge,

```

```

    int                index)
{
    WEEdgeClass *new_class;
    WEEdge      *this_edge,
               *next_edge;
    WEEdgeSide  leading_side;

    /*
     * The cases where the link of the edge's midpoint is (*nn), (2*n)
     * or (22n) have already been handled as mI edge classes.
     * Here we treat the case where the link of the midpoint is a sphere
     * or cross surface.
     */

    /*
     * Allocate and initialize the new WEEdgeClass.
     */
    new_class = NEW_STRUCT(WEEdgeClass);
    new_class->index      = index;
    new_class->hue        = index_to_hue(index);
    new_class->num_elements = 0;
    INSERT_BEFORE(new_class, &polyhedron->edge_class_end);

    /*
     * Start with "edge" and work our way around the edge class.
     *
     * We need to exit the loop at a different point from where we enter,
     * so we must use a "while (TRUE) {}" loop and break from the middle.
     *
     * At each step, this_edge will be the edge currently under
     * consideration, and leading_side will be the side (left or right)
     * where the next_edge is attached.
     */

    this_edge      = edge;
    leading_side    = left;

    while (TRUE)
    {
        /*
         * Assign the edge class.
         */
        this_edge->e_class = new_class;

        /*
         * Increment the count.
         */
        new_class->num_elements++;

        /*
         * Which edge is next?
         */
        next_edge = this_edge->neighbor[leading_side];

        /*
         * If the next_edge is the original edge we started with, we
         * note the topology of the edge class and then break from the
         * while (TRUE) {} loop.
         */

        if (next_edge == edge)
        {
            /*
             * Check the topology.
             */
            if (this_edge->preserves_direction[leading_side] == TRUE)
                new_class->link = orbifold_nn; /* sphere */
            else
                new_class->link = orbifold_no; /* cross surface */

            /*
             * Exit the "while (TRUE) {}" loop.
             */
            break;
        }
    }
}

```

```

    }

    /*
     * We want the edge directions to be consistent whenever possible.
     * So if this_edge and next_edge aren't consistently directed,
     * reverse the direction of next_edge.
     */
    if (this_edge->preserves_direction[leading_side] == FALSE)
        redirect_edge(next_edge, TRUE);

    /*
     * We now know that preserves_direction is TRUE, so
     *
     *     leading_side will change
     *     iff preserves_orientation is FALSE
     *     iff preserves_sides is TRUE
     */
    if (this_edge->preserves_orientation[leading_side] == FALSE)
        leading_side = ! leading_side;

    /*
     * Move on to the next_edge, and continue with the loop.
     */
    this_edge = next_edge;
}

}

static void vertex_classes(
    WEPolyhedron    *polyhedron)
{
    WEVertex        *vertex;

    /*
     * Initialize polyhedron->num_vertex_classes to zero.
     */

    polyhedron->num_vertex_classes = 0;

    /*
     * Initialize all vertex->v_class fields to NULL so we can tell which
     * ones have been set and which haven't.
     */

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)

        vertex->v_class = NULL;

    /*
     * Create a vertex class for each vertex which doesn't yet have one,
     * and assign that class to all equivalent vertices.
     */

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)

        if (vertex->v_class == NULL)

            create_vertex_class(polyhedron, vertex);
}

static void create_vertex_class(
    WEPolyhedron    *polyhedron,
    WEVertex        *vertex)
{
    WEVertexClass    *new_class;
    Boolean          progress;
    WEEdge           *edge;
    WEEdgeEnd        which_end;
    WEEdgeSide       which_side;

```

```

WEEdge      *nbr_edge;
WEEdgeEnd    nbr_end;

/*
 * Create the new class.
 * Don't worry about the solid angles for now;
 * they'll be computed later.
 */

new_class      = NEW_STRUCT(WVertexClass);
new_class->index      = polyhedron->num_vertex_classes++;
new_class->hue        = index_to_hue(new_class->index);
new_class->num_elements = 0;
INSERT_BEFORE(new_class, &polyhedron->vertex_class_end);

/*
 * Assign the initial vertex to the new_class.
 */

vertex->v_class = new_class;
new_class->num_elements++;

/*
 * Find all other vertices belong to this class.
 * One could write an "efficient" algorithm to do this -- by carefully
 * locating the given vertex's neighbors and then continuing
 * recursively -- but for any reasonable polyhedron it will be just
 * as fast to simply keep scanning the edge list looking for
 * unassigned neighbors, and the code for this will be much simpler.
 * If this algorithm ever proves to be too slow, we can switch to
 * the more sophisticated approach.
 */

do
{
    /*
     * We'll repeat the loop as long as we keep making progress.
     * Initialize progress to FALSE, and then set it to TRUE if and
     * when we assign the new_class to a previously unclassified vertex.
     */

    progress = FALSE;

    /*
     * Look for edges which identify a new_class vertex to an
     * unassigned vertex.
     */

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

        for (which_end = 0; which_end < 2; which_end++) /* which_end = tail, tip */

            if (edge->v[which_end]->v_class == new_class)

                for (which_side = 0; which_side < 2; which_side++) /* which_side =
left, right */
                {
                    nbr_edge = edge->neighbor[which_side];
                    nbr_end   = edge->preserves_direction[which_side] ?
                                which_end :
                                ! which_end;
                    if (nbr_edge->v[nbr_end]->v_class == NULL)
                    {
                        nbr_edge->v[nbr_end]->v_class = new_class;
                        new_class->num_elements++;
                        progress = TRUE;
                    }
                }
        } while (progress == TRUE);
}

```

```

static void subdivide_edges_where_necessary(
    WEPolyhedron *polyhedron)
{
    Boolean changes_made;
    WEEdge *edge;

    changes_made = FALSE;

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

        switch (edge->e_class->link)
        {
            /*
             * In the following three cases there is a covering
             * transformation which reverses the direction of the edge.
             * The covering transformation fixes the point closest to
             * the origin. We want to split the edge at that point.
             */
            case orbifold_no:
            case orbifold_2xn:
            case orbifold_22n:
                compute_edge_distance(edge);
                split_edge(edge, edge->closest_point_on_edge, FALSE);
                polyhedron->num_vertices++;
                polyhedron->num_edges++;
                changes_made = TRUE;
                break;

            /*
             * In the following two cases the direction of the edge
             * is not reversed, so there is no need to subdivide it.
             */
            case orbifold_nn:
            case orbifold_xnn:
                /*
                 * Do nothing.
                 */
                break;

            default:
                uFatalError("subdivide_edges_where_necessary", "Dirichlet_extras");
        }

    if (changes_made == TRUE)
    {
        delete_face_classes(polyhedron);
        delete_edge_classes(polyhedron);
        delete_vertex_classes(polyhedron);

        face_classes(polyhedron);
        edge_classes(polyhedron);
        vertex_classes(polyhedron);
    }
}

static void subdivide_faces_where_necessary(
    WEPolyhedron *polyhedron)
{
    Boolean changes_made;
    WEFace *face;
    double trace;

    changes_made = FALSE;

    for (face = polyhedron->face_list_begin.next;
        face != &polyhedron->face_list_end;
        face = face->next)

        if (face->mate == face)
        {

```

```

/*
 * A point P in the interior of a 2-cell on the boundary
 * of the Dirichlet domain is equidistant from the basepoint
 * and precisely one of the basepoint's translates. Therefore
 * P is fixed by the identity and at most one other covering
 * transformation. (Recall that SnapPea chooses a basepoint
 * which does not lie in the singular set, and therefore is
 * not fixed by any covering transformation.)
 *
 * Proposition. The non-identity covering transformation may be
 *
 * (1) a reflection through a point,
 * (2) a reflection through a line, or
 * (3) a reflection across a plane.
 *
 * Proof: The classification of isometries in complex_length.c
 * shows that an elliptic isometry of  $H^3$  is a rotation about
 * axis, possibly followed by reflection in a plane orthogonal
 * to that axis. The only such isometries of order two are
 * the three ones listed above. Q.E.D.
 *
 * In case (1) there is an isolated cone point at the
 * center of the face.
 * In case (2) there is an order two cone axis bisecting
 * the face.
 * In case (3) the entire face is a mirror reflector.
 *
 * The three cases may be distinguished by the traces of
 * the covering transformation.
 *
 * In case (1) the trace is -2.
 * In case (2) the trace is 0.
 * In case (3) the trace is +2.
 */

