

```

/*
 * Dirichlet_construction.c
 *
 * The Dirichlet domain code is divided among several files. The header
 * file Dirichlet.h explains the organization of the three files, and
 * provides the common definitions and declarations.
 *
 * This file provides the function
 *
 *      WEPolyhedron *compute_Dirichlet_domain( MatrixPairList  *gen_list,
 *                                              double             vertex_epsilon);
 *
 * compute_Dirichlet_domain() computes the Dirichlet domain defined by
 * the list of generators, and sets the list of generators equal to
 * the face pairings of the Dirichlet domain, sorted by order of increasing
 * image height (see Dirichlet_basepoint.c for the definition of "image
 * height"). It does not set certain cosmetic fields (vertex->ideal, etc.)
 * which aren't needed for the computation; it assumes the calling
 * function will set them at the very end (cf. bells_and_whistles() in
 * Dirichlet_extras.c). If compute_Dirichlet_domain() fails (as explained
 * immediately below) it returns NULL.
 *
 * Error detection. No Dirichlet domain algorithm can be perfect.
 * If you give it, say, the generators for a (p,q) Dehn surgery on the
 * figure eight knot, then it will surely fail for large enough p and q,
 * because the true Dirichlet domain will have a vast number of tiny
 * faces, and the numerical precision of the underlying hardware won't
 * be good enough to resolve them all correctly. The present algorithm
 * will attempt to resolve the faces as best it can, but will return
 * NULL if it fails. Its strategy is to start with a cube and intersect
 * it with appropriate half planes, one at a time. At each step it checks
 * how the slicing plane intersects the existing polyhedron. Vertices
 * lying very close to the slicing plane are assumed to lie on it (the
 * precise interpretation of "very close" is defined by vertex_epsilon).
 * Each edge passing from one side of the slicing plane to the other is
 * divided in two by introducing a new vertex at the point where the edge
 * intersects the slicing plane. Similarly, each face which passes through
 * the slicing plane is divided in two. The program then checks rigorously
 * that the slicing plane divides the surface of the polyhedron into two
 * combinatorial disks. If it does, the vertices, edges and faces on
 * the "far" side of the slicing plane are discarded, a new face is
 * introduced, and we know for sure that the surface of the polyhedron
 * is still a combinatorial ball. If the slicing plane does not divide
 * the surface of the polyhedron into two combinatorial disk (as could
 * happen when the vertices are so close together that roundoff error
 * introduces inconsistent combinatorial relations), the program gives up
 * and returns NULL.
 *
 * The basic approach of the algorithm is to first use the given generators
 * to find a polyhedron which
 *
 * (1) is a superset of the true Dirichlet domain, and
 *
 * (2) does not extend beyond the sphere at infinity.
 *
 * Once it has such a polyhedron, it checks whether the faces match correctly.
 * If they all do, it's done. Otherwise it uses the pairs of nonmatching
 * faces to find new group elements which are guaranteed to slice off at
 * least one vertex of the candidate polyhedron. In this way it is guaranteed
 * to arrive at the true Dirichlet domain in a finite (and typically small)
 * number of steps. The algorithm is described in more detail in the code
 * itself.
 *
 * For closed manifolds the algorithm works quickly and reliably. For
 * large cusped manifolds, computing a candidate polyhedron satisfying
 * (1) and (2) above is a bottleneck. I hope to eventually find a quicker
 * way to compute it.
 *
 * The Dirichlet domain algorithm uses matrices in  $O(3,1)$ . Despite their
 * many theoretical advantages, they lead to rapidly accumulating numerical
 * errors when the manifold is at all large (see Dirichlet_precision.c for
 * a more detailed discussion of the problem). Occasionally the algorithm will
 * encounter topological inconsistencies and fail, and will return NULL.
 * Fortunately the problem can usually be corrected. The root of the problem

```

```

* is the need to decide whether two closely spaced vertices should be
* interpreted as distinct vertices of the Dirichlet domain, or as two images
* of a single vertex, differing only by roundoff error. The decision is
* guided by the value of the vertex_epsilon parameter. Use a small value
* (e.g. 1e-12) to compute Dirichlet domains with many small faces, e.g. a
* (100,1) Dehn surgery on the figure eight knot. Use a larger value when
* computing simple Dirichlet domains for larger manifolds. (Larger manifolds
* tend to have less accurately defined face pairings, due to the numerical
* problems in O(3,1).) The most extreme vertex resolution values (say less
* than 1e-16 or more than 1e-2) usually fail, so try to stay within that
* range unless you're really desperate.
*
* Technical note: the WEPolyhedron's num_vertices, num_edges and num_faces
* fields are not maintained during the construction, but are set at the end.
*
* By the way, compute_Dirichlet_domain() assumes no nontrivial group element
* fixes the origin. In other words, it assumes that the basepoint does not
* lie in an orbifold's singular set. Dirichlet_from_generators_with_displacement()
* in Dirichlet.c makes sure that doesn't happen.
*/

#include "kernel.h"
#include "Dirichlet.h"
#include <stdlib.h> /* needed for qsort() */

/*
* The Dirichlet domain computation begins with a large cube enclosing the
* projective model of hyperbolic space, and intersects it with the half
* spaces corresponding to the initial set of generators. CUBE_SIZE
* is half the length of the cube's side. CUBE_SIZE must be at least 1.0
* so that the Dirichlet domain fits inside, but it shouldn't be so large
* that numerical accuracy suffers.
*/
#define CUBE_SIZE 2.0

/*
* If the distance between the basepoint (1, 0, 0, 0) and one
* of its translates is less than about ERROR_EPSILON,
* compute_normal_to_Dirichlet_plane() returns func_failed. If it
* weren't for roundoff error this should never happen, since we
* take care to move t moves towards a maximum of the injectivity
* radius it should go still further from the singular set. But
* roundoff error may produce elements which should be the identity,
* but aren't close enough to have been recognized as such.
* Note that ERROR_EPSILON needs to be coordinated with MATRIX_EPSILON
* in Dirichlet.h, but there not much slack between them.
*/
#define ERROR_EPSILON 1e-4

/*
* A vertex is considered hyperideal iff o3l_inner_product(vertex->x, vertex->x)
* is greater than HYPERIDEAL_EPSILON. Recall that vertex->x[0] is always 1,
* so that if the vertex is at a Euclidean distance (1 + epsilon) from the origin
* in the Klein model, o3l_inner_product(vertex->x, vertex->x) will be
* -1 + (1 + epsilon)^2 = 2*epsilon.
*
* 96/9/5 Changed HYPERIDEAL_EPSILON from 1e-4 to 1e-3 to avoid
* infinite loop when computing Dirichlet domain for x004 with
* "coarse" vertex resolution.
*/
#define HYPERIDEAL_EPSILON 1e-3

/*
* verify_group() considers one O3lMatrix to be simpler than
* another if its height is at least VERIFY_EPSILON less.
*/
#define VERIFY_EPSILON 1e-4

/*
* intersect_with_halfspaces() will report a failure if the MatrixPair
* it is given deviates from O(3,1) by more than DEVIATION_EPSILON.
*/
#define DEVIATION_EPSILON 1e-3

```

```

static WEPolyhedron *initial_polyhedron(MatrixPairList *gen_list, double vertex_epsilon);
static WEPolyhedron *new_WEPolyhedron(void);
static void make_cube(WEPolyhedron *polyhedron);
static FuncResult slice_polyhedron(WEPolyhedron *polyhedron, MatrixPairList *gen_list);
static FuncResult intersect_with_halfspaces(WEPolyhedron *polyhedron, MatrixPair *
matrix_pair);
static Boolean same_image_of_origin(O3lMatrix m0, O3lMatrix m1);
static FuncResult slice_with_hyperplane(WEPolyhedron *polyhedron, O3lMatrix m, WEFace **
new_face);
static FuncResult compute_normal_to_Dirichlet_plane(O3lMatrix m, O3lVector normal_vector)
;
static void compute_vertex_to_hyperplane_distances(WEPolyhedron *polyhedron,
O3lVector normal_vector);
static Boolean positive_vertices_exist(WEPolyhedron *polyhedron);
static void cut_edges(WEPolyhedron *polyhedron);
static FuncResult cut_faces(WEPolyhedron *polyhedron);
static FuncResult check_topology_of_cut(WEPolyhedron *polyhedron);
static void install_new_face(WEPolyhedron *polyhedron, WEFace *new_face);
static Boolean face_still_exists(WEPolyhedron *polyhedron, WEFace *face);
static Boolean has_hyperideal_vertices(WEPolyhedron *polyhedron);
static void compute_all_products(WEPolyhedron *polyhedron, MatrixPairList *
product_list);
static void poly_to_current_list(WEPolyhedron *polyhedron, MatrixPairList *
current_list);
static void current_list_to_product_tree(MatrixPairList *current_list, MatrixPair **product_tree);
static Boolean already_on_product_tree(O3lMatrix product, MatrixPair *product_tree);
static void add_to_product_tree(O3lMatrix product, MatrixPair **product_tree);
static void product_tree_to_product_list(MatrixPair *product_tree, MatrixPairList *
product_list);
static void append_tree_to_list(MatrixPair *product_tree, MatrixPair *list_end);
static FuncResult check_faces(WEPolyhedron *polyhedron);
static FuncResult pare_face(WEFace *face, WEPolyhedron *polyhedron, Boolean *
face_was_pared);
static FuncResult pare_mated_face(WEFace *face, WEPolyhedron *polyhedron, Boolean *
face_was_pared);
static FuncResult pare_mateless_face(WEFace *face, WEPolyhedron *polyhedron, Boolean *
face_was_pared);
static FuncResult try_this_alpha(O3lMatrix *alpha, WEFace *face, WEPolyhedron *polyhedron,
Boolean *face_was_pared);
static void count_cells(WEPolyhedron *polyhedron);
static void sort_faces(WEPolyhedron *polyhedron);
static int CDECL compare_face_distance(const void *ptr1, const void *ptr2);
static Boolean verify_faces(WEPolyhedron *polyhedron);
static FuncResult verify_group(WEPolyhedron *polyhedron, MatrixPairList *gen_list);
static void rewrite_gen_list(WEPolyhedron *polyhedron, MatrixPairList *gen_list);

WEPolyhedron *compute_Dirichlet_domain(
MatrixPairList *gen_list,
double vertex_epsilon)
{
WEPolyhedron *polyhedron;

/*
* Initialize the polyhedron to be the intersection of the
* half spaces defined by the elements of the gen_list.
*/
polyhedron = initial_polyhedron(gen_list, vertex_epsilon);

/*
* If topological problems caused by roundoff errors get
* in the way, report a failure.
*/
if (polyhedron == NULL)
return NULL;

/*
* Check whether pairs of faces match. If a pair fails to
* match exactly, use the corresponding group elements to
* create a new face plane which slices off another bit of
* the WEPolyhedron. Perform a similar operation if some
* face lacks a mate. If roundoff errors get in the way,

```

```

    * free the polyhedron and return NULL.
    */
    if (check_faces(polyhedron) == func_failed)
    {
        free_Dirichlet_domain(polyhedron);
        return NULL;
    }

    /*
    * Count the number of vertices, edges and faces.
    */
    count_cells(polyhedron);

    /*
    * Put the faces in order of increasing distance from the basepoint.
    */
    sort_faces(polyhedron);

    /*
    * Count the number of edges incident to each face. If a face and
    * its mate don't have the same number of incident edges, free the
    * polyhedron and return NULL.
    */
    if (verify_faces(polyhedron) == func_failed)
    {
        free_Dirichlet_domain(polyhedron);
        return NULL;
    }

    /*
    * Verify that the face pairings generate the group as
    * originally specified by gen_list. This ensures that
    * we have a Dirichlet domain for the manifold itself,
    * not some finite-sheeted cover.
    */
    if (verify_group(polyhedron, gen_list) == func_failed)
    {
        free_Dirichlet_domain(polyhedron);
        return NULL;
    }

    /*
    * Discard the old list to generators and replace it with the
    * set of face pairing transformations (plus the identity).
    * They will be in ascending order, because we've already sorted
    * the face list.
    */
    rewrite_gen_list(polyhedron, gen_list);

    return polyhedron;
}

static WEPolyhedron *initial_polyhedron(
    MatrixPairList *gen_list,
    double vertex_epsilon)
{
    WEPolyhedron *polyhedron;
    MatrixPairList product_list;

    /*
    * Allocate a WEPolyhedron data structure, and initialize
    * its doubly-linked lists of vertices, edges and faces.
    */
    polyhedron = new_WEPolyhedron();

    /*
    * Set the vertex_epsilon;
    */
    polyhedron->vertex_epsilon = vertex_epsilon;

    /*
    * Initialize the polyhedron to be a big cube.
    */

