

```

/*
 * Dirichlet.c
 *
 * This file provides the functions
 *
 * WEPolyhedron *Dirichlet( Triangulation *manifold,
 *                          double vertex_epsilon,
 *                          Boolean centroid_at_origin,
 *                          DirichletInteractivity interactivity,
 *                          Boolean maximize_injectivity_radius);
 *
 * WEPolyhedron *Dirichlet_from_generators(
 *                          O3lMatrix generators[],
 *                          int num_generators,
 *                          double vertex_epsilon,
 *                          DirichletInteractivity interactivity,
 *                          Boolean maximize_injectivity_radius);
 *
 * void change_basepoint(
 *                          WEPolyhedron **polyhedron,
 *                          Triangulation *manifold,
 *                          O3lMatrix *generators,
 *                          int num_generators,
 *                          double displacement[3],
 *                          double vertex_epsilon,
 *                          Boolean centroid_at_origin,
 *                          DirichletInteractivity interactivity,
 *                          Boolean maximize_injectivity_radius);
 *
 * void free_Dirichlet_domain(WEPolyhedron *polyhedron);
 *
 * Dirichlet() computes a Dirichlet domain for the given manifold.
 * The Dirichlet domain will be centered at a local maximum of the
 * injectivity radius function, so as to bring out the manifold's
 * symmetry. The Dirichlet domain will be for the current Dehn
 * filled solution; the Dehn filling coefficients of filled cusps
 * must be integers, but they need not be relatively prime. That is,
 * the code works for orbifolds as well as manifolds. Throughout
 * this documentation when I say "manifold" I really mean "manifold
 * or orbifold". If the Dehn filling coefficients are all integers,
 * Dirichlet() computes the Dirichlet domain in a winged edge data
 * structure (cf. winged_edge.h) and returns a pointer to it. If the
 * Dehn filling coefficients aren't all integers, or if the algorithm
 * fails, Dirichlet() returns NULL. The documentation at the top of
 * the file Dirichlet_construction.c explains why any Dirichlet domain
 * algorithm must fail for sufficiently difficult manifolds, and how an
 * appropriate choice of vertex_epsilon minimizes the chances of failure.
 * If DirichletInteractivity == Dirichlet_interactive, the algorithm
 * queries the user for how to proceed when the basepoint happens to be
 * at a critical point; otherwise it either computes the Dirichlet
 * domain based at that point or moves on, according to whether
 * DirichletInteractivity is Dirichlet_stop_here or Dirichlet_keep_going.
 *
 * Dirichlet_from_generators() computes a Dirichlet domain directly
 * from a set of generators. Dirichlet() (see code below) does nothing
 * but compute a set of generators for the manifold and pass them to
 * Dirichlet_from_generators(). Dirichlet_from_generators() is externally
 * available so the UI can compute a Dirichlet domain from an explicit list
 * of matrix generators. Such an approach is essential for orbifolds whose
 * singular sets are more complicated than just a collection of circles.
 *
 * change_basepoint() reads the face pairing matrices from the polyhedron
 * (if *polyhedron != NULL), shifts the basepoint by the given displacement,
 * lets the basepoint move to a local maximum of the injectivity radius
 * function, and recomputes the Dirichlet domain.
 * If *polyhedron is NULL, it computes the Dirichlet domain directly from
 * the given manifold or generators, but with the given displacement of
 * the initial basepoint. In either case, a pointer to the resulting
 * Dirichlet domain (or NULL if an error occurs as described in Dirichlet()
 * above) is written to *polyhedron.
 *
 * free_Dirichlet_domain() frees the storage occupied by a WEPolyhedron.
 */

```

```

#include "kernel.h"

/*
 * 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.
 */

#include "Dirichlet.h"

/*
 * simplify_generators() considers one MatrixPair to be simpler than
 * another if its height is at least SIMPLIFY_EPSILON less.
 */
#define SIMPLIFY_EPSILON 1e-2

/*
 * Two vectors are considered linearly dependent iff the length of their
 * cross product is less than LENGTH_EPSILON. LENGTH_EPSILON can be fairly
 * large, because (1) the vectors involved are normals to faces and are
 * unlikely to be almost but not quite linearly dependent, and (2) if we
 * don't like one face normal, we'll move on to the next one.