trace = o3l_trace(*face->group_element);

if (fabs(fmod(fabs(trace) + 0.5, 1.0) - 0.5) > TRACE_ERROR_EPSILON)
    uFatalError("subdivide_faces_where_necessary", "Dirichlet_extras");

switch ((int) floor(trace + 0.5))
{
    case -2:
        cone_face_to_center(face, polyhedron);
        changes_made = TRUE;
        break;

    case 0:
        bisect_face(face, polyhedron);
        changes_made = TRUE;
        break;

    case +2:
        /*
         * The whole face is a mirror reflector.
         * No subdivision is needed.
         */
        break;

    default:
        uFatalError("subdivide_faces_where_necessary", "Dirichlet_extras");
}
}

if (changes_made == TRUE)
{
    delete_face_classes(polyhedron);
    delete_edge_classes(polyhedron);
    delete_vertex_classes(polyhedron);

    face_classes(polyhedron);
    edge_classes(polyhedron);
    vertex_classes(polyhedron);
}

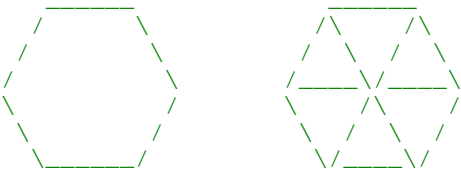
```

```

}

static void cone_face_to_center(
    WEFace      *face,
    WEPolyhedron *polyhedron)
{
    int      old_num_sides;
    WEEdge   **side_edge,
             **radial_edge;
    WEFace    **new_face;
    WEVertex  *central_vertex;
    O3lVector  fixed_point;
    int        i;

    /*
     * Note how many sides the face has before we subdivide.
     */
    old_num_sides = face->num_sides;
    if (old_num_sides % 2 != 0)
        uFatalError("cone_face_to_center", "Dirichlet_extras");

    /*
     * In this case there is no pre-existing function in
     * Dirichlet_construction.c for us to call, so I'll
     * write the low-level code here. The basic idea is
     * to replace
     *
     *           this           with this
     *
     * 
     *
     */

    /*
     * To simplify the subsequent code, reorient the WEEdges so all are
     * directed counterclockwise around the face.
     */
    all_edges_counterclockwise(face, TRUE);

    /*
     * Allocate some arrays to keep track of the edges and faces.
     */
    side_edge   = NEW_ARRAY(old_num_sides, WEEdge *);
    radial_edge = NEW_ARRAY(old_num_sides, WEEdge *);
    new_face    = NEW_ARRAY(old_num_sides, WEFace *);

    /*
     * Record the side_edges.
     */
    {
        WEEdge *edge;
        int     count;

        edge = face->some_edge;
        count = 0;
        do
        {
            side_edge[count++] = edge;
            edge = edge->e[tip][left];
        } while (edge != face->some_edge);

        if (count != old_num_sides)
            uFatalError("cone_face_to_center", "Dirichlet_extras");
    }

    /*
     * Allocate the radial_edges.
     */
    for (i = 0; i < old_num_sides; i++)
    {

```



```

    radial_edge[i]->f[right]    = new_face[ip];

    side_edge[i]->e[tail][left] = radial_edge[im];
    side_edge[i]->e[tip ][left] = radial_edge[i];
    side_edge[i]->f[left]    = new_face[i];

    new_face[i]->some_edge      = side_edge[i];
    new_face[i]->mate           = new_face[io];
    new_face[i]->group_element = NEW_STRUCT(O3lMatrix);
    o3l_copy(*new_face[i]->group_element, *face->group_element);
    new_face[i]->num_sides      = 3;
}

/*
 * Remove the original face.
 *
 * (subdivide_faces_where_necessary()) takes responsibility for
 * freeing its WEFaceClass.)
 */
REMOVE_NODE(face);
my_free(face->group_element);
my_free(face);
face = NULL;

/*
 * Adjust cell counts.
 */
polyhedron->num_vertices++;
polyhedron->num_edges += old_num_sides;
polyhedron->num_faces += old_num_sides - 1;

/*
 * Free the arrays.
 */
my_free(side_edge);
my_free(radial_edge);
my_free(new_face);
}

static void bisect_face(
    WEFace      *face,
    WEPolyhedron *polyhedron)
{
    int    count,
           current_side;
    WEEdge *edge;
    int    old_num_sides;

    /*
     * We want to let cut_face_if_necessary() in Dirichlet_construction.c
     * do the low-level work. We set the incident vertices'
     * which_side_of_plane fields to show where the cut should be made.
     */

    /*
     * To simplify the subsequent code, reorient the WEEdges so all are
     * directed counterclockwise around the face.
     */
    all_edges_counterclockwise(face, TRUE);

    /*
     * Mark the vertices where the order 2 axis meets the
     * face's perimeter by setting their which_side_of_plane
     * fields to 0. (Note that the order 2 axis must meet
     * the perimeter at vertices -- not midpoints of edges --
     * because we've already bisected such edges.)
     */
    /*
     * We assign which_side_of_plane = -1 and which_side_of_plane = +1
     * to appropriate vertices by arbitrarily starting with
     * current_side = -1, and toggling it whenever we pass a vertex with
     * which_side_of_plane = 0.
     */

```

```

    count = 0;
    current_side = -1;

    edge = face->some_edge;
do
{
    WEEdge *next_edge;

    next_edge = edge->e[tip][left];

    if (edge->neighbor[left] == next_edge)
    {
        edge->v[tip]->which_side_of_plane = 0;
        count++;
        current_side = -current_side;
    }
    else
        edge->v[tip]->which_side_of_plane = current_side;

    edge = next_edge;
} while (edge != face->some_edge);

if (count != 2)
    uFatalError("bisect_face", "Dirichlet_extras");

/*
 * Note how many sides the face has before we make the cut.
 */
old_num_sides = face->num_sides;
if (old_num_sides % 2 != 0)
    uFatalError("bisect_face", "Dirichlet_extras");

/*
 * Now we can make the call to cut_face_if_necessary().
 * (Here, of course, the cut will be necessary!)
 */
cut_face_if_necessary(face, FALSE);

/*
 * Adjust num_sides.
 */
face->num_sides =
face->mate->num_sides = (old_num_sides + 2) / 2;

/*
 * Adjust cell counts.
 */
polyhedron->num_edges++;
polyhedron->num_faces++;
}

static void delete_face_classes(
    WEPolyhedron *polyhedron)
{
    WEFaceClass *dead_face_class;
    WEFace *face;

    while (polyhedron->face_class_begin.next != &polyhedron->face_class_end)
    {
        dead_face_class = polyhedron->face_class_begin.next;
        REMOVE_NODE(dead_face_class);
        my_free(dead_face_class);
    }

    for (face = polyhedron->face_list_begin.next;
         face != &polyhedron->face_list_end;
         face = face->next)

        face->f_class = NULL;
}

```

```

static void delete_edge_classes(
    WEPolyhedron    *polyhedron)
{
    WEEdgeClass    *dead_edge_class;
    WEEdge         *edge;

    while (polyhedron->edge_class_begin.next != &polyhedron->edge_class_end)
    {
        dead_edge_class = polyhedron->edge_class_begin.next;
        REMOVE_NODE(dead_edge_class);
        my_free(dead_edge_class);
    }