```

```

make_cube(polyhedron);

/*
 * Intersect the cube with the halfspaces defined by the elements
 * of gen_list.  If topological problems due to roundoff error
 * get in the way, free the polyhedron and return NULL.
 */
if (slice_polyhedron(polyhedron, gen_list) == func_failed)
{
    free_Dirichlet_domain(polyhedron);
    return NULL;
}

/*
 * While hyperideal vertices remain (i.e. vertices lying beyond the
 * sphere at infinity), compute all products of face->group_elements,
 * and intersect the polyhedron with the corresponding half spaces.
 */
while (has_hyperideal_vertices(polyhedron) == TRUE)
{
    compute_all_products(polyhedron, &product_list);
    if (slice_polyhedron(polyhedron, &product_list) == func_failed)
    {
        free_matrix_pairs(&product_list);
        free_Dirichlet_domain(polyhedron);
        return NULL;
    }
    free_matrix_pairs(&product_list);
}

return polyhedron;
}

static WEPolyhedron *new_WEPolyhedron()
{
    WEPolyhedron    *new_polyhedron;

    new_polyhedron = NEW_STRUCT(WEPolyhedron);

    new_polyhedron->num_vertices    = 0;
    new_polyhedron->num_edges       = 0;
    new_polyhedron->num_faces       = 0;

    new_polyhedron->vertex_list_begin.prev = NULL;
    new_polyhedron->vertex_list_begin.next = &new_polyhedron->vertex_list_end;
    new_polyhedron->vertex_list_end .prev = &new_polyhedron->vertex_list_begin;
    new_polyhedron->vertex_list_end .next = NULL;

    new_polyhedron->edge_list_begin.prev = NULL;
    new_polyhedron->edge_list_begin.next = &new_polyhedron->edge_list_end;
    new_polyhedron->edge_list_end .prev = &new_polyhedron->edge_list_begin;
    new_polyhedron->edge_list_end .next = NULL;

    new_polyhedron->face_list_begin.prev = NULL;
    new_polyhedron->face_list_begin.next = &new_polyhedron->face_list_end;
    new_polyhedron->face_list_end .prev = &new_polyhedron->face_list_begin;
    new_polyhedron->face_list_end .next = NULL;

    new_polyhedron->vertex_class_begin.prev = NULL;
    new_polyhedron->vertex_class_begin.next = &new_polyhedron->vertex_class_end;
    new_polyhedron->vertex_class_end .prev = &new_polyhedron->vertex_class_begin;
    new_polyhedron->vertex_class_end .next = NULL;

    new_polyhedron->edge_class_begin.prev = NULL;
    new_polyhedron->edge_class_begin.next = &new_polyhedron->edge_class_end;
    new_polyhedron->edge_class_end .prev = &new_polyhedron->edge_class_begin;
    new_polyhedron->edge_class_end .next = NULL;

    new_polyhedron->face_class_begin.prev = NULL;
    new_polyhedron->face_class_begin.next = &new_polyhedron->face_class_end;
    new_polyhedron->face_class_end .prev = &new_polyhedron->face_class_begin;
    new_polyhedron->face_class_end .next = NULL;
}

```

```

    return new_polyhedron;
}

static void make_cube(
    WEPolyhedron *polyhedron)
{
    /*
     * This function accepts a pointer to an empty WEPolyhedron (i.e. the
     * WEPolyhedron data structure is allocated and the doubly linked lists
     * are initialized, but the WEPolyhedron has no vertices, edges or
     * faces), and sets it to be a cube of side twice CUBE_SIZE. Each face
     * of the cube has its group_element field set to NULL to indicate that
     * it doesn't correspond to any element of the group. The vertices,
     * faces and edges of the cube are indexed as shown in the following
     * diagram of a cut open cube.
     */
    /*
     *
     *      1-----5->-----3
     *      |               |
     *      2               3
     *      |               |
     *      V               V
     *      |               |
     *      1-----2->-----5-----7->-----7-----<-3-----3
     *      |               |               |               |
     *      ^               ^               ^               ^
     *      |               |               |               |
     *      8               9               11              10
     *      |               |               |               |
     *      0-----0->-----4-----6->-----6-----<-1-----2
     *      |               |               |               |
     *      ^               ^               ^               ^
     *      |               |               |               |
     *      0               4               1               2
     *      |               |               |               |
     *      0-----4->-----2
     *      |               |               |               |
     *      8               10              0
     *      |               |               |               |
     *      V               V               V               V
     *      |               |               |               |
     *      1-----5->-----3
     */
    int i,
        j,
        k;
    WEVertex *initial_vertices[8];
    WEEdge *initial_edges[12];
    WEFace *initial_faces[6];

    const static int evdata[12][2] =
    {
        {0, 4},
        {2, 6},
        {1, 5},
        {3, 7},
        {0, 2},
        {1, 3},
        {4, 6},
        {5, 7},
        {0, 1},
        {4, 5},
        {2, 3},
        {6, 7}
    };

```

```

const static int eedata[12][2][2] =
{
    {{ 8, 4}, { 9, 6}},
    {{ 4, 10}, { 6, 11}},
    {{ 5, 8}, { 7, 9}},
    {{10, 5}, {11, 7}},
    {{ 0, 8}, { 1, 10}},
    {{ 8, 2}, {10, 3}},
    {{ 9, 0}, {11, 1}},
    {{ 2, 9}, { 3, 11}},
    {{ 4, 0}, { 5, 2}},
    {{ 0, 6}, { 2, 7}},
    {{ 1, 4}, { 3, 5}},
    {{ 6, 1}, { 7, 3}}
};
const static int efdata[12][2] =
{
    {2, 4},
    {4, 3},
    {5, 2},
    {3, 5},
    {4, 0},
    {0, 5},
    {1, 4},
    {5, 1},
    {0, 2},
    {2, 1},
    {3, 0},
    {1, 3}
};
const static int fdata[6] = {4, 6, 0, 1, 0, 2};

/*
 * If we were keeping track of the number of vertices, edges and faces,
 * we'd want to set the counts here. But we're not, so we won't.
 */
polyhedron->num_vertices = 8;
polyhedron->num_edges = 12;
polyhedron->num_faces = 6;

for (i = 0; i < 8; i++)
{
    initial_vertices[i] = NEW_STRUCT(WVertex);
    INSERT_BEFORE(initial_vertices[i], &polyhedron->vertex_list_end);
}

for (i = 0; i < 12; i++)
{
    initial_edges[i] = NEW_STRUCT(WEdge);
    INSERT_BEFORE(initial_edges[i], &polyhedron->edge_list_end);
}

for (i = 0; i < 6; i++)
{
    initial_faces[i] = NEW_STRUCT(WFace);
    INSERT_BEFORE(initial_faces[i], &polyhedron->face_list_end);
}

for (i = 0; i < 8; i++)
{
    initial_vertices[i]->x[0] = 1.0;
    initial_vertices[i]->x[1] = (i & 4) ? CUBE_SIZE : -CUBE_SIZE;
    initial_vertices[i]->x[2] = (i & 2) ? CUBE_SIZE : -CUBE_SIZE;
    initial_vertices[i]->x[3] = (i & 1) ? CUBE_SIZE : -CUBE_SIZE;
}

for (i = 0; i < 12; i++)
{
    for (j = 0; j < 2; j++) /* j = tail, tip */
        initial_edges[i]->v[j] = initial_vertices[evdata[i][j]];

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

```

```

        for (k = 0; k < 2; k++)          /* k = left, right */
            initial_edges[i]->e[j][k] = initial_edges[eedata[i][j][k]];

    for (j = 0; j < 2; j++)              /* j = left, right */
        initial_edges[i]->f[j] = initial_faces[efdata[i][j]];
}

for (i = 0; i < 6; i++)
{
    initial_faces[i]->some_edge      = initial_edges[fdata[i]];
    initial_faces[i]->mate          = NULL;
    initial_faces[i]->group_element = NULL;
}

static FuncResult slice_polyhedron(
    WEPolyhedron *polyhedron,
    MatrixPairList *gen_list)
{
    MatrixPair *matrix_pair;

    /*
     * Intersect the polyhedron with the pair of halfspaces corresponding
     * to each MatrixPair on gen_list, except for the identity.
     *
     * Technical note: intersect_with_halfspaces() may set some faces'
     * face->clean fields to FALSE. We aren't using the face->clean field
     * at this point, so these operations are unnecessary but harmless.
     * check_faces() uses the face->clean fields, and calls low-level
     * functions which call intersect_with_halfspaces().
     */

    for (matrix_pair = gen_list->begin.next;
         matrix_pair != &gen_list->end;
         matrix_pair = matrix_pair->next)

        if (o3l_equal(matrix_pair->m[0], O3l_identity, MATRIX_EPSILON) == FALSE)

            if (intersect_with_halfspaces(polyhedron, matrix_pair) == func_failed)
                return func_failed;

    return func_OK;
}

static FuncResult intersect_with_halfspaces(
    WEPolyhedron *polyhedron,
    MatrixPair *matrix_pair)
{
    /*
     * Intersect the polyhedron with the halfspaces corresponding
     * to each of the O3lMatrices in the matrix_pair. This will create
     * 0, 1 or 2 new faces, depending on whether the hyperplanes actually
     * slice off a nontrivial part of the polyhedron.
     */

    int i;
    WEFace *new_face[2];

    /*
     * If the given MatrixPair deviates too far from O(3,1), report a failure.
     * (It suffices to check matrix_pair->m[0], because matrix_pair->m[1] is
     * computed as the transpose of matrix_pair->m[0] with appropriate
     * elements negated.)
     */
    if (o3l_deviation(matrix_pair->m[0]) > DEVIATION_EPSILON)
        return func_failed;

    /*
     * To allow for orbifolds as well as manifolds, we must be prepared
     * for the possibility that matrix_pair->m[0] and matrix_pair->m[1]
     * define the same hyperplane, which will be its own mate. Let
     * f0 and f1 be the isometries of hyperbolic 3-space defined by