 */
#define LENGTH_EPSILON 1e-2

/*
 * An O(3,1) matrix is considered to fix the basepoint (1, 0, 0, 0) iff its
 * (0,0)th entry is less than 1.0 + FIXED_BASEPOINT_EPSILON. When choosing
 * FIXED_BASEPOINT_EPSILON, recall that a point d units from the basepoint
 * lies at a height cosh(d), which for small d is approximately 1 + (1/2)d^2.
 * So, for example, to detect points within a distance of about 1e-3 of
 * the basepoint, you should set FIXED_BASEPOINT_EPSILON to 1e-6.
 */
#define FIXED_BASEPOINT_EPSILON 1e-6

static void array_to_matrix_pair_list(O3lMatrix generators[], int num_generators,
    MatrixPairList *gen_list);
static Boolean is_matrix_on_list(O3lMatrix m, MatrixPairList *gen_list);
static void insert_matrix_on_list(O3lMatrix m, MatrixPairList *gen_list);
static void simplify_generators(MatrixPairList *gen_list);
static Boolean generator_fixes_basepoint(MatrixPairList *gen_list);
static double product_height(O3lMatrix a, O3lMatrix b);
static void generators_from_polyhedron(WEPolyhedron *polyhedron, O3lMatrix **
    generators, int *num_generators);

WEPolyhedron *Dirichlet(
    Triangulation *manifold,
    double vertex_epsilon,
    Boolean centroid_at_origin,
    DirichletInteractivity interactivity,
    Boolean maximize_injectivity_radius)
{
    double null_displacement[3] = {0.0, 0.0, 0.0};

    return Dirichlet_with_displacement( manifold,
                                        null_displacement,
                                        vertex_epsilon,
                                        centroid_at_origin,
                                        interactivity,
                                        maximize_injectivity_radius);
}

WEPolyhedron *Dirichlet_from_generators(
    O3lMatrix generators[],
    int num_generators,
    double vertex_epsilon,
    DirichletInteractivity interactivity,
    Boolean maximize_injectivity_radius)
{
    double null_displacement[3] = {0.0, 0.0, 0.0};

```

```

    return Dirichlet_from_generators_with_displacement(
        generators,
        num_generators,
        null_displacement,
        vertex_epsilon,
        interactivity,
        maximize_injectivity_radius);
}

WEPolyhedron *Dirichlet_with_displacement(
    Triangulation      *manifold,
    double             displacement[3],
    double             vertex_epsilon,
    Boolean            centroid_at_origin,
    DirichletInteractivity interactivity,
    Boolean            maximize_injectivity_radius)
{
    MoebiusTransformation *Moebius_generators;
    O3lMatrix              *o3l_generators;
    WEPolyhedron           *polyhedron;

    /*
     * Make sure we have a hyperbolic manifold.
     */
    if (get_filled_solution_type(manifold) != geometric_solution
        && get_filled_solution_type(manifold) != nongeometric_solution)
        return NULL;

    /*
     * Make sure all the Dehn filling coefficients are integers.
     * This will ensure that we are working with a manifold or
     * orbifold, and the group is discrete.
     */
    if ( ! all_Dehn_coefficients_are_integers(manifold) )
        return NULL;

    /*
     * Compute a set of generators, and convert them from
     * MoebiusTransformations to O3lMatrices.
     */
    choose_generators(manifold, FALSE, FALSE); /* counts the generators so we can allocate
    the arrays */
    Moebius_generators = NEW_ARRAY(manifold->num_generators, MoebiusTransformation);
    o3l_generators      = NEW_ARRAY(manifold->num_generators, O3lMatrix);
    matrix_generators(manifold, Moebius_generators, centroid_at_origin);
    Moebius_array_to_O3l_array(Moebius_generators, o3l_generators, manifold->
    num_generators);

    /*
     * Compute the Dirichlet domain.
     */
    polyhedron = Dirichlet_from_generators_with_displacement(
        o3l_generators,
        manifold->num_generators,
        displacement,
        vertex_epsilon,
        interactivity,
        maximize_injectivity_radius);

    /*
     * Free the generators.