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

        edge->e_class = NULL;
}

static void delete_vertex_classes(
    WEPolyhedron    *polyhedron)
{
    WEVertexClass    *dead_vertex_class;
    WEVertex         *vertex;

    while (polyhedron->vertex_class_begin.next != &polyhedron->vertex_class_end)
    {
        dead_vertex_class = polyhedron->vertex_class_begin.next;
        REMOVE_NODE(dead_vertex_class);
        my_free(dead_vertex_class);
    }

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)

        vertex->v_class = NULL;
}

static void dihedral_angles(
    WEPolyhedron    *polyhedron)
{
    WEEdgeClass    *edge_class;
    WEEdge         *edge;
    int            i,
                  j;
    O3lMatrix      *m[2];
    O3lVector      normal[2];
    double         length,
                  angle_between_normals;

    /*
     * Initialize the total dihedral angle at each edge class to zero.
     */

    for (    edge_class = polyhedron->edge_class_begin.next;
          edge_class != &polyhedron->edge_class_end;
          edge_class = edge_class->next)

        edge_class->dihedral_angle = 0.0;

    /*
     * Compute the dihedral angle at each edge
     * and add it to the running total for its edge class.
     *
     * Proposition. The dihedral angle between two faces is the angle
     * between their normal vectors.
     *
     * Proof. Change coordinates so that the line of intersection
     * passes through the origin.
     */
}

```

```

for (edge = polyhedron->edge_list_begin.next;
     edge != &polyhedron->edge_list_end;
     edge = edge->next)
{
    /*
     * Compute the outward pointing normal to each face, and normalize
     * its length to one (it's guaranteed to be spacelike).
     */

    for (i = 0; i < 2; i++)
    {
        /*
         * Let m[i] be the group_element at edge->f[i].
         */
        m[i] = edge->f[i]->group_element;

        /*
         * The first column of m[i] gives the image of the origin.
         */
        for (j = 0; j < 4; j++)
            normal[i][j] = (*m[i])[j][0];

        /*
         * Subtract off the coordinates of the basepoint (1, 0, 0, 0)
         * to get an outward pointing normal vector to face i.
         * (To see why this is correct, shift coordinates so that
         * the point midway between the origin and the origin's image
         * under the group_element lies at the origin.)
         */
        normal[i][0] -= 1.0;

        /*
         * Normalize the normal vector to have length one.
         * (And forgive the two different uses of the word "normal".)
         */
        length = safe_sqrt(o3l_inner_product(normal[i], normal[i]));
        for (j = 0; j < 4; j++)
            normal[i][j] /= length;
    }

    /*
     * Use <u, v> = |u| |v| cos(angle) to compute the angle
     * between normal[left] and normal[right].
     * We know |u| = |v| = 1 because we've normalized the normals.
     */
    angle_between_normals = safe_acos(o3l_inner_product(normal[left], normal[right]));

    /*
     * The interior angle is pi minus the exterior angle.
     */
    edge->dihedral_angle = PI - angle_between_normals;

    /*
     * Add this to the total for the edge class.
     */
    edge->e_class->dihedral_angle += edge->dihedral_angle;
}

/*
 * Compute the singularity_order for each edge class as
 * 2pi/dihedral_angle, rounded to the nearest integer.
 */

for (edge_class = polyhedron->edge_class_begin.next;
     edge_class != &polyhedron->edge_class_end;
     edge_class = edge_class->next)

    edge_class->singularity_order = (int) floor((TWO_PI / edge_class->dihedral_angle) +
0.5);
}

static void solid_angles(

```

```

    WEPolyhedron    *polyhedron)
{
    WEVertex        *vertex;
    WEEdge          *edge;
    WEEdgeEnd       which_end;
    WEVertexClass   *vertex_class;

    /*
     * Compute the solid angle at each vertex.
     *
     * The solid angle is the total curvature of the link of the vertex.
     * For a finite vertex in a manifold, the link will be a 2-sphere,
     * and the total solid angle in the vertex class will be 4pi (orbifolds
     * admit other possibilities). For an ideal vertex in a manifold,
     * the link will be a torus or Klein bottle, and the total solid
     * angle will be zero (orbifolds admit other possible links, but all
     * will have zero Euler characteristic and zero total solid angle).
     *
     * Use the formula
     *
     *      solid angle = (sum of incident dihedral angles) - (n - 2)pi
     *
     * Computationally, the plan is to first initialize the solid angle
     * at each vertex to 2pi, then add in (dihedral angle - pi) for each
     * incident edge.
     */

    /*
     * Initialize each solid angle to 2pi.
     */

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)

        vertex->solid_angle = TWO_PI;

    /*
     * Go down the list of edges, adding (dihedral angle - pi) to the
     * solid angles of the incident vertices.
     */

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

        for (which_end = 0; which_end < 2; which_end++) /* which_end = tail, tip */

            edge->v[which_end]->solid_angle += edge->dihedral_angle - PI;

    /*
     * Initialize the total solid angle at each vertex class to zero.
     */

    for (vertex_class = polyhedron->vertex_class_begin.next;
         vertex_class != &polyhedron->vertex_class_end;
         vertex_class = vertex_class->next)

        vertex_class->solid_angle = 0.0;

    /*
     * Add the solid angle at each vertex to the total for its class.
     */

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)

        vertex->v_class->solid_angle += vertex->solid_angle;

    /*
     * Compute the singularity_order for each vertex class as
     * 4pi/solid_angle, rounded to the nearest integer.

```

```

    * 94/10/2 JRW
    *
    * Set vertex_class->singularity_order to zero for ideal vertices.
    * The vertex_class->ideal field hasn't yet been set, so we must
    * decide whether the vertex is ideal based on the solid angle.
    * 96/1/4 JRW
    */

for (    vertex_class = polyhedron->vertex_class_begin.next;
        vertex_class != &polyhedron->vertex_class_end;
        vertex_class = vertex_class->next)
{
    if (vertex_class->solid_angle > SOLID_ANGLE_EPSILON)
        vertex_class->singularity_order = (int) floor((FOUR_PI / vertex_class->
solid_angle) + 0.5);
    else
        vertex_class->singularity_order = 0;
}
}

static FuncResult vertex_distances(
    WEPolyhedron    *polyhedron)
{
    WEVertex        *vertex;
    WEVertexClass    *vertex_class;

    /*
    * Compute the distances to the individual vertices.
    */

    for (vertex = polyhedron->vertex_list_begin.next;
        vertex != &polyhedron->vertex_list_end;
        vertex = vertex->next)

        compute_vertex_distance(vertex);

    /*
    * Initialize the dist field in the vertex class to zero.
    * Initialize min_dist to INFINITE_DISTANCE and max_dist to zero.
    */

    for (    vertex_class = polyhedron->vertex_class_begin.next;
        vertex_class != &polyhedron->vertex_class_end;
        vertex_class = vertex_class->next)
    {
        vertex_class->dist      = 0.0;
        vertex_class->min_dist  = INFINITE_DISTANCE;
        vertex_class->max_dist  = 0.0;
    }

    /*
    * Initialize the global vertex counts to zero.
    */

    polyhedron->num_finite_vertices = 0;
    polyhedron->num_ideal_vertices  = 0;

    polyhedron->num_finite_vertex_classes = 0;
    polyhedron->num_ideal_vertex_classes  = 0;

    /*
    * Use the dist field to record the sum of the distances of the
    * individual vertices.
    *
    * Note the minimum and maximum values.
    *
    * Count the finite and ideal vertices.
    *
    * Note whether the vertex class is ideal. (If some vertices were
    * ideal and some weren't, the error would be caught when comparing
    * vertex_class->min_dist and vertex_class->max_dist below. A finite
    * vertex will have a distance of at most about 17, as explained in
    * compute_vertex_distance() below.)
    */

```