```



```

* matrix_pair->m[0] and matrix_pair->m[1], respectively.
*
* Proposition. The following are equivalent.
*
* (1) The hyperplanes defined by f0 and f1 are equal.
*
* (2)  $f_0(\text{origin}) = f_1(\text{origin})$ 
*
* (3)  $f_0(f_0(\text{origin})) = \text{origin}$ 
*
* (4) f0 is a half turn about an axis which passes orthogonally
*      through the midpoint of the segment connecting the
*      origin to  $f_0(\text{origin})$ .
*      or
*      f0 is a reflection in the plane which passes orthogonally
*      through the midpoint of the segment connecting the
*      origin to  $f_0(\text{origin})$ .
*      or
*      f0 is a reflection through the midpoint of the segment
*      connecting the origin to  $f_0(\text{origin})$ .
*
* (5)  $f_0 = f_1$ 
*
* Proof. Throughout this proof, keep in mind that we first gave the
* basepoint a small random displacement, and then perhaps moved it
* towards a point of maximum injectivity radius, so we may assume
* that its images under the covering transformation group are distinct.
*
* (1 => 2). Obvious.
*
* (2 => 3).  $f_0(f_0(\text{origin})) = f_0(f_1(\text{origin})) = \text{origin}$ , because
*  $f_0$  and  $f_1$  are inverses of one another.
*
* (3 => 4).  $f_0$  interchanges the origin and  $f_0(\text{origin})$ . Therefore
* it fixes the midpoint M of the segment S connecting the origin
* to  $f_0(\text{origin})$ . Furthermore, the plane P which passes through
* M and is orthogonal to S will be taken to itself (setwise, but
* probably not pointwise). Consider the possibilities for the
* action of  $f_0$  on P.
*
* Case 1.  $f_0$  preserves the orientation of hyperbolic 3-space, and
* therefore reverses the orientation of P. Because  $f_0$  fixes the
* point M, it must act on P as a reflection in some line L which
* contains M and is contained in P. It follows that  $f_0$  acts on
* hyperbolic 3-space as a half turn about the line L.
*
* Case 2.  $f_0$  reverses the orientation of hyperbolic 3-space, and
* therefore preserves the orientation of P. Because  $f_0$  fixes the
* point M, it must act on P as a rotation about M through some angle
* theta. This implies that  $f_0^2$  acts on hyperbolic 3-space as a
* rotation about the line containing S through an angle  $2\theta$ .
* But the nondegenerate choice of basepoint (cf. the comments at the
* beginning of this proof) implies that  $2\theta = 0 \pmod{2\pi}$ .
* So  $\theta = 0$  or  $\pi \pmod{2\pi}$ . If  $\theta = 0$ ,  $f_0$  acts on hyperbolic
* 3-space as a reflection in the plane P. If  $\theta = \pi$ ,  $f_0$  acts on
* hyperbolic 3-space as a reflection in the point M.
*
* (4 => 5). In each of the three cases lists (half turn about axis,
* reflection in plane, reflection in point)  $f_0$  is its own inverse.
* Therefore  $f_0 = f_1$ .
*
* (5 => 1). Trivial.
*
* Q.E.D.
*/

if (same_image_of_origin(matrix_pair->m[0], matrix_pair->m[1]) == TRUE)
{
    /*
    * The images of the origin are the same, so the above proposition
    * implies that the matrices must be the same as well. As a
    * guard against errors, let's check that this really is the case.
    */
    if (o31_equal(matrix_pair->m[0], matrix_pair->m[1], MATRIX_EPSILON) == FALSE)

```

```

        uFatalError("intersect_with_halfspaces", "Dirichlet_construction");

    /*
     * For documentation on these operations,
     * please see the generic case below.
     */

    if (slice_with_hyperplane(polyhedron, matrix_pair->m[0], &new_face[0]) ==
func_failed)
        return func_failed;

    if (new_face[0] != NULL)
        new_face[0]->mate = new_face[0];

    return func_OK;
}

/*
 * Slice the polyhedron with each of the two hyperplanes, and
 * record the newly created face, if any. If no new face is created,
 * slice_with_hyperplane() sets new_face[i] to NULL. If error
 * conditions (e.g. due to roundoff error) are encountered, return
 * func_failed.
 */
for (i = 0; i < 2; i++)
    if (slice_with_hyperplane(polyhedron, matrix_pair->m[i], &new_face[i]) ==
func_failed)
        return func_failed;

/*
 * The first hyperplane might have created a new face, only to have it
 * completely removed by the second hyperplane. (Technical note:
 * this check relies on the fact that slice_with_hyperplane() frees the
 * faces it removes AFTER allocating the new face.)
 */
if (new_face[0] != NULL && face_still_exists(polyhedron, new_face[0]) == FALSE)
    new_face[0] = NULL;

/*
 * Tell the new_faces about each other.
 */
for (i = 0; i < 2; i++)
    if (new_face[i] != NULL)
        new_face[i]->mate = new_face[!i];

return func_OK;
}

static Boolean same_image_of_origin(
    O31Matrix    m0,
    O31Matrix    m1)
{
    int i;

    for (i = 0; i < 4; i++)
        if (fabs(m0[i][0] - m1[i][0]) > MATRIX_EPSILON)
            return FALSE;

    return TRUE;
}

static FuncResult slice_with_hyperplane(
    WEPolyhedron *polyhedron,
    O31Matrix    m,
    WEFace       **new_face)
{
    /*
     * Remove the parts of the polyhedron cut off by the hyperplane P
     * determined by the matrix m. Set all fields except the mate field
     * of the newly created face, which is temporarily set to NULL here,
     * and set correctly by the calling routine.
     */

```

```

    */

    O31Vector        normal_vector;

    /*
    *   Initialize *new_face to NULL, just in case we detect an error
    *   condition and return before creating *new_face.
    */

    *new_face = NULL;

    /*
    *   Compute a vector normal to the hyperplane P determined by
    *   the matrix m.  The normal vector is, of course, defined relative
    *   to the Minkowski space metric, not the Euclidean metric.
    */
    if (compute_normal_to_Dirichlet_plane(m, normal_vector) == func_failed)
        return func_failed;

    /*
    *   Let P' be the restriction of the hyperplane P to the projective
    *   model, that is, to the 3-space  $x[0] == 1$ .  Measure the distance
    *   from P' to each vertex of the polyhedron, and store the result
    *   in the vertex's distance_to_plane field.  (Actually we compute
    *   numbers which are proportional to the vertices' distances to P',
    *   but that's good enough.)  In addition, set each vertex's
    *   which_side_of_plane field to +1, 0 or -1 according to whether,
    *   after accounting for possible roundoff error, the distance_to_plane
    *   is positive, zero or negative, respectively.
    */
    compute_vertex_to_hyperplane_distances(polyhedron, normal_vector);

    /*
    *   If no vertices have which_side_of_plane == +1, then there is
    *   nothing to be cut off, so return func_OK immediately.
    *   If no vertices have which_side_of_plane == -1, then we're
    *   cutting off everything, which merits a call to uFatalError().
    */
    if (positive_vertices_exist(polyhedron) == FALSE)
        return func_OK;

    /*
    *   Introduce a new vertex wherever an edge crosses the plane P'.
    *   Such edges can be recognized by the fact that one endpoint
    *   has which_side_of_plane == +1 while the other has
    *   which_side_of_plane == -1.
    */
    cut_edges(polyhedron);

    /*
    *   Introduce a new edge wherever a face crosses the plane P'.
    *   If any face has more than two 0-vertices (as might occur
    *   due to roundoff error) return func_failed.
    */
    if (cut_faces(polyhedron) == func_failed)
        return func_failed;

    /*
    *   Check that the curve we're cutting along is a single,
    *   simple closed curve.
    *
    *   Proposition.  The cut locus divides the surface of the ball into
    *   two combinatorial disks.  All the interior vertices on one side
    *   are positive (which_side_of_plane == +1) and all those on the
    *   other side are negative (which_side_of_plane == -1).
    *
    *   Proof.  check_topology_of_cut() insures that the cut locus is a
    *   simple closed curve.  The Schoenflies theorem says that a simple
    *   closed curve divides the surface of a 2-sphere into two disks.
    *   The function positive_vertices_exist() (cf. above) has already
    *   checked that both positive and negative vertices exist.  The
    *   function cut_edges() (cf. above) guarantees that no edge connects
    *   a positive vertex to a negative one.  Therefore it follows that
    *   one of the Schoenflies disks contains only positive vertices in its

```

```

    * interior, and the other contains only negative vertices.  Q.E.D.
    *
    * Corollary.  If we remove
    *
    * (1) all positive vertices,
    * (2) all edges incident to at least one positive vertex, and
    * (3) all faces incident to at least one positive vertex,
    *
    * and replace them with a single new face, the new domain will still
    * be a combinatorial ball.
    *
    * Proof.  This is almost obvious as a corollary of the above
    * proposition.  We just need to verify that conditions (1)-(3)
    * characterize the vertices, edges and face in the interior of
    * the positive Schoenflies disk.
    *
    * (1) We've already seen that all interior vertices must be positive.
    *
    * (2) An interior edge can't have two vertices on the boundary of the
    * Schoenflies disk, because if it did it would be a 0-edge and
    * would be part of the cut locus.  Therefore at least one vertex
    * is positive.
    *
    * (3) If all the vertices of a given face were 0-vertices, its
    * boundary would have to be the entire cut locus, and no vertices
    * could exist in its interior.  This would contradict the
    * existence of both positive and negative vertices.  Therefore
    * at least one vertex of an interior face must be positive.
    *
    * Q.E.D.
    */
if (check_topology_of_cut(polyhedron) == func_failed)
    return func_failed;

/*
 * Allocate the new_face.
 */
*new_face = NEW_STRUCT(WFace);

/*
 * Set its mate field to NULL and its clean field to FALSE.
 */
(*new_face)->mate = NULL;
(*new_face)->clean = FALSE;

/*
 * Allocate space for its group_element, and copy in the matrix m.
 */
(*new_face)->group_element = NEW_STRUCT(O31Matrix);
o31_copy((*new_face)->group_element, m);

/*
 * Throw away the vertices, edges and faces which are no longer
 * needed, and install the new face.
 */
install_new_face(polyhedron, *new_face);

return func_OK;
}

static FuncResult compute_normal_to_Dirichlet_plane(
    O31Matrix    m,
    O31Vector    normal_vector)
{
    /*
     * Compute a vector normal to the hyperplane P determined by
     * the matrix m.  The normal vector is, of course, defined relative
     * to the Minkowski space metric, not the Euclidean metric.
     *
     * The group element represented by the matrix m takes the basepoint
     *  $b = (1, 0, 0, 0)$  to one of its translates  $m(b)$ .  The hyperplane P
     * determined by m has normal vector  $m(b) - b$ .  Note that  $m(b)$  is
     * just the first column of the matrix m.
     */
}

```

```

*
* For numerical reasons, we want to scale the normal vector so that
* its largest entries have absolute value one. This gives
* compute_vertex_to_hyperplane_distances() a consistent idea of
* what sort of accuracy to expect when it computes the inner product
* of the normal vector with the vertices of the polyhedron.
* The absolute values of the entries in the (unscaled) normal vector
* grow exponentially with the translation distance, so without the
* scaling the magnitude of the anticipated error would also grow
* exponentially. With the scaling, the anticipated error will be
* on the order of DBL_EPSILON. (This is just the error in the inner
* product computation -- other errors are of course introduced
* elsewhere!)
*/

int      i;
double   max_abs;

/*
* Read m(b) from the first column of the matrix m.
*/
for (i = 0; i < 4; i++)
    normal_vector[i] = m[i][0];

/*
* Subtract off b = (1, 0, 0, 0) to get the normal_vector.
*/
normal_vector[0] -= 1.0;

/*
* Scale the normal_vector so that the largest absolute value of its
* entries is one.
*
* Technical digression. Except for very small translation distances
* it would be good enough to simply divide through by
* fabs(normal_vector[0]). This is because normal_vector[] is
* numerically similar to a lightlike vector, with a large [0]
* component whose square almost cancels the squares of the [1], [2]
* and [3] components. (Proof: m(b) is timelike while m(b) - b =
* m(b) - 1.0 is spacelike.) Here, though, we'll take a more
* pedestrian approach.
*/

/*
* Compute the largest absolute value of an entry.
*/
max_abs = 0.0;
for (i = 0; i < 4; i++)
    if (fabs(normal_vector[i]) > max_abs)
        max_abs = fabs(normal_vector[i]);

/*
* If max_abs is close to zero, something has gone wrong.
* In particular, we don't allow m(b) = b. For an orbifold, we should
* have already moved the basepoint away from the singular set, and
* as it moves towards a maximum of the injectivity radius it should
* go still further from the singular set.
*/
if (max_abs < ERROR_EPSILON)
    return func_failed;

/*
* Divide the normal vector by max_abs.
*/
for (i = 0; i < 4; i++)
    normal_vector[i] /= max_abs;

return func_OK;
}

static void compute_vertex_to_hyperplane_distances(
    WEPolyhedron *polyhedron,
    O31Vector     normal_vector)