     */
    my_free(Moebius_generators);
    my_free(o3l_generators);

    /*
     * Return a pointer to the Dirichlet domain.
     */
    return polyhedron;
}

WEPolyhedron *Dirichlet_from_generators_with_displacement(

```

```

    O3lMatrix      generators[],
    int            num_generators,
    double         displacement[3],
    double         vertex_epsilon,
    DirichletInteractivity interactivity,
    Boolean        maximize_injectivity_radius)
{
    MatrixPairList gen_list;
    WEPolyhedron   *polyhedron;
    Boolean        basepoint_moved;
    double         small_displacement[3] = {0.01734, 0.02035, 0.00721};

    /*
     * Convert the array of generators to a MatrixPairList.
     */
    array_to_matrix_pair_list(generators, num_generators, &gen_list);

    /*
     * Roundoff error tends to accumulate throughout this algorithm.
     * In general we can't eliminate roundoff error, but in many of
     * the nicest examples (e.g. those coming from triangulations with
     * 60-60-60 or 45-45-90 ideal tetrahedra) the matrix entries are
     * quarter integer multiples of 1, sqrt(2), and sqrt(3). In these
     * cases, if we can recognize a matrix entry to be a nice number to
     * fairly good precision, we can set it equal to that number to full
     * precision. If we do so after each matrix multiplication, the
     * roundoff error will stay under control. Please see
     * Dirichlet_precision.c for details.
     */
    precise_generators(&gen_list);

    /*
     * Displace the basepoint as required.
     */
    conjugate_matrices(&gen_list, displacement);

    /*
     * There is a danger that when initial_polyhedron() in
     * Dirichlet_construction.c slices its cube with the elements of
     * gen_list, the resulting set of face->group_elements will not suffice
     * to generate the full group. So we simplify the generators ahead of
     * time to try to avoid this problem. Please see simplify_generators()
     * for more details.
     */
    simplify_generators(&gen_list);

    /*
     * If the singular set of an orbifold passes through the initial
     * basepoint, we won't be able to compute the Dirichlet domain.
     * So if any element of gen_list fixes the basepoint, we must give
     * the basepoint a small arbitrary displacement. Thereafter
     * the algorithm for maximizing the injectivity radius will
     * automatically keep the basepoint away from the singular set.
     * (Note that this approach is not foolproof. It's possible -- but
     * I hope unlikely -- that some product of the generators will fix
     * the basepoint even though no generator alone does. If this happens,
     * compute_Dirichlet_domain() will fail. My hope is that the
     * preceding call to simplify_generators() will find any elements
     * fixing the basepoint, if they aren't already explicitly included
     * in the gen_list.)
     */
    if (generator_fixes_basepoint(&gen_list) == TRUE)
        conjugate_matrices(&gen_list, small_displacement);

    while (TRUE)
    {
        /*
         * Compute the Dirichlet domain relative to the current basepoint.
         * compute_Dirichlet_domain() modifies gen_list to correspond
         * to the face pairings of the Dirichlet domain. The generators
         * are giving in order of increasing image height (see
         * Dirichlet_basepoint.c for the definition of "image height").
         */
        polyhedron = compute_Dirichlet_domain(&gen_list, vertex_epsilon);
    }
}

```

```

/*
 * If we ran into problems with loss of numerical accuracy,
 * return NULL.
 */
if (polyhedron == NULL)
{
    free_matrix_pairs(&gen_list);
    return NULL;
}

/*
 * If necessary, move the basepoint to a local maximum of
 * the injectivity radius function. Set the flag
 * basepoint_moved to indicate whether a change of basepoint
 * was both requested and necessary.
 */

if (maximize_injectivity_radius == TRUE)
    /* Move the basepoint to a local maximum */
    /* of the injectivity radius function. */
    maximize_the_injectivity_radius(&gen_list, &basepoint_moved, interactivity);
else
    /* Leave the basepoint where it is. */
    basepoint_moved = FALSE;

/*
 * If the basepoint was already at a local maximum of the
 * injectivity radius, clean up and go home.
 */
if (basepoint_moved == FALSE)
{
    free_matrix_pairs(&gen_list);
    if (Dirichlet_bells_and_whistles(polyhedron) == func_OK)
        return polyhedron;
    else
    {
        free_Dirichlet_domain(polyhedron);
        return NULL;
    }
}

/*
 * We're not at a local maximum, so discard the Dirichlet domain
 * and repeat the loop.