```

    */

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)
    {
        vertex->v_class->dist += vertex->dist;

        if (vertex->dist < vertex->v_class->min_dist)
            vertex->v_class->min_dist = vertex->dist;

        if (vertex->dist > vertex->v_class->max_dist)
            vertex->v_class->max_dist = vertex->dist;

        if (vertex->ideal == FALSE)
            polyhedron->num_finite_vertices++;
        else
            polyhedron->num_ideal_vertices++;

        vertex->v_class->ideal = vertex->ideal;
    }

    /*
     * For each vertex class, divide the sum of the individual distances
     * by the number of vertices in the class to get the average distance.
     *
     * Check that the minimum and maximum values are sufficiently close.
     *
     * Increment num_finite_vertex_classes or num_ideal_vertex_classes,
     * as appropriate.
     */

    for ( vertex_class = polyhedron->vertex_class_begin.next;
         vertex_class != &polyhedron->vertex_class_end;
         vertex_class = vertex_class->next)
    {
        vertex_class->dist /= vertex_class->num_elements;

        if (vertex_class->max_dist - vertex_class->min_dist > DIST_EPSILON)
            return func_failed;

        if (vertex_class->ideal == FALSE)
            polyhedron->num_finite_vertex_classes++;
        else
            polyhedron->num_ideal_vertex_classes++;
    }

    /*
     * A quick error check, just to be safe.
     */

    if (polyhedron->num_finite_vertex_classes
        + polyhedron->num_ideal_vertex_classes
        != polyhedron->num_vertex_classes)

        uFatalError("vertex_distances", "Dirichlet_extras");

    return func_OK;
}

static void compute_vertex_distance(
    WEVertex *vertex)
{
    /*
     * Compute the distance from the vertex to the origin,
     * and decide whether the vertex is ideal.
     *
     * For simplicity, consider a point d units from the origin on the
     * 1-dimensional "hyperbolic line" in (1,1)-dimensional Minkowski space.
     * The analysis for a point in 3-dimensional hyperbolic space in
     * (3,1)-dimensional Minkowski space is essentially the same, but
     * messier to write down. The point will have coordinates
     * (cosh d, sinh d). The first coordinate is timelike, the second

```



```

*    spacelike.
*
*    Recall that all vertices have been normalized to have x[0] == 1.0.
*    Normalizing (cosh d, sinh d) to x[0] == 1 gives coordinates
*
*           (cosh d, sinh d)
*    ----- = (1, tanh d) = x[1].
*           cosh d
*
*    The squared norm of (cosh d, sinh d) is -1, so the squared norm of
*    (1, tanh d) will be -1/(cosh d)^2. The latter may be computed
*    directly as <x[], x[]>, so we can turn things around to solve
*    for cosh d.
*
*           <x[], x[]> = -1/(cosh d)^2
*
*           cosh d = sqrt( -1 / <x[], x[]> )
*
*    Computationally speaking, the vertex will appear ideal iff the
*    square of its norm cannot be distinguished from zero. How far will
*    the vertex be from the origin when this occurs? It's easy to find
*    an approximation to tanh d for large d:
*
*           e^d - e^-d      1 - e^-2d
*    tanh d = ----- = ----- ~ 1 - 2 e^-2d
*           e^d + e^-d      1 + e^-2d
*
*    The squared norm of (1, 1 - 2 e^-2d) is
*
*           -1 + (1 - 2 e^-2d)^2 ~ -1 + (1 - 4 e^-2d)
*
*    This number ceases to be computable when -1 and (1 - 4 e^-2d)
*    become numerically indistinguishable. Of course numerical accuracy
*    begins to suffer long before that point. For what value of d
*    does this occur? On a 680x0 Macintosh, DBL_EPSILON is about 1e-19
*    (on most other platforms -- where doubles have 6-byte rather than
*    8-byte mantissas -- DBL_EPSILON will be more like 7e-15). To get
*    any reasonable accuracy, we'd need to have 4 e^-2d greater than,
*    say, 1e5 * DBL_EPSILON. This works out to about d = 17 on the
*    680x0 Mac, or d = 11 on other platforms.
*
*    The d = 17 (or d = 11) estimate is the farthest vertex distance we
*    could possibly hope to compute. In practice the vertex coordinates
*    won't be known to full accuracy, so ideal vertices may appear to
*    be closer. For example, some ideal vertices of L110123 appear at
*    distance d = 14 on a Mac. To be safe, we'll consider all vertices
*    at distance d > 8 to be ideal. This gives 4 e^-2d ~ 4e-7.
*
*    These considerations lead us to declare the vertex to be ideal iff
*    its squared norm (which is a negative number) is greater than
*    - IDEAL_EPSILON = -4e-7.
*/

double    norm_squared;

norm_squared = o3l_inner_product(vertex->x, vertex->x);

if (norm_squared < - IDEAL_EPSILON)
{
    vertex->dist    = arccosh( safe_sqrt( -1.0 / norm_squared ) );
    vertex->ideal    = FALSE;
}
else
{
    vertex->dist    = INFINITE_DISTANCE;
    vertex->ideal    = TRUE;
}

}

static FuncResult edge_distances(
    WEPolyhedron    *polyhedron)
{
    WEEdge          *edge;

```

```

WEEdgeClass *edge_class;

/*
 * Compute the distances to the individual edges.
 */

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

    compute_edge_distance(edge);

/*
 * Initialize the dist_line_to_origin and dist_edge_to_origin fields
 * in the edge class to zero. Initialize min_line_dist to
 * INFINITE_DISTANCE and max_line_dist to zero.
 */

for (    edge_class = polyhedron->edge_class_begin.next;
        edge_class != &polyhedron->edge_class_end;
        edge_class = edge_class->next)
{
    edge_class->dist_line_to_origin = 0.0;
    edge_class->dist_edge_to_origin = 0.0;

    edge_class->min_line_dist      = INFINITE_DISTANCE;
    edge_class->max_line_dist      = 0.0;
}

/*
 * Use the distance fields to record the sum of the distances to the
 * individual edges. Also note the minimum and maximum values.
 */

for (edge = polyhedron->edge_list_begin.next;
     edge != &polyhedron->edge_list_end;
     edge = edge->next)
{
    edge->e_class->dist_line_to_origin += edge->dist_line_to_origin;
    edge->e_class->dist_edge_to_origin += edge->dist_edge_to_origin;

    if (edge->dist_line_to_origin < edge->e_class->min_line_dist)
        edge->e_class->min_line_dist = edge->dist_line_to_origin;

    if (edge->dist_line_to_origin > edge->e_class->max_line_dist)
        edge->e_class->max_line_dist = edge->dist_line_to_origin;
}

/*
 * For each edge class, divide the sum of the individual distances
 * by the number of edges in the class to get the average distances.
 *
 * Check that the minimum and maximum values are sufficiently close.
 */

for (    edge_class = polyhedron->edge_class_begin.next;
        edge_class != &polyhedron->edge_class_end;
        edge_class = edge_class->next)
{
    edge_class->dist_line_to_origin /= edge_class->num_elements;
    edge_class->dist_edge_to_origin /= edge_class->num_elements;

    if (edge_class->max_line_dist - edge_class->min_line_dist > DIST_EPSILON)
        return func_failed;
}

return func_OK;
}

static void compute_edge_distance(
WEEdge *edge)
{
    O31Vector    p[2],

```

```

        v[2],
        w,
        u,
        component;
double    length,
        projection,
        c[3],
        u_coord,
        p0_coord,
        p1_coord,
        basepoint[4] = {1.0, 0.0, 0.0, 0.0};

/*
 * We want to find the minimum distance from the basepoint to the line
 * containing the given edge, and decide whether that minimum occurs
 * within the edge itself. The basepoint and the line lie in a plane
 * in  $H^3$  (the plane is not unique if the line passes through the
 * basepoint, but our algorithm works correctly in that case too).
 * The plane in  $H^3$  determines a 3-dimensional subspace of Minkowski
 * space spanned by edge->v[tail], edge->v[tip] and the basepoint.
 * We will restrict our attention (and our sketches, which you should
 * make as you read along) to that 3-dimensional subspace. The
 * endpoints edge->v[tail] and edge->v[tip] may be either finite
 * vertices (timelike vectors) or ideal vertices (lightlike vectors)
 * independently of one another.
 */

o3l_copy_vector(p[0], edge->v[tail]->x);
o3l_copy_vector(p[1], edge->v[tip]->x);