```

```

{
    WEVertex      *vertex;

    /*
     * Compute the inner product of each vertex's coordinates x[] with the
     * normal_vector.  Because each vertex lies in the 3-space x[0] == 1,
     * the inner product <x, normal_vector> is proportional to the
     * 3-dimensional Euclidean distance in the projective model from the
     * vertex to the Dirichlet plane defined by the normal_vector.
     *
     * For numerical considerations, see compute_normal_to_Dirichlet_plane()
     * above.  (I may eventually also have to do some additional numerical
     * thinking here.)
     */

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)
    {
        vertex->distance_to_plane = o3l_inner_product(vertex->x, normal_vector);

        /*
         * Decide whether the vertex lies beyond (+1), beneath (-1),
         * or on (0) the Dirichlet plane.  Allow for possible roundoff
         * error.
         */
        if (vertex->distance_to_plane > polyhedron->vertex_epsilon)
            vertex->which_side_of_plane = +1;
        else if (vertex->distance_to_plane < - polyhedron->vertex_epsilon)
            vertex->which_side_of_plane = -1;
        else
            vertex->which_side_of_plane = 0;
    }
}

static Boolean positive_vertices_exist(
    WEPolyhedron *polyhedron)
{
    WEVertex      *vertex;
    Boolean        positive_vertices_exist,
                  negative_vertices_exist;

    positive_vertices_exist = FALSE;
    negative_vertices_exist = FALSE;

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)
    {
        if (vertex->which_side_of_plane == +1)
            positive_vertices_exist = TRUE;

        if (vertex->which_side_of_plane == -1)
            negative_vertices_exist = TRUE;
    }

    if (negative_vertices_exist == FALSE)
        uFatalError("positive_vertices_exist", "Dirichlet_construction");

    return positive_vertices_exist;
}

static void cut_edges(
    WEPolyhedron *polyhedron)
{
    WEEdge        *edge;
    int            i,
                  j;
    double         t;
    O3lVector      cut_point;

    for (edge = polyhedron->edge_list_begin.next;

```

```

    edge != &polyhedron->edge_list_end;
    edge = edge->next)

    for (i = 0; i < 2; i++)

        if (edge->v[i]->which_side_of_plane == -1
            && edge->v[!i]->which_side_of_plane == +1)
        {
            /*
             * Place the new vertex at the point where
             * the edge crosses the Dirichlet plane.
             */

            t = (
                    0.0 - edge->v[tail]->distance_to_plane) /
                (edge->v[tip]->distance_to_plane - edge->v[tail]->distance_to_plane);

            for (j = 0; j < 4; j++)
                cut_point[j] = (1.0 - t) * edge->v[tail]->x[j] + t * edge->v[tip]->x[j]

        }

    split_edge(edge, cut_point, TRUE);
}

void split_edge(
    WEEdge      *old_edge,
    O3lVector    cut_point,
    Boolean      set_Dirichlet_construction_fields)
{
    /*
     * This function is also called in Dirichlet_extras.c.  94/10/4  JRW
     */

    /*
     * Introduce a new vertex in old_edge at the cut_point.
     *
     * before
     *
     * after
     *
     * old edge
     *
     * new vertex
     *
     * new edge
     */

    WEEdge      *new_edge,
                *left_neighbor,
                *right_neighbor;
    WEVertex     *new_vertex;

    /*
     * Allocate space for the new_edge and new_vertex.
     */

    new_edge = NEW_STRUCT(WEEdge);
    new_vertex = NEW_STRUCT(WEVertex);

    /*
     * Set the fields of the new_edge.
     */

    new_edge->v[tail] = old_edge->v[tail];
    new_edge->v[tip] = new_vertex;

```

```

new_edge->e[tail][left] = old_edge->e[tail][left];
new_edge->e[tail][right] = old_edge->e[tail][right];
new_edge->e[tip][left] = old_edge;
new_edge->e[tip][right] = old_edge;

new_edge->f[left] = old_edge->f[left];
new_edge->f[right] = old_edge->f[right];

/*
 * Adjust the fields of the old_edge.
 */

old_edge->v[tail] = new_vertex;

old_edge->e[tail][left] = new_edge;
old_edge->e[tail][right] = new_edge;

/*
 * Adjust the fields of the other two edges which "see" new_edge.
 */

left_neighbor = new_edge->e[tail][left];
if (left_neighbor->v[tip] == new_edge->v[tail])
    left_neighbor->e[tip][left] = new_edge;
else
if (left_neighbor->v[tail] == new_edge->v[tail])
    left_neighbor->e[tail][right] = new_edge;
else
    uFatalError("split_edge", "Dirichlet_construction");

right_neighbor = new_edge->e[tail][right];
if (right_neighbor->v[tip] == new_edge->v[tail])
    right_neighbor->e[tip][right] = new_edge;
else
if (right_neighbor->v[tail] == new_edge->v[tail])
    right_neighbor->e[tail][left] = new_edge;
else
    uFatalError("split_edge", "Dirichlet_construction");

/*
 * Set the fields for the new vertex.
 */

o3l_copy_vector(new_vertex->x, cut_point);

if (set_Dirichlet_construction_fields)
{
    new_vertex->distance_to_plane = 0.0;
    new_vertex->which_side_of_plane = 0;
}

/*
 * Insert new_edge and new_vertex on their respective lists.
 * Note that the new_edge goes on the list just before the old_edge;
 * this is fine because we don't need to check the new_edge in the
 * loop in cut_edges() above, or in subdivide_edges_where_necessary()
 * in Dirichlet_extras.c.
 */

INSERT_BEFORE(new_edge, old_edge);
INSERT_BEFORE(new_vertex, old_edge->v[tip]);

/*
 * Dirichlet_construction.c isn't maintaining the faces' num_sides
 * fields, but Dirichlet_extras.c is.
 */

old_edge->f[left]->num_sides++;
old_edge->f[right]->num_sides++;
}

static FuncResult cut_faces(

```



```

    WEPolyhedron    *polyhedron)
{
    WEFace    *face;

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

        if (cut_face_if_necessary(face, TRUE) == func_failed)

            return func_failed;

    return func_OK;
}

/*
 * cut_face_if_necessary() returns func_failed if any topological
 * abnormality is found.  See details below.
 * It returns func_OK otherwise.
 */

FuncResult cut_face_if_necessary(
    WEFace    *face,
    Boolean    called_from_Dirichlet_construction)
{
    /*
     * Dirichlet_extras.c also calls cut_face_if_necessary().
     * The argument called_from_Dirichlet_construction tells who's called us,
     * so we can handle the face pairing accordingly.  94/10/5  JRW
     */

    int        i,
               count;
    WEEdge    *edge,
               *edge_before_vertex[2],
               *edge_after_vertex[2],
               *temp_edge;
    WEEdge    *new_edge;
    WEFace    *new_face;

    /*
     * Edges passing from the negative to the positive side of the
     * hyperplane have already been cut in cut_edges(), so we expect
     * each face to be of one of the following types.  A "0-vertex" is
     * a vertex whose which_side_of_plane field is 0.
     *
     * No 0-vertices.  The face should be left alone.
     *
     * One 0-vertex.  The face should be left alone.
     *
     * Two 0-vertices.
     *
     *     If the 0-vertices are adjacent, the face should be left alone.
     *
     *     If the 0-vertices are not adjacent, the face should be cut.
     *
     * In each of the above cases, we return func_OK.
     * If more than two 0-vertices occur, we return func_failed.
     * (For example, roundoff errors in especially tiny faces might
     * cause a face to have three 0-vertices.)
     */

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

    /*
     * Count the number of 0-vertices.
     */
    count = 0;
    edge = face->some_edge;

```

```

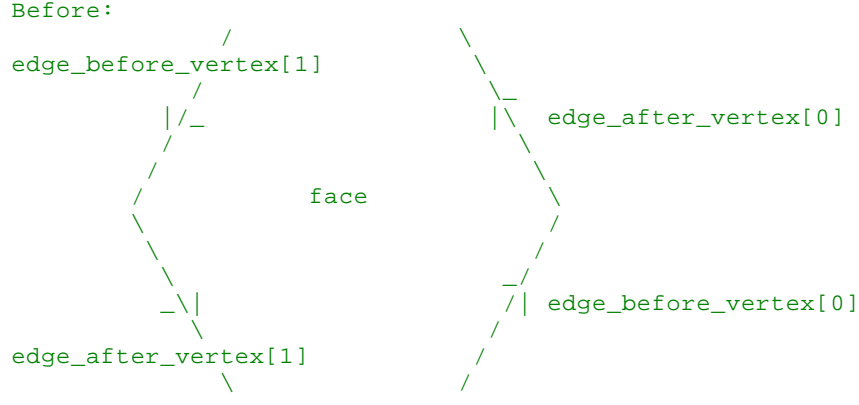
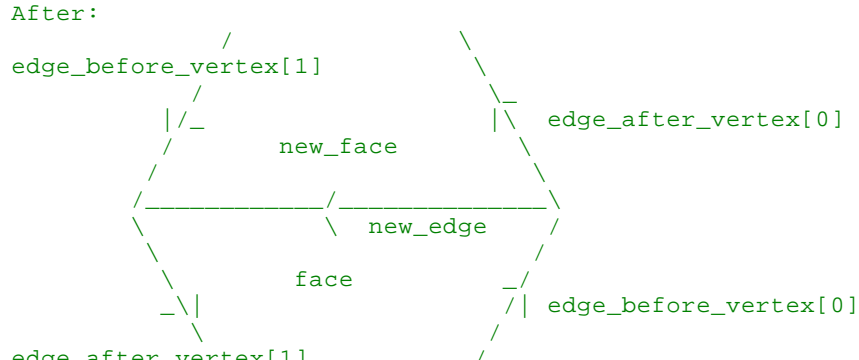
do
{
    if (edge->v[tip]->which_side_of_plane == 0)
    {
        /*
         * If this is our third 0-vertex, something has gone
         * wrong with the roundoff errors.
         */
        if (count == 2)
            return func_failed;
        /*
         * Note which edge precedes the 0-vertex.
         */
        edge_before_vertex[count] = edge;
        count++;
    }
    edge = edge->e[tip][left];
} while (edge != face->some_edge);

/*
 * If there are fewer than two 0-vertices, return.
 */
if (count < 2)
    return func_OK;

/*
 * Note which edges follow the 0-vertices.
 */

for (i = 0; i < 2; i++)
    edge_after_vertex[i] = edge_before_vertex[i]->e[tip][left];

/*
 * If the 0-vertices are adjacent, return.
 */
for (i = 0; i < 2; i++)
    if (edge_after_vertex[i] == edge_before_vertex[!i])
        return func_OK;

/*
 * Cut the face in two by introducing a new edge connecting
 * the two 0-vertices.
 */
/*
 * Before:
 */

/*
 * After:
 */


```

```

    *
    * \
    * /

    /*
    * First, for convenience, make sure the lower half is the
    * half we want to keep. (Because there are precisely two
    * incident 0-vertices, and no edge connects minus directly to plus,
    * we know that all the signs on a given side of the new_edge
    * must be the same.)
    */
    if (edge_before_vertex[0]->v[tail]->which_side_of_plane == -1
    && edge_after_vertex [0]->v[tip] ->which_side_of_plane == +1)
    {
        /*
        * Great. This is what we want. Do nothing.
        */
    }
    else
    if (edge_before_vertex[0]->v[tail]->which_side_of_plane == +1
    && edge_after_vertex [0]->v[tip] ->which_side_of_plane == -1)
    {
        /*
        * Swap the sides.
        */

        temp_edge          = edge_before_vertex[0];
        edge_before_vertex[0] = edge_before_vertex[1];
        edge_before_vertex[1] = temp_edge;

        temp_edge          = edge_after_vertex[0];
        edge_after_vertex[0] = edge_after_vertex[1];
        edge_after_vertex[1] = temp_edge;
    }
    else
    /*
    * The two sides have the same signs. Something has gone wrong.
    */
    return func_failed;

    /*
    * Now allocate and install the new_edge and new_face.
    */

    new_edge = NEW_STRUCT(WEEdge);
    new_face = NEW_STRUCT(WEFace);

    new_edge->v[tail]    = edge_before_vertex[0]->v[tip];
    new_edge->v[tip]     = edge_before_vertex[1]->v[tip];

    new_edge->e[tail][left]    = edge_before_vertex[0];
    new_edge->e[tail][right]   = edge_after_vertex[0];
    new_edge->e[tip][left]    = edge_after_vertex[1];
    new_edge->e[tip][right]   = edge_before_vertex[1];

    edge_before_vertex[0]->e[tip][left]    = new_edge;
    edge_after_vertex [0]->e[tail][left]   = new_edge;
    edge_before_vertex[1]->e[tip][left]    = new_edge;
    edge_after_vertex [1]->e[tail][left]   = new_edge;

    for (edge = edge_after_vertex[0]; edge != new_edge; edge = edge->e[tip][left])
        edge->f[left] = new_face;

    new_edge->f[left]    = face;
    new_edge->f[right]   = new_face;

    new_face->some_edge = new_edge;
    face    ->some_edge = new_edge;

    /*
    * How we handle the face pairings depends on whether we were called
    * from Dirichlet_construction.c or Dirichlet_extras.c.
    */
    if (called_from_Dirichlet_construction == TRUE)
    {

```

```

    /*
     * new_face will be removed when we clear out all the vertices, edges
     * and faces detached by the cut. So we set its mate and group_element
     * fields to NULL.
     *
     * face keeps its same mate, which may or may not survive the cut.
     * (But at least if face->mate gets cut in two, the half that's
     * retained will still be called face->mate.)
     */
    new_face->mate = NULL;
    new_face->group_element = NULL;

    /*
     * If face has a mate, we can no longer guarantee that it's a subset
     * of face under the action of the group_element.
     */
    if (face->mate != NULL)
        face->mate->clean = FALSE;
}
else /* called from Dirichlet_extras.c */
{
    face->mate = new_face;
    new_face->mate = face;

    new_face->group_element = NEW_STRUCT(O3lMatrix);
    o3l_copy(*new_face->group_element, *face->group_element);
}

/*
 * We may insert new_face before face, because there is no need for
 * cut_faces() to check it. new_edge may go anywhere.
 */
INSERT_BEFORE(new_edge, edge_before_vertex[0]);
INSERT_BEFORE(new_face, face);

return func_OK;
}

void all_edges_counterclockwise(
    WEFace *face,
    Boolean redirect_neighbor_fields)
{
    WEEdge *edge;

    edge = face->some_edge;
    do
    {
        if (edge->f[left] != face)
            redirect_edge(edge, redirect_neighbor_fields);
        edge = edge->e[tip][left];
    } while (edge != face->some_edge);
}

void redirect_edge(
    WEEdge *edge,
    Boolean redirect_neighbor_fields)
{
    WEVertex *temp_vertex;
    WEEdge *temp_edge;
    WEFace *temp_face;

    /*
     * Swap the tip and tail vertices.
     */
    temp_vertex = edge->v[tail];
    edge->v[tail] = edge->v[tip];
    edge->v[tip] = temp_vertex;

    /*
     * Swap the tail-left and tip-right edges.
     */
    temp_edge = edge->e[tail][left];

```

```

edge->e[tail][left]      = edge->e[tip][right];
edge->e[tip][right]      = temp_edge;

/*
 * Swap the tail-right and tip-left edges.
 */
temp_edge                = edge->e[tail][right];
edge->e[tail][right]      = edge->e[tip][left];
edge->e[tip][left]        = temp_edge;

/*
 * Swap the left and right faces.
 */
temp_face                = edge->f[left];
edge->f[left]             = edge->f[right];
edge->f[right]            = temp_face;

/*
 * When called from Dirichlet_extras.c,
 * we need to preserve some additional fields.
 */
if (redirect_neighbor_fields)
{
    WEEdge                *nbr_edge;
    WEEdgeSide            side,
                        nbr_side;
    WEEdge                *temp_edge;
    Boolean                temp_boolean;

    /*
     * Note that the following code works even if
     *
     * (1) the two sides of the edge are glued to each other, or
     *
     * (2) one or both of the sides is glued to itself.
     *
     * To convince yourself of this, trace through the following code
     * and note that in the degenerate cases the preserves_sides and
     * preserves_direction flags each get toggled twice.
     */

    /*
     * Fix up the neighbors.
     * Toggle their preserves_sides and preserves_direction fields,
     * but leave their neighbor and preserves_orientation field alone.
     */

    for (side = 0; side < 2; side++)
    {
        nbr_edge = edge->neighbor[side];
        nbr_side = (edge->preserves_sides[side] ? side : !side);

        nbr_edge->preserves_sides[nbr_side]
            = ! nbr_edge->preserves_sides[nbr_side];

        nbr_edge->preserves_direction[nbr_side]
            = ! nbr_edge->preserves_direction[nbr_side];
    }

    /*
     * Toggle our own preserves_sides and preserves_direction fields.
     */

    for (side = 0; side < 2; side++)
    {
        edge->preserves_sides[side]      = ! edge->preserves_sides[side];
        edge->preserves_direction[side] = ! edge->preserves_direction[side];
    }

    /*
     * Swap the left and right values.
     */

    temp_edge                = edge->neighbor[left];

```

```

    edge->neighbor[left]    = edge->neighbor[right];
    edge->neighbor[right]   = temp_edge;

    temp_boolean            = edge->preserves_sides[left];
    edge->preserves_sides[left] = edge->preserves_sides[right];
    edge->preserves_sides[right] = temp_boolean;

    temp_boolean            = edge->preserves_direction[left];
    edge->preserves_direction[left] = edge->preserves_direction[right];
    edge->preserves_direction[right] = temp_boolean;

    temp_boolean            = edge->preserves_orientation[left];
    edge->preserves_orientation[left] = edge->preserves_orientation[right];
    edge->preserves_orientation[right] = temp_boolean;
}
}

```

```

static FuncResult check_topology_of_cut(
    WEPolyhedron *polyhedron)
{
    int          num_zero_edges,
                count;
    WEVertex     *vertex,
                *tip_vertex;
    WEEdge       *edge,
                *starting_edge;

    /*
     * Define a 0-vertex to be a vertex with which_side_of_plane == 0.
     *
     * Define a 0-edge to be an edge incident to two 0-vertices.
     *
     * We'll be cutting along the union of the 0-edges. We want to
     * verify that it's a simple closed curve. We do this in two steps:
     *
     * (1) Verify that precisely two 0-edges are incident to each 0-vertex.
     *     This insures that the cut locus is a (possibly empty) union of
     *     simple closed curves.
     *
     * (2) Verify that the cut locus is connected.
     */

    /*
     * Initialize the zero_order fields to 0.
     */

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

        vertex->zero_order = 0;

    /*
     * Count the number of 0-edges adjacent to each 0-vertex.
     *
     * While we're at it, keep a global count of the number of the
     * number of 0-edges, for use below in checking that the cut locus
     * is connected.
     */

    num_zero_edges = 0;

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

        if (edge->v[tail]->which_side_of_plane == 0
            && edge->v[tip]->which_side_of_plane == 0)
        {
            edge->v[tail]->zero_order++;
            edge->v[tip]->zero_order++;

            num_zero_edges++;
        }
    }
}

```

```

    }

    /*
     * Check that all 0-vertices are incident to precisely two 0-edges.
     */

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

        if (vertex->which_side_of_plane == 0
            && vertex->zero_order != 2)

            return func_failed;

    /*
     * We now know that the cut locus is a union of simple
     * closed curves. We want to check that it is connected.
     */

    /*
     * Find a starting edge.
     */

    starting_edge = NULL;

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

        if (edge->v[tail]->which_side_of_plane == 0
            && edge->v[tip]->which_side_of_plane == 0)
        {
            starting_edge = edge;
            break;
        }

    /*
     * If we didn't find a starting edge, something went horribly wrong.
     */

    if (starting_edge == NULL)
        uFatalError("check_topology_of_cut", "Dirichlet_construction");

    /*
     * Traverse the cut locus, beginning at the starting_edge.
     * Count the number of edges in the locus.
     */

    count = 0;

    edge = starting_edge;

    do
    {
        /*
         * Count this edge.
         */
        count++;

        /*
         * Move on to the next edge.
         * To do so, rotate clockwise until we find another 0-edge.
         * For convenience, we reorient the edges as necessary as we
         * go along, so they always point towards the tip_vertex.
         * Once we finally get to another 0-edge we reorient it once
         * more, to preserve the direction of the curve (i.e. we now
         * want to point away from the old tip_vertex, and towards
         * what will become the new tip_vertex).
         */
        tip_vertex = edge->v[tip];
        do
        {
            edge = edge->e[tip][left];

```

```

        if (edge->v[tip] != tip_vertex)
            redirect_edge(edge, FALSE);
    } while (edge->v[tail]->which_side_of_plane != 0);
    redirect_edge(edge, FALSE);

} while (edge != starting_edge);

/*
 * The cut locus will be connected iff count == num_zero_edges.
 */

if (count == num_zero_edges)
    return func_OK;
else
    return func_failed;
}

static void install_new_face(
    WEPolyhedron *polyhedron,
    WEFace *new_face)
{
    WEFace *face;
    WEEdge *edge;
    WEVertex *vertex;
    WEFace *dead_face;
    WEEdge *dead_edge;
    WEVertex *dead_vertex;
    WEVertex *tip_vertex;
    WEEdge *left_edge,
            *right_edge;
    int i;

    /*
     * The overall plan here is to remove those faces, edges and vertices
     * (in that order) which have been cut off by the slicing plane,
     * and install the new_face.
     */

    /*
     * We'll want to remove precisely those faces which are incident to
     * at least one positive vertex. (For a proof, see the corollary
     * in the documentation in slice_with_hyperplane().) To find these
     * faces, we'll scan the list of edges, and whenever an edge is
     * incident to at least one positive vertex, we'll see the neighboring
     * faces' to_be_removed fields to TRUE. (This is easier than
     * traversing the perimeter of each face explicitly, worrying about
     * the directions of the edges.)
     */

    /*
     * Initialize all faces' to_be_removed fields to FALSE.
     */

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

        face->to_be_removed = FALSE;

    /*
     * Scan the edge list to locate edges -- and hence faces -- which
     * are incident to at least one positive vertex.
     */

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

        if (edge->v[tail]->which_side_of_plane == +1
            || edge->v[tip]->which_side_of_plane == +1)
        {
            edge->f[left]->to_be_removed = TRUE;
            edge->f[right]->to_be_removed = TRUE;

```



```

    }

/*
 * Set the edge->f[] fields for edges which will be incident to the
 * new_face. Also make sure the new_face's some_edge field sees
 * an appropriate edge (note that it's easier to keep setting
 * new_face->some_edge over and over than to check whether it's
 * already been set).
 */

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

    if (edge->v[tail]->which_side_of_plane == 0
        && edge->v[tip]->which_side_of_plane == 0)

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

            if (edge->f[i]->to_be_removed == TRUE)
            {
                edge->f[i] = new_face;
                new_face->some_edge = edge;
            }

/*
 * Delete the faces which are being removed.
 */

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

    if (face->to_be_removed == TRUE)
    {
        /*
         * If the face has a mate, set the mate's mate field to NULL
         * and its clean field to FALSE.
         */
        if (face->mate != NULL && face->mate != face)
        {
            face->mate->mate = NULL;
            face->mate->clean = FALSE;
        }

        /*
         * If the face has a group_element, free it.
         */
        if (face->group_element != NULL)
            my_free(face->group_element);

        /*
         * Remove the face from the doubly-linked list. First set
         * face to face->prev so the loop will continue correctly.
         */
        dead_face = face;
        face = face->prev;
        REMOVE_NODE(dead_face);

        /*
         * Delete the face.
         */
        my_free(dead_face);
    }

/*
 * Delete the edges which are being removed.
 */

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

    if (edge->v[tail]->which_side_of_plane == +1

```

```

    || edge->v[tip] ->which_side_of_plane == +1)
{
    /*
     * If this edge is indident to a 0-vertex,
     * fix up the neighbors' edge->e[][] fields.
     */

    /*
     * If it's the tail which is incident to a 0-vertex, call
     * redirect_edge(). This simplifies the subsequent code,
     * because we may assume that if a 0-vertex is incident to the
     * edge, it will be at the tip. Note that redirect_edge() will
     * be interchanging some dangling WEFace pointers, but that's OK.
     */
    if (edge->v[tail]->which_side_of_plane == 0)
        redirect_edge(edge, FALSE);

    /*
     * Is there a 0-vertex at the tip?
     */

    tip_vertex = edge->v[tip];

    if (tip_vertex->which_side_of_plane == 0)
    {
        /*
         * Who are the neighbors?
         */
        left_edge = edge->e[tip][left];
        right_edge = edge->e[tip][right];

        /*
         * Let the left_edge see the right_edge.
         */
        if (left_edge->v[tip] == tip_vertex)
            left_edge->e[tip][right] = right_edge;
        else
            if (left_edge->v[tail] == tip_vertex)
                left_edge->e[tail][left] = right_edge;
        else
            uFatalError("install_new_face", "Dirichlet_construction");

        /*
         * Let the right_edge see the left_edge.
         */
        if (right_edge->v[tip] == tip_vertex)
            right_edge->e[tip][left] = left_edge;
        else
            if (right_edge->v[tail] == tip_vertex)
                right_edge->e[tail][right] = left_edge;
        else
            uFatalError("install_new_face", "Dirichlet_construction");
    }

    /*
     * Remove the edge from the doubly-linked list. First set
     * edge to edge->prev so the loop will continue correctly.
     */
    dead_edge = edge;
    edge = edge->prev;
    REMOVE_NODE(dead_edge);

    /*
     * Delete the edge.
     */
    my_free(dead_edge);
}

/*
 * Delete the vertices which are being removed.
 */

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