/*
 * To avoid fussing over whether the endpoints are finite or ideal,
 * we'll switch to a more convenient basis. Define
 *
 *          v[0] = p[1] + p[0]
 *          v[1] = p[1] - p[0]
 */

o3l_vector_sum (p[1], p[0], v[0]);
o3l_vector_diff(p[1], p[0], v[1]);

/*
 * Lemma. v[0] is timelike.
 *
 * Proof. Draw p[0] with its tail at the basepoint. Its tip lies
 * inside (resp. on) the forward light cone when p[0] is a finite
 * (resp. ideal) vertex. Now draw p[1] with its tail at the tip of
 * p[0], and draw a forward light cone centered at the p[1]'s tail.
 * If p[0] is timelike, then the forward lightcone at p[1]'s tail must
 * lie completely within the forward lightcone at p[0]'s tail, and
 * therefore p[0] + p[1] must be timelike. If p[0] is lightlike,
 * then the forward lightcone at p[1]'s tail intersects the forward
 * lightcone at p[0]'s tail along the ray determined by p[0]. But
 * p[0] and p[1] represent distinct points in  $H^3$ , so p[1] cannot
 * also lie along that ray. Therefore p[0] + p[1] must be timelike.
 * Q.E.D.
 *
 * Lemma. v[1] is spacelike.
 *
 * Proof. We've normalized the x[] coordinates of each vertex to
 * have x[0] == 1. So v[1][0] == 0. Q.E.D.
 *
 * Note: The first Lemma expresses a general fact about points in
 * Minkowski space. The second Lemma relies on our normalization
 * convention.
 */

/*
 * Normalize v[0] to unit length.
 */
length = safe_sqrt( - o3l_inner_product(v[0], v[0]) );
o3l_constant_times_vector(1.0/length, v[0], v[0]);

/*

```

```

    *   Make v[1] orthogonal to v[0].
    */
projection = - o3l_inner_product(v[0], v[1]);
o3l_constant_times_vector(projection, v[0], component);
o3l_vector_diff(v[1], component, v[1]);

/*
    *   Normalize v[1] to unit length.
    */
length = safe_sqrt(o3l_inner_product(v[1], v[1]));
o3l_constant_times_vector(1.0/length, v[1], v[1]);

/*
    *   Express the basepoint as a linear combination
    *   c[0]v[0] + c[1]v[1] + c[2]v[2], where v[2] is a spacelike unit
    *   vector orthogonal to both v[0] and v[1].  (If the basepoint lies
    *   in the plane spanned by v[0] and v[1], then c[2] = 0 and
    *   v[2] is undefined.)
    */

o3l_copy_vector(w, basepoint);

c[0] = - o3l_inner_product(w, v[0]);
o3l_constant_times_vector(c[0], v[0], component);
o3l_vector_diff(w, component, w);

c[1] =  o3l_inner_product(w, v[1]);
o3l_constant_times_vector(c[1], v[1], component);
o3l_vector_diff(w, component, w);

c[2] = safe_sqrt(o3l_inner_product(w, w));

/*
    *   If c[2] == 0, then the basepoint = c[0]v[0] + c[1]v[1] actually lies
    *   on the given line, so the distance is zero.
    *
    *   Otherwise, consider the 2-plane in Minkowski space spanned by the
    *   basepoint and v[2].  (The vectors w and v[2] cannot be colinear
    *   because one is timelike and the other spacelike.)  This 2-plane
    *   defines a line in  $H^3$  which passes through w and is orthogonal to
    *   the given line (proof: rotate coordinates so that the 2-plane is
    *   vertical in your picture and v[2] remains horizontal).  It follows
    *   that the distance from the point to the line is  $\sinh(c[2])$ , and the
    *   point of closest approach is c[0]v[0] + c[1]v[1].
    */

/*
    *   Compute u = c[0]v[0] + c[1]v[1] = basepoint - w and
    *   normalize the zeroth coordinate to one.
    */
o3l_vector_diff(basepoint, w, u);
o3l_constant_times_vector(1.0/u[0], u, u);
o3l_copy_vector(edge->closest_point_on_line, u);

/*
    *   Record the distance from the basepoint to the line.
    */
edge->dist_line_to_origin = arcsinh(c[2]);

/*
    *   u lies between p[0] and p[1] as points in  $H^3$ 
    *
    *   iff u lies between p[0] and p[1] as points in the projective model
    *   (i.e. projected into the hyperplane with zeroth coordinate one)
    *
    *   iff the v[1]-coordinate of u lies between
    *   the v[1]-coordinates of p[0] and p[1].
    */

u_coord = o3l_inner_product(v[1], u);
p0_coord = o3l_inner_product(v[1], p[0]);
p1_coord = o3l_inner_product(v[1], p[1]);

/*

```

```

    * Technical note: The construction of v[1] guarantees that the
    * v[1]-coordinate of p[1] exceeds that of p[0].
    */
    if (p0_coord >= p1_coord)
        uFatalError("compute_edge_distance", "Dirichlet_extras");

    if (u_coord < p0_coord)
    {
        o3l_copy_vector(edge->closest_point_on_edge, p[0]);
        edge->dist_edge_to_origin = edge->v[tail]->dist;
    }
    else if (u_coord > p1_coord)
    {
        o3l_copy_vector(edge->closest_point_on_edge, p[1]);
        edge->dist_edge_to_origin = edge->v[tip]->dist;
    }
    else
    {
        o3l_copy_vector(edge->closest_point_on_edge, edge->closest_point_on_line);
        edge->dist_edge_to_origin = edge->dist_line_to_origin;
    }
}

static void face_distances(
    WEPolyhedron *polyhedron)
{
    /*
    * Compute the distance from the origin to the face plane.
    * The point closest to the origin may or may not lie on the face itself.
    *
    * The first column of the group_element gives the image of the
    * origin (1, 0, 0, 0) under the face pair isometry. Hence
    * group_element[0][0] equals cosh(2*dist).
    */

    WEFace *face;
    int i;
    O3lVector the_image,
              the_sum,
              the_midpoint;
    O3lVector the_origin = {1.0, 0.0, 0.0, 0.0};

    for (face = polyhedron->face_list_begin.next;
         face != &polyhedron->face_list_end;
         face = face->next)
    {
        /*
        * Compute the distance to the face plane.
        */

        face->dist = 0.5 * arccosh((*face->group_element)[0][0]);
        face->f_class->dist = face->dist;

        /*
        * Find the point on the face plane which realizes the distance.
        */

        /*
        * Find the image of the origin under the action of the group_element.
        */
        for (i = 0; i < 4; i++)
            the_image[i] = (*face->group_element)[i][0];

        /*
        * Find the point midway between the origin (1,0,0,0) and the_image.
        * (Conceptually we should think of the midpoint as being halfway
        * between the_origin and the_image in H^3 itself, not in the
        * ambient E^(3,1). But either way determines the same ray through
        * (0,0,0,0). Proof: visualize the construction in a coordinate
        * system in which the_midpoint lies on the positive
        * 0-th coordinate axis.)
        */
    }
}

```

```

    o3l_vector_sum(the_origin, the_image, the_sum);
    o3l_constant_times_vector(0.5, the_sum, the_midpoint);

    /*
     * Normalize the_midpoint to have zeroth coordinate 1.0.
     * (Normalizing it to have length one might make more sense,
     * but we want to be consistent with how other points are recorded.)
     */
    o3l_constant_times_vector(
        1.0 / the_midpoint[0],
        the_midpoint,
        face->closest_point);
}
}

```

```

static FuncResult edge_lengths(
    WEPolyhedron *polyhedron)
{
    WEEdge *edge;
    WEEdgeClass *edge_class;

    /*
     * Compute the lengths of the individual edges.
     */

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

        compute_edge_length(edge);