```

```

    vertex = vertex->next)

    if (vertex->which_side_of_plane == +1)
    {
        /*
         * Remove the vertex from the doubly-linked list. First set
         * vertex to vertex->prev so the loop will continue correctly.
         */
        dead_vertex = vertex;
        vertex = vertex->prev;
        REMOVE_NODE(dead_vertex);

        /*
         * Delete the vertex.
         */
        my_free(dead_vertex);
    }

    /*
     * Install the new_face in the doubly-linked list.
     */
    INSERT_BEFORE(new_face, &polyhedron->face_list_end);
}

static Boolean face_still_exists(
    WEPolyhedron    *polyhedron,
    WEFace          *face0)
{
    /*
     * Check whether face0 is still part of the polyhedron.
     */

    WEFace    *face;

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

        if (face == face0)

            return TRUE;

    return FALSE;
}

static Boolean has_hyperideal_vertices(
    WEPolyhedron    *polyhedron)
{
    WEVertex        *vertex;

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

        if (o3l_inner_product(vertex->x, vertex->x) > HYPERIDEAL_EPSILON)

            return TRUE;

    return FALSE;
}

static void compute_all_products(
    WEPolyhedron    *polyhedron,
    MatrixPairList   *product_list)
{
    MatrixPairList   current_list;
    MatrixPair        *product_tree;

    /*
     * Compute a MatrixPairList containing all current face pairings.
     */

```

```

poly_to_current_list(polyhedron, &current_list);

/*
 * Compute all products of two group elements from the current_list.
 *
 * Record the products on a binary tree instead of a doubly-linked list,
 * so that:
 *
 * (1) We can check for duplications in log(n) time instead
 *     of linear time.
 *
 * (2) The final list will be sorted by height (cf. the height field
 *     in WEFace). My expectation is that intersecting the polyhedron
 *     with the lowest height group elements first will minimize the
 *     time spent slicing. That is, we'll make the most important
 *     cuts first, rather than creating a lot of more distant faces
 *     which are themselves later cut off. (I don't have any firm
 *     evidence that this will be the case, but it seems likely, and
 *     in any case it costs us nothing.)
 */

current_list_to_product_tree(&current_list, &product_tree);

/*
 * Transfer the products from the product_tree to the product_list.
 */

product_tree_to_product_list(product_tree, product_list);

/*
 * We're done with the current_list;
 */
free_matrix_pairs(&current_list);
}

static void poly_to_current_list(
    WEPolyhedron *polyhedron,
    MatrixPairList *current_list)
{
    WEFace *face;
    MatrixPair *matrix_pair;

    /*
     * Initialize the current_list.
     */
    current_list->begin.prev = NULL;
    current_list->begin.next = &current_list->end;
    current_list->end .prev = &current_list->begin;
    current_list->end .next = NULL;

    /*
     * Use the face->copied fields to avoid copying both a face and its
     * mate as separate MatrixPairs. First initialize all face->copied
     * fields to FALSE.
     */

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

        face->copied = FALSE;

    /*
     * Go down the list of faces, skipping faces of the original cube.
     * For each face which hasn't already been done, copy it's group_element
     * to a MatrixPair and append the MatrixPair to the current_list.
     * Set the face->copied field to TRUE, and if the face has a mate,
     * set its mate's copied field to TRUE too.
     */

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

```

```

        face = face->next)

    if (face->group_element != NULL && face->copied == FALSE)
    {
        matrix_pair = NEW_STRUCT(MatrixPair);
        o3l_copy(matrix_pair->m[0], *face->group_element);
        o3l_invert(matrix_pair->m[0], matrix_pair->m[1]);
        matrix_pair->height = matrix_pair->m[0][0][0];
        INSERT_BEFORE(matrix_pair, &current_list->end);

        face->copied = TRUE;
        if (face->mate != NULL)
            face->mate->copied = TRUE;
    }
}

static void current_list_to_product_tree(
    MatrixPairList *current_list,
    MatrixPair **product_tree)
{
    MatrixPair *matrix_pair_a,
               *matrix_pair_b;
    int i,
        j;
    O3lMatrix product;

    /*
     * Initialize the product_tree to NULL.
     */
    *product_tree = NULL;

    /* For each pair {a, A}, {b, B} of MatrixPairs on the current_list,
     * add the MatrixPairs {ab, BA}, {aB, bA}, {Ab, Ba} and {AB, ba}
     * to product_list if they aren't already there.
     *
     * Note that once we've considered the ordered pair ({a, A}, {b, B})
     * of MatrixPairs, there is no need to consider the ordered pair
     * ({b, B}, {a, A}). It would generate the MatrixPairs
     * {ba, AB}, {bA, aB}, {Ba, Ab} and {BA, ab}, which are the same as
     * those generated by ({a, A}, {b, B}), only within each resulting
     * MatrixPair the order of the O3lMatrices is reversed. Therefore we
     * consider only order pairs ({a, A}, {b, B}) where the the MatrixPair
     * {a, A} occurs no later than {b, B} in the current_list.
     */

    for (matrix_pair_a = current_list->begin.next;
         matrix_pair_a != &current_list->end;
         matrix_pair_a = matrix_pair_a->next)

        for (matrix_pair_b = matrix_pair_a;
             matrix_pair_b != &current_list->end;
             matrix_pair_b = matrix_pair_b->next)

            for (i = 0; i < 2; i++) /* which element of matrix_pair_a */
                for (j = 0; j < 2; j++) /* which element of matrix_pair_b */
                {
                    precise_o3l_product(matrix_pair_a->m[i], matrix_pair_b->m[j], product);

                    if (already_on_product_tree(product, *product_tree) == FALSE)

                        add_to_product_tree(product, product_tree);
                }
    }
}

static Boolean already_on_product_tree(
    O3lMatrix product,
    MatrixPair *product_tree)
{
    MatrixPair *subtree_stack,
               *subtree;
    int i;

```

```

/*
 * Does the O3lMatrix product already appear on the product_tree?
 */

/*
 * Implement the recursive search algorithm using our own stack
 * rather than the system stack, to avoid the possibility of a
 * stack/heap collision.
 */

/*
 * Initialize the stack to contain the whole product_tree.
 */
subtree_stack = product_tree;
if (product_tree != NULL)
    product_tree->next_subtree = NULL;

/*
 * Process the subtrees on the stack one at a time.
 */
while (subtree_stack != NULL)
{
    /*
     * Pull a subtree off the stack.
     */
    subtree          = subtree_stack;
    subtree_stack    = subtree_stack->next_subtree;
    subtree->next_subtree = NULL;

    /*
     * If the product could possible appear on the left and/or right
     * subtrees, add them to the stack.
     */
    if (subtree->left_child != NULL
        && product[0][0] < subtree->height + MATRIX_EPSILON)
    {
        subtree->left_child->next_subtree = subtree_stack;
        subtree_stack = subtree->left_child;
    }
    if (subtree->right_child != NULL
        && product[0][0] > subtree->height - MATRIX_EPSILON)
    {
        subtree->right_child->next_subtree = subtree_stack;
        subtree_stack = subtree->right_child;
    }

    /*
     * Check the subtree's root.
     */
    for (i = 0; i < 2; i++)
        if (o3l_equal(product, subtree->m[i], MATRIX_EPSILON) == TRUE)
            return TRUE;
}

return FALSE;
}

static void add_to_product_tree(
    O3lMatrix    product,
    MatrixPair   **product_tree)
{
    MatrixPair   **home;
    double       product_height;

    /*
     * We need to find a home for the O3lMatrix product on the product_tree.
     *
     * We assume the product does not already appear on the product_tree.
     * (The call to already_on_product_tree() must return FALSE in
     * current_list_to_product_tree() for this function to be called.)
     */

```

```

    product_height = product[0][0];

    home = product_tree;

    while (*home != NULL)
    {
        if (product_height < (*home)->height)
            home = &(*home)->left_child;
        else
            home = &(*home)->right_child;
    }

    (*home) = NEW_STRUCT(MatrixPair);
    o3l_copy((*home)->m[0], product);
    o3l_invert((*home)->m[0], (*home)->m[1]);
    (*home)->height = (*home)->m[0][0][0];
    (*home)->left_child = NULL;
    (*home)->right_child = NULL;
    (*home)->next_subtree = NULL;
    (*home)->prev = NULL;
    (*home)->next = NULL;
}

static void product_tree_to_product_list(
    MatrixPair *product_tree,
    MatrixPairList *product_list)
{
    /*
     * Initialize the product_list.
     */
    product_list->begin.prev = NULL;
    product_list->begin.next = &product_list->end;
    product_list->end .prev = &product_list->begin;
    product_list->end .next = NULL;

    /*
     * Transfer the MatrixPairs from the product_tree to the product_list,
     * maintaining their order. Set the left_child and right_child fields
     * to NULL as we go along.
     */
    append_tree_to_list(product_tree, &product_list->end);
}

static void append_tree_to_list(
    MatrixPair *product_tree,
    MatrixPair *list_end)
{
    MatrixPair *subtree_stack,
               *subtree;

    /*
     * Implement the recursive tree traversal using our own stack
     * rather than the system stack, to avoid the possibility of a
     * stack/heap collision.
     */

    /*
     * Initialize the stack to contain the whole product_tree.
     */
    subtree_stack = product_tree;
    if (product_tree != NULL)
        product_tree->next_subtree = NULL;

    /*
     * Process the subtrees on the stack one at a time.
     */
    while (subtree_stack != NULL)
    {
        /*
         * Pull a subtree off the stack.
         */
        subtree = subtree_stack;

```

```

    subtree_stack      = subtree_stack->next_subtree;
    subtree->next_subtree = NULL;

    /*
     * If it has no further subtrees, append it to the list.
     *
     * Otherwise break it into three chunks:
     *
     *     the left subtree
     *     this node
     *     the right subtree
     *
     * and push them onto the stack in reverse order, so that they'll
     * come off in the correct order. Set this node's left_child and
     * right_child fields to NULL, so the next time it comes off the
     * stack we'll know the subtrees have been accounted for.
     */
    if (subtree->left_child == NULL && subtree->right_child == NULL)
    {
        INSERT_BEFORE(subtree, list_end);
    }
    else
    {
        /*
         * Push the right subtree (if any) onto the stack.
         */
        if (subtree->right_child != NULL)
        {
            subtree->right_child->next_subtree = subtree_stack;
            subtree_stack = subtree->right_child;
            subtree->right_child = NULL;
        }

        /*
         * Push this node onto the stack.
         * (Its left_child and right_child fields will soon be NULL.)
         */
        subtree->next_subtree = subtree_stack;
        subtree_stack = subtree;

        /*
         * Push the left subtree (if any) onto the stack.
         */
        if (subtree->left_child != NULL)
        {
            subtree->left_child->next_subtree = subtree_stack;
            subtree_stack = subtree->left_child;
            subtree->left_child = NULL;
        }
    }
}

static FuncResult check_faces(
    WEPolyhedron *polyhedron)
{
    WEFace *face;
    Boolean face_was_pared;

    /*
     * The face->clean fields keep track of which faces are known to be
     * subsets of their mates under the action of the group_element.
     * When all face->clean fields are TRUE, we've found the Dirichlet
     * domain.
     */

    /*
     * Initialize all face->clean fields to FALSE.
     */

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

```



```

    face->clean = FALSE;

/*
 * Go down the doubly-linked list of faces, looking for a dirty one.
 */

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

/*
 * If a dirty one is found . . .
 */

if (face->clean == FALSE)
{
    /*
     * . . . see whether some group element will pare away
     * some or all of it.
     *
     * There are two possibilities.
     *
     * (1) The face was clean after all, and we just didn't know it.
     *     In this case, pare_face() sets face->clean to TRUE and
     *     *face_was_pared to FALSE, and returns func_OK. The
     *     polyhedron remains unchanged. We keep going with the
     *     for(;;) loop.
     *
     * (2) The face really was dirty. In this case, pare_face()
     *     slices the polyhedron with a plane which lops off a
     *     nontrivial piece. If pare_face() encounters topological
     *     problems due to roundoff error, it returns func_failed,
     *     and so do we. Otherwise it sets the face->clean fields
     *     to FALSE for all affected faces, sets *face_was_pared to
     *     TRUE, and returns func_OK. At this point we have no
     *     idea which faces have been eliminated entirely, so we
     *     restart the for(;;) loop.
     */

    if (pare_face(face, polyhedron, &face_was_pared) == func_failed)
        return func_failed;

    if (face_was_pared == TRUE)
        face = &polyhedron->face_list_begin;
}

/*
 * The above for(;;) loop will terminate only when all the face->clean
 * fields are TRUE. So we can return func_OK.
 */

return func_OK;
}

static FuncResult pare_face(
    WEFace          *face,
    WEPolyhedron    *polyhedron,
    Boolean          *face_was_pared)
{
    /*
     * pare_face()'s mission is to decide whether face is a subset of its
     * mate under the action of the group_element. (If its mate does not
     * exist, then clearly it's not a subset.)
     *
     * If face is a subset of its mate, set face->clean to TRUE and
     * *face_was_pared to FALSE. (Setting *face_was_pared to FALSE
     * signifies that we didn't have to do anything: the face was
     * already a subset of its mate under the action of the group_element,
     * even though we didn't know it.)
     *
     * If face is not a subset of its mate, find a new group element
     * which cuts off a part of face. Leave face->clean FALSE, and

```

```

    * set *face_was_pared to TRUE.
    *
    * If we encounter topological problems due to roundoff error,
    * return func_failed. Otherwise return func_OK.
    *
    * We split into two cases, according to whether face has a mate.
    */

if (face->mate != NULL)
    return pare_mated_face(face, polyhedron, face_was_pared);
else
    return pare_mateless_face(face, polyhedron, face_was_pared);
}

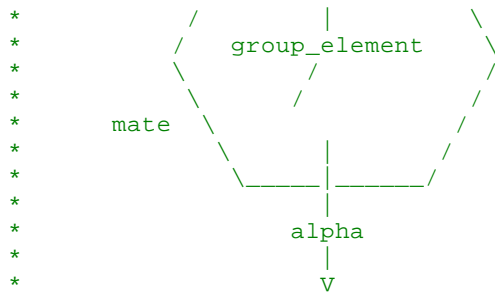
static FuncResult pare_mated_face(
WEFace      *face,
WEPolyhedron *polyhedron,
Boolean     *face_was_pared)
{
/*
 * The face and its mate are both convex sets -- both are
 * intersections of half planes. So to check that the face is
 * contained in the image of its mate under the action of the
 * group_element, it suffices to check that all the face's vertices
 * are contained in the image of the mate.
 *
 * Schematically, the polyhedron looks like this. Note that the
 * group_element takes the mate to the face, not the other way around.
 *
 *
 *

 *
 * Look at the image of the mate under the action of the group_element.
 * The figure shows what happens when the face is not a subset of
 * the image of the mate.
 *
 *

 *
 * One of the vertices of the face extends beyond one of the sides
 * of the image of the mate. Let alpha be the group_element
 * belonging to the face adjacent to that side of mate.
 *
 *

 *

```



```
* Consider the product beta = (group_element)(alpha). (Matrices act
* on column vectors on the left, so (group_element)(alpha) means
* first do alpha, then do group_element.)
```

```

*   Let V be the vertex in question -- the vertex of face which lies
*   outside the image of mate.

```

```
* distance(V, origin) = distance(V, group_element(origin))
```

- * because V lies on face.

```
* distance(V, group_element(origin)) > distance(V, beta(origin))
```

* because V lies beyond that side of the image of the mate.

* Combining the above two observations yields

```
* distance(V, origin) > distance(V, beta(origin))
```

```
*  which implies that the half space defined by beta will cut the
*  vertex V off of the polyhedron.
```

- * So our algorithm will be

```

*   for (each neighbor of the mate)
*       compute beta = group_element * alpha
*       for (each vertex V of the face)
*           if (beta cuts off V)
*               add faces determined by beta and its inverse
*               to the polyhedron
*               if roundoff-related errors occurred
*                   return func_failed
*               otherwise
*                   set *face_was_pared to TRUE
*                   return func_OK
*   set face->clean to TRUE
*   set *face_was_pared to FALSE
*   return func_OK

```

* Technical digression.

```
*      The old version of SnapPea used a "clever trick" to check that
*      a face is a subset of its mate.  I've chosen not to use that
*      trick here, and in this digression I will briefly explain why.
*      Feel free to skip this digression if you want.  I'm writing it
*      mainly for my own good, so if I come back a few years from now
*      and wonder why I didn't use the trick, I'll at least know what
*      I was thinking at the time.
```

```
*      The "clever trick" was to try to simultaneously traverse the
*      perimeters of the face and its mate, checking that the group
*      element beta (defined above) is already the group element
*      corresponding to one of the face's neighbors.  If this didn't
*      work, either because the faces didn't in fact match up, or
*      (less likely) because the edges weren't of order 3, then the
*      old code would fall back to an algorithm like the one described
*      above.
```

```
*      I've chosen not to use this "clever trick" in the present code
*      for the following reasons.
```

```

*      (1) For nonorientable manifolds, the code would get messier
*      because we wouldn't know a priori which way we need to
*      traverse the faces.
*
*      (2) The code wasn't that much more efficient to begin with.
*      To match two n-gons, it would have to do n matrix
*      multiplications. Each matrix multiplication requires
*      about  $4^3$  operations, so the run time is about  $O(n * 4^3)$ .
*      The simpler algorithm (the one used here) requires, in
*      addition, the computation of  $n^2$  inner products, to
*      check whether each of n points lies on the correct side
*      of n hyperplanes. This requires a time  $O(n^2 * 4)$ .
*      At first glance  $O(n^2)$  looks worse than  $O(n)$ , but we
*      have to remember that n is not a large number. One can
*      prove that for a typical polyhedron, the average value
*      of n is around 6. (For an atypical polyhedron it's even
*      less.) So the simpler algorithm requires little additional
*      time. Indeed, if one wanted to streamline it, one could
*      compute only the first column of beta, unless it turned
*      out that you needed the whole matrix to add a new face.
*      So the simpler algorithm might be even faster.
*
*      (3) Whether the old algorithm is faster than the new one isn't
*      completely clear, but given that the run times will be
*      similar, I feel that the simplicity of the code should
*      be the top priority.
*
*      End of technical digression.
*/

WEEdge      *edge;
O31Matrix   *alpha;

/*
*      Consider each neighbor of face->mate.
*/

edge = face->mate->some_edge;
do
{
    /*
    *      First an error check.
    */
    if (edge->f[left] == edge->f[right])
        uFatalError("pare_mated_face", "Dirichlet_construction");

    /*
    *      Find the element alpha defined above.
    */
    if (edge->f[left] == face->mate)
        alpha = edge->f[right]->group_element;
    else
        alpha = edge->f[left] ->group_element;

    /*
    *      Check whether this alpha defines a beta which cuts a vertex off
    *      of face. If so, intersect the polyhedron with the faces defined
    *      by beta and its inverse.
    *
    *      If topological problems due to roundoff error are
    *      encountered, return func_failed.
    */
    if (try_this_alpha(alpha, face, polyhedron, face_was_pared) == func_failed)
        return func_failed;

    /*
    *      If the face was actually pared, return func_OK.
    *      Otherwise keep going with the loop.
    */
    if (*face_was_pared == TRUE)
        return func_OK;

    /*
    *      Move on to the next edge.

```

```

    */
    if (edge->f[left] == face->mate)
        edge = edge->e[tip][left];
    else
        edge = edge->e[tail][right];
} while (edge != face->mate->some_edge);

/*
 * The face is a subset of the image of its mate.
 */
face->clean = TRUE;

/*
 * We didn't have to do anything to the polyhedron,
 * so set *face_was_pared to FALSE.
 */
*face_was_pared = FALSE;

/*
 * Given that we modified nothing, we could hardly have encountered
 * topological problems due to roundoff errors, so return func_OK.
 */
return func_OK;
}

static FuncResult pare_mateless_face(
WEFace          *face,
WEPolyhedron     *polyhedron,
Boolean         *face_was_pared)
{
    /*
     * The situation here is similar to that in pare_mated_face() above.
     * (In particular, you should understand pare_mated_face()'s
     * documentation before you read this.) The difference is that
     * mate has been entirely cut off by other group elements.
     * (Well, almost entirely cut off. A 1-dimensional subset might
     * still remain, but no 2-dimensional subset.) In the following
     * diagram, the plane of the nonexistent mate is shown by a line
     * of stars. (By the "plane of the nonexistent mate", we mean the
     * image of the plane of face under the action of the inverse of
     * face->group_element.)
     *
     *      \       /
     *       \     /
     *        \   /
     *         \|/
     *image of mate *****
     *
     *           ^
     *          / | \
     *         /  |  \
     *        /---\
     *       /-----\
     *      /             \
     *mate *               *
     *              *
     *                *
     *
     * Let P be a point which lies in the image of face under the action of
     * the inverse of face->group_element, but is not a point of the
     * polyhedron. Some face plane of the polyhedron, with group element
     * alpha, must exclude P from the polyhedron. (Proof: If P were on
     * the "correct" side of all face planes, then it would be contained
     * in the polyhedron, since the polyhedron is the intersection of the
     * halfspaces corresponding to its face planes.)
     *
     *                    conjugate
     *                   --/--> of
     *                  \  /  alpha
     *                 \|/
     *            image of mate

```

```

*****
*                                     P'
*      _____ face _____
*     /         ^         \
*    /   group_element   \
*   /                     \
*  *                       *
* mate *                   *
*      |                   |
*      P                   alpha
*      *                   V
*
This says that
distance(P, origin) > distance(P, alpha(origin))

Let P' be the image of P under the action of face->group_element,
that is, P' = face->group_element(P).

Let face->group_element act on the preceding inequality:
distance(P', group_element(origin))
> distance(P', group_element(alpha(origin)))

Because P' lies on face,
distance(P', origin) = distance(P', group_element(origin))

Substituting this equation into the preceding inequality yields
distance(P', origin) > distance(P', group_element(alpha(origin)))

This implies that beta = group_element * alpha will define a
plane which cuts P' off of the polyhedron. Because face is the
convex hull of its vertices, beta must also remove some vertex V
from the polyhedron.

How do we find the group element alpha? I don't know of any
clever way to find it, so we use brute force. We look at each
face in turn until we find one that works. Here's the algorithm:

for each alpha belonging to a face of the polyhedron
compute beta = group_element * alpha
for (each vertex V of the face)
if (beta cuts off V)
add faces determined by beta and its inverse
to the polyhedron
if roundoff-related errors occurred
return func_failed
otherwise
set *face_was_pared to TRUE
return func_OK
report a roundoff error, because we proved above that a
suitable alpha must exist
*/

WEFace      *facel;
O3lMatrix   *alpha;

/*
 * Consider each facel of the polyhedron.
 */
for (facel = polyhedron->face_list_begin.next;
     facel != &polyhedron->face_list_end;
     facel = facel->next)
{
    /*
     * Find alpha.
     */
    alpha = facel->group_element;

    /*
     * Check whether this alpha defines a beta which cuts a vertex off