    /*
     * Initialize each edge class's length field to zero.
     * Initialize min_length to INFINITE_LENGTH and max_length to zero.
     */

    for (edge_class = polyhedron->edge_class_begin.next;
         edge_class != &polyhedron->edge_class_end;
         edge_class = edge_class->next)
    {
        edge_class->length = 0.0;

        edge_class->min_length = INFINITE_LENGTH;
        edge_class->max_length = 0.0;
    }

    /*
     * Use the length field to record the sum of the lengths of the
     * individual edges. Also note the minimum and maximum values.
     */

    for (edge = polyhedron->edge_list_begin.next;
         edge != &polyhedron->edge_list_end;
         edge = edge->next)
    {
        edge->e_class->length += edge->length;

        if (edge->length < edge->e_class->min_length)
            edge->e_class->min_length = edge->length;

        if (edge->length > edge->e_class->max_length)
            edge->e_class->max_length = edge->length;
    }

    /*
     * For each edge class, divide the sum of the individual lengths
     * by the number of edges in the class to get the average length.
     *
     * Check that the minimum and maximum values are sufficiently close.
     */

    for (edge_class = polyhedron->edge_class_begin.next;
         edge_class != &polyhedron->edge_class_end;

```

```

        edge_class = edge_class->next)
    {
        edge_class->length /= edge_class->num_elements;

        if (edge_class->max_length - edge_class->min_length > LENGTH_EPSILON)
            return func_failed;
    }

    return func_OK;
}

static void compute_edge_length(
    WEEdge *edge)
{
    if (edge->v[tail]->dist == INFINITE_DISTANCE
        || edge->v[tip]->dist == INFINITE_DISTANCE)

        edge->length = INFINITE_LENGTH;

    else

        edge->length = arccosh(
            -o3l_inner_product(edge->v[tail]->x, edge->v[tip]->x)
            /
            (
                safe_sqrt(-o3l_inner_product(edge->v[tail]->x, edge->v[tail]->x))
                * safe_sqrt(-o3l_inner_product(edge->v[tip]->x, edge->v[tip]->x))
            ));
}

static void compute_approx_volume(
    WEPolyhedron *polyhedron)
{
    /*
     * The plan is to decompose the Dirichlet domain into "birectangular
     * tetrahedra", whose volumes may be computed using the formula in
     *
     * E. B. Vinberg, Ob'emy neevklidovykh mnogogrannikov,
     * Uspekhi Matematicheskix Nauk, May(?) 1993, 17-46.
     *
     * Each birectangular tetrahedron has vertices at
     *
     * (1) the origin (1,0,0,0),
     * (2) a point which realizes the minimum distance from a face
     * plane to the origin (this point may or may not lie within
     * the face itself),
     * (3) a point which realizes the minimum distance from one of
     * the face's edges to the origin (this point may or may not
     * lie within the edge itself), and
     * (4) one of the edge's endpoints.
     *
     * I recommend that you make yourself a sketch to see how the above
     * definition serves to divide the Dirichlet domain into birectangular
     * tetrahedra. The fact that the points in (2) and (3) minimize
     * the distance from a face or edge to the origin insures that all
     * the necessary right angles are present.
     *
     * If some of the points in (2) and (3) lie outside their respective
     * faces and edges, then some of the birectangular tetrahedra will
     * be negatively oriented. But if we keep track of which are
     * positively oriented and which are negatively oriented, we can still
     * compute a correct volume. Note that Vinberg's formula does not
     * "automatically" work for negatively oriented tetrahedra, which is
     * why we must keep track of the orientations ourselves.
     */

    double total_volume,
           tetrahedron_volume;
    WEEdge *edge;
    int i,
        j,
        k;

```

```

Boolean      nominal_orientation,
             actual_orientation;

/*
 * The {a, b, c, d} correspond to Vinberg's notation.
 */
O31Vector    a,          /* at vertex   = (4) above */
             b,          /* on edge    = (3) above */
             c,          /* on face    = (2) above */
             d;          /* at origin  = (1) above */

GL4RMatrix   abcd;
O31Vector    origin = {1.0, 0.0, 0.0, 0.0};

o31_copy_vector(d, origin);

total_volume = 0.0;

for (edge = polyhedron->edge_list_begin.next;
     edge != &polyhedron->edge_list_end;
     edge = edge->next)
{
    o31_copy_vector(b, edge->closest_point_on_line);

    for (i = 0; i < 2; i++)      /* i = left, right */
    {
        o31_copy_vector(c, edge->f[i]->closest_point);

        for (j = 0; j < 2; j++) /* j = tail, tip */
        {
            o31_copy_vector(a, edge->v[j]->x);

            /*
             * If the tetrahedron's actual orientation matches its
             * nominal orientation, we add its volume to the total.
             * Otherwise we subtract it.
             */

            /*
             * We don't have to strain our brains figuring out which
             * orientation should be called positive and which should
             * be called negative. All that matters is that the
             * nominal_orientation and actual_orientation are computed
             * consistently. That will ensure that the we end up with
             * either the true volume or its negative. (If we end
             * up with its negative, I'll come back and change the
             * definition of the nominal_orientation to the opposite
             * of what it was before.)
             */

            /*
             * The nominal_orientation toggles if we toggle i
             * (leaving j fixed) or toggle j (leaving i fixed).
             */
            nominal_orientation = (i != j);

            /*
             * The determinant toggles when the actual_orientation
             * toggles, so we may use the former to compute the latter.
             */
            for (k = 0; k < 4; k++)
            {
                abcd[0][k] = a[k];
                abcd[1][k] = b[k];
                abcd[2][k] = c[k];
                abcd[3][k] = d[k];
            }
            actual_orientation = (gl4R_determinant(abcd) > 0.0);

            tetrahedron_volume = birectangular_tetrahedron_volume(a, b, c, d);

            if (nominal_orientation == actual_orientation)
                total_volume += tetrahedron_volume;
            else
                total_volume -= tetrahedron_volume;
        }
    }
}

```



```

    }
}

polyhedron->approximate_volume = total_volume;
}

static void compute_inradius(
    WEPolyhedron *polyhedron)
{
    /*
     * Definition. The "inradius" is the radius of the largest sphere
     * centered at the basepoint which may be inscribed in the Dirichlet
     * domain.
     *
     * Definition. A "face plane" is a plane containing a face of
     * the Dirichlet domain.
     *
     * Proposition. The inradius is the minimum distance from the
     * basepoint to a face plane.
     *
     * Comment. We care only about the distance from the origin to the
     * face plane. We don't care whether that minimum occurs within the
     * face itself.
     *
     * Proof. The Dirichlet domain is the intersection of the halfspaces
     * determined by the face planes. Therefore a sphere centered at the
     * basepoint will be contained in the Dirichlet domain iff it is
     * contained in all the aforementioned halfspaces. The sphere will be
     * contained in all the aforementioned halfspaces iff its radius is at
     * most the distance from the origin to the closest face plane. Q.E.D.
     */

    WEFace *face;
    double min_value;

    /*
     * The distance from the origin to a face plane is
     * 0.5 * arccosh(face->group_element[0][0]). So we look for the
     * minimum value of face->group_element[0][0].
     */

    min_value = INFINITE_RADIUS;

    for (face = polyhedron->face_list_begin.next;
         face != &polyhedron->face_list_end;
         face = face->next)

        if ((*face->group_element)[0][0] < min_value)

            min_value = (*face->group_element)[0][0];

    /*
     * Convert min_value to the true hyperbolic distance.
     */

    polyhedron->inradius = 0.5 * arccosh(min_value);
}

static void compute_outradius(
    WEPolyhedron *polyhedron)
{
    /*
     * The Dirichlet domain is convex, so the outradius will be
     * the maximum distance from a vertex to the origin.
     */

    WEVertex *vertex;
    double max_projective_distance,
           projective_distance;

    /*

```

```

    * First find the maximum distance from the basepoint (1, 0, 0, 0)
    * to the point (1, x, y, z) relative to the Euclidean metric of
    * the projective model. (Actually we'll compute the square of
    * the distance.)
    */

max_projective_distance = 0.0;

for (vertex = polyhedron->vertex_list_begin.next;
     vertex != &polyhedron->vertex_list_end;
     vertex = vertex->next)
{
    /*
    * We assume the vertex->ideal field has already been set.
    * If this is an ideal vertex, the outradius is infinite
    * and we're done.
    */
    if (vertex->ideal == TRUE)
    {
        polyhedron->outradius = INFINITE_RADIUS;
        return;
    }

    /*
    * Compute the squared Euclidean distance to the origin
    * in the projective model.
    */
    projective_distance = vertex->x[1] * vertex->x[1]
                        + vertex->x[2] * vertex->x[2]
                        + vertex->x[3] * vertex->x[3];
    if (projective_distance > max_projective_distance)
        max_projective_distance = projective_distance;
}

/*
* Convert the squared projective distance to the true hyperbolic
* distance. Let d denote the (unsquared) projective distance.
* The true hyperbolic distance will be the same as the distance
* from (1, 0) to (1, d) in the 1-dimensional projective model.
* To compute that distance, transfer the points to the hyperbolic line
*  $H^1 = \{(y, x) \mid -y^2 + x^2 = -1\}$  in the Minkowski space model.
*
* (1, 0) maps to (1, 0)
* (1, d) maps to (1/sqrt(1 - d^2), d/sqrt(1 - d^2))
*
* Recall that for two points u and v in the Minkowski space model
* of  $H^n$ ,  $\cosh(\text{dist}(u, v)) = -\langle u, v \rangle$ . So the distance between the two
* above points is therefore  $\text{arccosh}(1/\sqrt{1 - d^2})$ .
*/
polyhedron->outradius = arccosh( 1.0 / safe_sqrt(1.0 - max_projective_distance) );
}

static void compute_spine_radius(
    WEPolyhedron *polyhedron)
{
    /*
    * Parts of the following documentation appear in the paper
    *
    * C. Hodgson and J. Weeks, Symmetries, isometries and length
    * spectra of closed hyperbolic 3-manifolds, to appear in
    * Experimental Mathematics.
    */

    /*
    * Definition. The "spine radius" is the infimum of the radii (measured
    * from the origin) of all spines dual to the Dirichlet domain.
    *
    * Definition (local to the following proposition). The "maximin
    * edge distance" is the maximum over all edges of the minimum distance
    * from the edge to the origin. (Computationally, it's the maximum
    * value of dist_edge_to_origin over all edge classes.)
    *
    * Proposition. The spine radius equals the maximin edge distance.
    */
}

```

```

*
* Proof. Any spine dual to the Dirichlet domain must intersect every
* edge, so the spine radius is greater than or equal to the maximin
* edge distance. It remains to show that for any epsilon greater than
* zero, we can construct a spine whose radius is within epsilon of the
* maximin edge distance.
*
* Step 1. On each edge, mark the point closest to the origin.
*         If that point is at an endpoint, displace it a distance
*         epsilon into the interior of the edge. Note that
*         (a) the edge identifications respect the marked points, and
*         (b) the marked points all lie within the maximin edge
*             distance plus epsilon of the origin.
*
* Step 2. On each face, mark the point closest to the origin.
*         If that point is on the boundary, displace it a distance
*         epsilon into the interior of the face. Note that
*         (a) the face identifications respect the marked points, and
*         (b) the marked points all lie within the maximin edge
*             distance plus epsilon of the origin.
*
* Step 3. Draw lines from the marked point in the interior of each
*         face to the marked points on the incident edges. Note that
*         (a) the face identifications respect the lines, and
*         (b) the lines all lie within the maximin edge
*             distance plus epsilon of the origin.
*
* Step 4. Cone the complex created in steps (1), (2) and (3) to
*         the origin. This gives a spine which is dual to the
*         Dirichlet domain and lies within the maximin edge
*         distance plus epsilon of the origin.
*
* Q.E.D.
*
* Modifications to the above construction.
*
* The spine radius discussed above works correctly for all manifolds,
* but it can be large for a cusped manifold whose Dirichlet
* domain contains a vertex lying "out in the cusp". For example,
* the manifold m015 has a vertex at a distance 3.29 from the center
* of the Dirichlet domain. This yields a large value for the spine
* radius, which in turn makes the (exponential time) length spectrum
* algorithm run very slowly. Fortunately, such a large spine radius
* is unnecessary. Roughly speaking, vertices "out in the cusp" should
* be considered part of the cusp. We can make this idea rigorous
* as follows. [Note: After applying the following modifications,
* the spine radius for m015 went from 3.29 down to 0.84. Therefore
* the tiling radius for a length spectrum to  $L = 1.0$  went from over 6
* to about 2, i.e. from nearly impossible to almost instantaneous.]
*
* Definition. In the following discussion, "the space" means
* the manifold or orbifold obtained by gluing the faces of the
* Dirichlet domain.
*
* The spine divides the space into 3-dimensional regions dual
* to a vertices of the Dirichlet domain. In a manifold, a region
* dual to a finite vertex will be a 3-ball, but in an orbifold
* a region dual to a finite vertex could be a cone on any spherical
* 2-orbifold. Similarly, a region dual to an ideal vertex will be
* either a torus or Klein bottle cross a half line, but in an orbifold
* it may be any Euclidean 2-orbifold cross a half line.
*
* Proposition. If a 2-cell in the spine separates two distinct
* regions, at least one of which is topologically a 3-ball, then
* we may remove the 2-cell and still retain the essential property
* of the spine, namely that every closed geodesic must intersect it.
*
* Proof. Obvious. Q.E.D.
*
* Definition. A "free edge" of a 2-cell in a spine is an edge
* which is adjacent to no other 2-cells (nor to any other edges
* of the given 2-cell). Initially there are no free edges, but
* some may be created as 2-cells are eliminated as in the above

```

```

* proposition.
*
* Proposition. If a 2-cell in the spine is dual to a nonsingular
* edge (in the Dirichlet complex) and has a free edge (in the spine),
* then we may remove the 2-cell and still retain the essential property
* of the spine, namely that every closed geodesic must intersect it.
*
* Proof. The 2-cell is a disk. Isotope the free edge across
* the 2-cell to eliminate both. The topology of the incident region
* does not change, so neither does the fact that every geodesic
* must intersect the spine. Q.E.D.
*
* Proposition. If, after applying the above propositions, we find
* a 1-cell in the spine with no incident 2-cells, we may remove the
* 1-cell and still retain the essential property of the spine, namely
* that every closed geodesic must intersect it.
*
* Proof. We may locally isotope a geodesic to avoid naked 1-cells.
* If the geodesic had no other intersections with the spine, then
* it would lie entirely within a single region, and therefore
* couldn't be a geodesic. Q.E.D.
*
* Comment. We remove the naked 1-cells only after we've finished
* removing 2-cells.
*
* Proposition. After removing some 2-cells and 1-cells from a spine
* as in the preceding propositions, the radius of the remaining spine
* will be the "maximin" edge distance (defined above), taken over the
* edges which are dual to the remaining 2-cells (i.e. excluding edges
* dual to 2-cells which have been removed).
*
* Proof. This is almost an immediate consequence of the algorithm for
* constructing the spine. The only situation that could get us into
* trouble would be a naked 1-cell in the spine, but the previous
* proposition shows that we may remove them. Q.E.D.
*
* Our algorithm will be to look at an edge for which
* edge_class->dist_edge_to_origin is a maximum. If it does not
* connect two distinct regions, one of which is a 3-ball, then
* we look for free edges. If that fails, then edge_class->
* dist_edge_to_origin is the spine_radius and we're done.
* If the edge does connect a 3-ball to some other region, we remove
* the dual 2-cell and continue with the edge having the next greatest
* value of edge_class->dist_edge_to_origin, and so on until we reach
* a 2-cell which cannot be removed, at which point we're done.
*/