```

```

    * of face. If so, intersect the polyhedron with the faces defined
    * by beta and its inverse.
    *
    * If topological problems due to roundoff error are
    * encountered, return func_failed.
    */
    if (try_this_alpha(alpha, face, polyhedron, face_was_pared) == func_failed)
        return func_failed;

    /*
    * If the face was actually pared, return func_OK.
    * Otherwise keep going with the loop.
    */
    if (*face_was_pared == TRUE)
        return func_OK;
}

/*
* We should never reach this point, because in the above documentation
* we proved that some alpha must define a beta which cuts off a vertex.
* So if we do end up at this point, it means that roundoff errors
* have corrupted our polyhedron.
*/
return func_failed;
}

static FuncResult try_this_alpha(
    O31Matrix      *alpha,
    WEFace         *face,
    WEPolyhedron   *polyhedron,
    Boolean        *face_was_pared)
{
    /*
    * try_this_alpha() performs the calculations common to
    * pare_mated_face() and pare_mateless_face().
    *
    * It computes beta = face->group_element * alpha (cf. the
    * documentation at the beginning of pare_mated_face()) and
    * checks whether beta cuts off any vertices of face.
    *
    * If beta cuts off vertices, then
    *     if topological problems due to roundoff error are
    *     encountered, it returns func_failed
    *     else it sets *face_was_pared to TRUE and returns func_OK
    * else
    *     it sets *face_was_pared to FALSE and returns func_OK.
    */

    WEVertex      *vertex;
    WEEdge         *edge;
    O31Matrix      beta;
    O31Vector      normal;
    MatrixPair     matrix_pair;

    /*
    * Compute beta = (group_element)(alpha) as explained in the
    * documentation in pare_mated_face().
    *
    * (If you have an overwhelming urge to optimize you could
    * compute only the first column of beta until you need the
    * rest, but for now I'll prefer simplicity over speed.)
    */
    precise_o31_product(*face->group_element, *alpha, beta);

    /*
    * Compute the normal vector to the hyperplane defined by beta.
    */
    compute_normal_to_Dirichlet_plane(beta, normal);

    /*
    * Consider each vertex of face.
    * (Technically speaking, we consider each edge, but then look
    * at the vertex on its counterclockwise end.)
    */

```

```

    */
    edge = face->some_edge;
do
{
    /*
    * Look at the vertex at the counterclockwise end of edge.
    */
    if (edge->f[left] == face)
        vertex = edge->v[tip];
    else
        vertex = edge->v[tail];

    /*
    * Does the vertex lie beyond the hyperplane determined by beta?
    */
    if (o3l_inner_product(vertex->x, normal) > polyhedron->vertex_epsilon)
    {
        /*
        * Great.
        * We've found a vertex which will be cut off by beta.
        */

        /*
        * Set up matrix_pair.m[0] and matrix_pair.m[1].
        */
        o3l_copy(matrix_pair.m[0], beta);
        o3l_invert(matrix_pair.m[0], matrix_pair.m[1]);

        /*
        * The other fields in matrix_pair aren't needed here.
        */
        matrix_pair.height = 0.0;
        matrix_pair.prev = NULL;
        matrix_pair.next = NULL;

        /*
        * Intersect the polyhedron with the half spaces.
        * If roundoff errors cause topological problems,
        * return func_failed.
        */
        if (intersect_with_halfspaces(polyhedron, &matrix_pair) == func_failed)
            return func_failed;

        /*
        * We've modified the polyhedron,
        * so set *face_was_pared to TRUE.
        */
        *face_was_pared = TRUE;

        /*
        * intersect_with_halfspaces() encountered no topological
        * problems due to roundoff errors, so return func_OK.
        */
        return func_OK;
    }

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

/*
* Beta didn't cut off any vertices, so set *face_was_pared to FALSE.
*/
*face_was_pared = FALSE;

/*
* We didn't cut the polyhedron, so we could hardly have encountered
* topological problems due to roundoff error. Return func_OK.
*/

```



```

    */
    return func_OK;
}

static void count_cells(
    WEPolyhedron *polyhedron)
{
    WEVertex *vertex;
    WEEdge *edge;
    WEFace *face;

    /*
     * The counts were intialized in new_WEPolyhedron(),
     * but we'll reinitialize them here just for good form.
     */

    polyhedron->num_vertices = 0;
    polyhedron->num_edges = 0;
    polyhedron->num_faces = 0;

    /*
     * Count the vertices.
     */

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

        polyhedron->num_vertices++;

    /*
     * Count the edges.
     */

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

        polyhedron->num_edges++;

    /*
     * Count the faces.
     */

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

        polyhedron->num_faces++;

    /*
     * Check the Euler characteristic.
     */

    if (polyhedron->num_vertices - polyhedron->num_edges + polyhedron->num_faces != 2)
        uFatalError("count_cells", "Dirichlet_construction");
}

static void sort_faces(
    WEPolyhedron *polyhedron)
{
    /*
     * Sort the faces by order of increasing distance from the basepoint.
     *
     * Note: sort_faces() assumes polyhedron->num_faces is correct, which
     * is true in the present context because compute_Dirichlet_domain()
     * calls count_cells() before sort_faces().
     */

    WEFace **array,
            *face;
    int i;

```

```

/*
 * This code assumes the polyhedron has at least two WEFaces.
 * But as long as we're doing an error check, let's insist that
 * the polyhedron have at least four faces.
 */
if (polyhedron->num_faces < 4)
    uFatalError("sort_faces", "Dirichlet_construction");

/*
 * Allocate an array to hold the addresses of the WEFaces.
 */
array = NEW_ARRAY(polyhedron->num_faces, WEFace *);

/*
 * Copy the addresses into the array.
 */
for (face = polyhedron->face_list_begin.next,
     i = 0;
     face != &polyhedron->face_list_end;
     face = face->next,
     i++)

    array[i] = face;

/*
 * Do a quick error check to make sure we copied
 * the right number of elements.
 */
if (i != polyhedron->num_faces)
    uFatalError("sort_faces", "Dirichlet_construction");

/*
 * Sort the array of pointers.
 */
qsort( array,
        polyhedron->num_faces,
        sizeof(WEFace *),
        compare_face_distance);

/*
 * Adjust the WEFaces' prev and next fields to reflect the new ordering.
 */

polyhedron->face_list_begin.next = array[0];
array[0]->prev = &polyhedron->face_list_begin;
array[0]->next = array[1];

for (i = 1; i < polyhedron->num_faces - 1; i++)
{
    array[i]->prev = array[i-1];
    array[i]->next = array[i+1];
}

array[polyhedron->num_faces - 1]->prev = array[polyhedron->num_faces - 2];
array[polyhedron->num_faces - 1]->next = &polyhedron->face_list_end;
polyhedron->face_list_end.prev = array[polyhedron->num_faces - 1];

/*
 * Free the array.
 */
my_free(array);
}

static int CDECL compare_face_distance(
    const void *ptr1,
    const void *ptr2)
{
    double diff;

    diff = (*( (WEFace **)ptr1 ))->group_element[0][0]
        - (*( (WEFace **)ptr2 ))->group_element[0][0];

```

```

    if (diff < 0.0)
        return -1;
    if (diff > 0.0)
        return +1;
    return 0;
}

static Boolean verify_faces(
    WEPolyhedron *polyhedron)
{
    WEEdge *edge;
    WEFace *face;

    /*
     * Initialize each face->num_sides to 0.
     */

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

        face->num_sides = 0;

    /*
     * Add the contribution of each edge to the adjacent face->num_sides.
     */

    for (edge = polyhedron->edge_list_begin.next;
         edge != &polyhedron->edge_list_end;
         edge = edge->next)
    {
        edge->f[left ]->num_sides++;
        edge->f[right]->num_sides++;
    }

    /*
     * Check that each face and its mate have the same num_sides.
     */

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

        if (face->num_sides != face->mate->num_sides)

            return func_failed;

    return func_OK;
}

static FuncResult verify_group(
    WEPolyhedron *polyhedron,
    MatrixPairList *gen_list)
{
    /*
     * Check that the face pairing isometries generate all the generators
     * on the original gen_list, to be sure that we have a Dirichlet domain
     * for the manifold/orbifold itself and not some finite-sheeted cover.
     * This check should proceed quickly, because
     *
     * (1) The gen_list was simplified in the function
     *     Dirichlet_from_generators_with_displacement() in Dirichlet.c, so
     *     typically the elements on gen_list will all be face pairing
     *     isometries to begin with, or close to it.
     *
     * (2) compute_Dirichlet_domain() has already called sort_faces() to
     *     sort the polyhedron's faces in order of increasing basepoint
     *     translation distance, so the face pairings we're most likely to
     *     need will be encountered near the beginning of the list.
     */

    MatrixPair *matrix_pair;

```

```

    O3lMatrix    m,
                candidate;
    Boolean      progress;
    WEFace       *face;
    double       verify_epsilon;

    for (matrix_pair = gen_list->begin.next;
        matrix_pair != &gen_list->end;
        matrix_pair = matrix_pair->next)
    {
        o3l_copy(m, matrix_pair->m[0]);

        verify_epsilon = VERIFY_EPSILON;

        while (o3l_equal(m, O3l_identity, MATRIX_EPSILON) == FALSE)
        {
            progress = FALSE;

            for (face = polyhedron->face_list_begin.next;
                face != &polyhedron->face_list_end;
                face = face->next)
            {
                o3l_product(m, *face->group_element, candidate);

                if (m[0][0] - candidate[0][0] > verify_epsilon)
                {
                    o3l_copy(m, candidate);
                    progress = TRUE;
                    break;
                }
            }

            if (progress == FALSE)
            {
                /*
                 * There are two possibilities, either
                 *
                 * (1) We have a Dirichlet domain for a finite-sheeted cover
                 *     of the manifold/orbifold, or
                 *
                 * (2) We have an orbifold -- perhaps one whose basepoint
                 *     is only slightly displaced from a fixed point --
                 *     with nontrivial group elements which move the basepoint
                 *     only a small distance.
                 *
                 * To decide which case we're in and respond appropriately,
                 * we set verify_epsilon to 0.0 and keep going. If we still
                 * can't make progress, then we know we are in case (1).
                 */

                if (verify_epsilon > 0.0)
                {
                    verify_epsilon = 0.0;

                    else
                    {
                        uAcknowledge("Please tell Jeff Weeks that SnapPea seems to have
computed a Dirichlet domain for a finite-sheeted cover of the manifold/orbifold.");
                        return func_failed;
                    }
                }
            }
        }

        return func_OK;
    }

static void rewrite_gen_list(
    WEPolyhedron *polyhedron,
    MatrixPairList *gen_list)
{
    WEFace       *face,
                *mate;

```

```

MatrixPair  *new_matrix_pair;

/*
 * First discard the gen_list's present contents.
 */
free_matrix_pairs(gen_list);

/*
 * Add the identity.
 */
new_matrix_pair = NEW_STRUCT(MatrixPair);
o3l_copy(new_matrix_pair->m[0], O3l_identity);
o3l_copy(new_matrix_pair->m[1], O3l_identity);
new_matrix_pair->height = 1.0;
INSERT_BEFORE(new_matrix_pair, &gen_list->end);

/*
 * Use the face->copied fields to avoid copying both a face and its
 * mate as separate MatrixPairs. First initialize all face->copied
 * fields to FALSE.
 */
for (face = polyhedron->face_list_begin.next;
     face != &polyhedron->face_list_end;
     face = face->next)

    face->copied = FALSE;

/*
 * Go down the list of faces, and for each face which hasn't already
 * been done, copy it's and it's mate's group_elements to a MatrixPair,
 * and append the MatrixPair to the gen_list.
 *
 * Note that the gen_list will be presorted,
 * because the list of faces has been sorted.
 */
for (face = polyhedron->face_list_begin.next;
     face != &polyhedron->face_list_end;
     face = face->next)

    if (face->copied == FALSE)
    {
        mate = face->mate;

        new_matrix_pair = NEW_STRUCT(MatrixPair);
        o3l_copy(new_matrix_pair->m[0], *face->group_element);
        o3l_copy(new_matrix_pair->m[1], *mate->group_element);
        new_matrix_pair->height = (*face->group_element)[0][0];
        INSERT_BEFORE(new_matrix_pair, &gen_list->end);

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

```