WEEdgeClass      *edge_class;
WEVertexClass    *vertex_class,
                 *vc[2],
                 *region[2];
double           max_value;
WEEdge           *edge,
                 *max_edge;
Boolean          union_is_3_ball;

/*
* Initialize all edge_class->removed flags to FALSE.
*/

for (    edge_class = polyhedron->edge_class_begin.next;
        edge_class != &polyhedron->edge_class_end;
        edge_class = edge_class->next)

    edge_class->removed = FALSE;

/*
* Initially the region dual to each vertex belongs
* to itself (i.e. none have been merged). A region
* is a 3-ball iff its solid_angle is 4pi.
*/

for (    vertex_class = polyhedron->vertex_class_begin.next;
        vertex_class != &polyhedron->vertex_class_end;

```

```

        vertex_class = vertex_class->next)
    {
        vertex_class->belongs_to_region = vertex_class;
        vertex_class->is_3_ball
            = (vertex_class->solid_angle > 4.0*PI - PI_EPSILON);
    }

/*
 * Look at each edge class in turn, starting with the one furthest
 * from the origin.  If its dual 2-cell may be removed, remove it.
 * Otherwise set the spine_radius and return.
 */

while (TRUE)
{
    /*
     * Find a representative of the furthest edge class
     * which has not already been removed.
     */

    max_value = 0.0;

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

        if (edge->e_class->removed == FALSE
            && edge->e_class->dist_edge_to_origin > max_value)
        {
            max_edge    = edge;
            max_value    = edge->e_class->dist_edge_to_origin;
        }

    if (max_value == 0.0)
        uFatalError("compute_spine_radius", "Dirichlet_extras");

    /*
     * Note the vertex classes at max_edge's endpoints.
     */
    vc[0] = max_edge->v[0]->v_class;
    vc[1] = max_edge->v[1]->v_class;

    /*
     * If the regions dual to max_edge's endpoints are distinct,
     * and at least one is a 3-ball, then max_edge may be removed.
     */
    if (vc[0]->belongs_to_region != vc[1]->belongs_to_region
        && (vc[0]->is_3_ball || vc[1]->is_3_ball))
    {
        /*
         * We found a removable edge!
         */

        /*
         * Remove the edge.
         */
        max_edge->e_class->removed = TRUE;

        /*
         * Annex vc[1]'s region to vc[0]'s.
         */

        region[0] = vc[0]->belongs_to_region;
        region[1] = vc[1]->belongs_to_region;

        for (    vertex_class = polyhedron->vertex_class_begin.next;
               vertex_class != &polyhedron->vertex_class_end;
               vertex_class = vertex_class->next)

            if (vertex_class->belongs_to_region == region[1])

                vertex_class->belongs_to_region = region[0];

        /*

```

```

    /* Is the union of the two regions a 3-ball?
    */

    union_is_3_ball = (vc[0]->is_3_ball && vc[1]->is_3_ball);

    for (    vertex_class = polyhedron->vertex_class_begin.next;
           vertex_class != &polyhedron->vertex_class_end;
           vertex_class = vertex_class->next)

        if (vertex_class->belongs_to_region == region[0])

            vertex_class->is_3_ball = union_is_3_ball;
    }
    else
    {
        /*
        * If we're lucky, some free edge removal might get rid
        * of the 2-cell dual to max_edge.
        */
        attempt_free_edge_removal(polyhedron);

        /*
        * Did free edge removal do the trick?
        */
        if (max_edge->e_class->removed == TRUE)
        {
            /*
            * Great. This edge is gone.
            * Continue with the loop to examine the next edge.
            */
        }
        else
        {
            /*
            * We found a nonremovable edge.
            */
            polyhedron->spine_radius = max_value;
            return;
        }
    }
}

/*
* The function returns from within the above loop.
*/
}

static void attempt_free_edge_removal(
    WEPolyhedron *polyhedron)
{
    WEFace *face;
    WEEdge *edge,
           *remaining_edge;
    int count;

    /*
    * Examine each of the polyhedron's faces.
    */
    for (face = polyhedron->face_list_begin.next;
         face != &polyhedron->face_list_end;
         face = face->next)
    {
        /*
        * Count how many of the incident edges have not yet
        * been removed. If a non-removed edge is found, remember it.
        */

        count = 0;
        remaining_edge = NULL;

        edge = face->some_edge;
        do
        {

```

```

    /*
     * Has this edge been removed?
     */
    if (edge->e_class->removed == FALSE)
    {
        count++;
        remaining_edge = edge;
    }

    /*
     * Advance counterclockwise to the next edge.
     */
    if (edge->f[left] == face)
        edge = edge->e[tip][left];
    else
        edge = edge->e[tail][right];
} while (edge != face->some_edge);

/*
 * If precisely one incident edge has a dual 2-cell which
 * has not been removed, then we have a free edge.
 */
if (count == 1)
{
    /*
     * We may isotope the free edge across the dual 2-cell
     * iff the edge is nonsingular.
     */
    if (remaining_edge->e_class->dihedral_angle > 2.0*PI - PI_EPSILON)
    {
        /*
         * Remove the edge.
         * (The incident 3-cell's belongs_to_region and
         * is_3_ball fields are not affected.)
         */
        remaining_edge->e_class->removed = TRUE;

        /*
         * Set face = &polyhedron->face_list_begin to restart
         * the loop, just in case removing this edge allows
         * other edges to be removed.
         */
        face = &polyhedron->face_list_begin;
    }
}
}

static void compute_deviation(
    WEPolyhedron *polyhedron)
{
    /*
     * Each face->group_element is, in theory, an element of SO(3,1).
     * Record the greatest deviation from O(3,1) in polyhedron->deviation,
     * so the UI has some idea how precise the calculations are.
     */

    WEFace *face;
    double the_deviation;

    polyhedron->deviation = 0.0;

    for (face = polyhedron->face_list_begin.next;
         face != &polyhedron->face_list_end;
         face = face->next)
    {
        the_deviation = o3l_deviation(*face->group_element);
        if (the_deviation > polyhedron->deviation)
            polyhedron->deviation = the_deviation;
    }
}

```

```

static void compute_geometric_Euler_characteristic(
    WEPolyhedron *polyhedron)
{
    /*
     * As explained in winged_edge.h the geometric Euler characteristic is
     *
     * 
$$c[0] - c[1] + c[2] - c[3]$$

     *
     * where
     *
     *  $c[0]$  = the sum of the solid angles at the vertices divided by  $4\pi$ ,
     *
     *  $c[1]$  = the sum of the dihedral angles at the edges divided by  $2\pi$ ,
     *
     *  $c[2]$  = half the number of faces of the Dirichlet domain,
     *
     *  $c[3]$  = the number of 3-cells, which is always one.
     */

    double c[4];
    WEVertexClass *vertex_class;
    WEEdgeClass *edge_class;

    /*
     * Compute c[0].
     */

    c[0] = 0.0;

    for ( vertex_class = polyhedron->vertex_class_begin.next;
          vertex_class != &polyhedron->vertex_class_end;
          vertex_class = vertex_class->next)

        c[0] += vertex_class->solid_angle;

    c[0] /= 4.0 * PI;

    /*
     * Compute c[1].
     */

    c[1] = 0.0;

    for ( edge_class = polyhedron->edge_class_begin.next;
          edge_class != &polyhedron->edge_class_end;
          edge_class = edge_class->next)

        c[1] += edge_class->dihedral_angle;

    c[1] /= 2.0 * PI;

    /*
     * Compute c[2].
     */

    c[2] = (double)polyhedron->num_faces / 2.0;

    /*
     * "Compute" c[3].
     */

    c[3] = 1.0;

    /*
     * Compute the geometric Euler characteristic of the quotient
     * manifold obtained by identifying faces of the Dirichlet domain.
     */

    polyhedron->geometric_Euler_characteristic = c[0] - c[1] + c[2] - c[3];
}

```