 */
free_Dirichlet_domain(polyhedron);
}

/*
 * The program will never reach this point.
 */
}

static void array_to_matrix_pair_list(
    O31Matrix    generators[],
    int          num_generators,
    MatrixPairList *gen_list)
{
    int          i;

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

    /*
     * We make no assumption about whether the array "generators"
     * includes the identity, but we do want to insure that the
     * gen_list does, so we insert the identity matrix now.
     */

```

```

    */
    insert_matrix_on_list(o3l_identity, gen_list);

    /*
    * Add the matrices from the array "generators" to the
    * MatrixPairList. We make no assumptions about whether
    * inverses are or are not present in the array "generators".
    */
    for (i = 0; i < num_generators; i++)
        if (is_matrix_on_list(generators[i], gen_list) == FALSE)
            insert_matrix_on_list(generators[i], gen_list);
}

static Boolean is_matrix_on_list(
    O3lMatrix      m,
    MatrixPairList *gen_list)
{
    MatrixPair *matrix_pair;
    int        i;

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

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

            if (o3l_equal(m, matrix_pair->m[i], MATRIX_EPSILON))

                return TRUE;

    return FALSE;
}

static void insert_matrix_on_list(
    O3lMatrix      m,
    MatrixPairList *gen_list)
{
    O3lMatrix      m_inverse;
    MatrixPair      *new_matrix_pair,
                    *mp;

    o3l_invert(m, m_inverse);

    new_matrix_pair = NEW_STRUCT(MatrixPair);
    o3l_copy(new_matrix_pair->m[0], m);
    o3l_copy(new_matrix_pair->m[1], m_inverse);
    new_matrix_pair->height = m[0][0];

    /*
    * Find the MatrixPair mp immediately following the point where
    * we want to insert the new_matrix_pair into the gen_list.
    */
    mp = gen_list->begin.next;
    while (mp != &gen_list->end && mp->height < new_matrix_pair->height)
        mp = mp->next;

    INSERT_BEFORE(new_matrix_pair, mp);
}

void free_matrix_pairs(
    MatrixPairList *gen_list)
{
    MatrixPair *dead_node;

    while (gen_list->begin.next != &gen_list->end)
    {
        dead_node = gen_list->begin.next;
        REMOVE_NODE(dead_node);
        my_free(dead_node);
    }
}

```

```

void free_Dirichlet_domain(
    WEPolyhedron    *polyhedron)
{
    WEVertex        *dead_vertex;
    WEEdge          *dead_edge;
    WEFace          *dead_face;
    WEVertexClass   *dead_vertex_class;
    WEEdgeClass     *dead_edge_class;
    WEFaceClass     *dead_face_class;

    if (polyhedron == NULL)
        uFatalError("free_Dirichlet_domain", "Dirichlet");

    while (polyhedron->vertex_list_begin.next != &polyhedron->vertex_list_end)
    {
        dead_vertex = polyhedron->vertex_list_begin.next;
        REMOVE_NODE(dead_vertex);
        my_free(dead_vertex);
    }

    while (polyhedron->edge_list_begin.next != &polyhedron->edge_list_end)
    {
        dead_edge = polyhedron->edge_list_begin.next;
        REMOVE_NODE(dead_edge);
        my_free(dead_edge);
    }

    while (polyhedron->face_list_begin.next != &polyhedron->face_list_end)
    {
        dead_face = polyhedron->face_list_begin.next;
        REMOVE_NODE(dead_face);
        if (dead_face->group_element != NULL)
            my_free(dead_face->group_element);
        my_free(dead_face);
    }

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

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

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

    my_free(polyhedron);
}

static void simplify_generators(
    MatrixPairList *gen_list)
{
    /*
     * There is a danger that when we slice the cube with elements of
     * gen_list, the resulting set of face->group_elements will not suffice
     * to generate the full group. To see this problem more clearly, forget
     * hyperbolic manifolds for a moment and consider the canonical  $Z + Z$ 
     * action on the Euclidean plane. The final Dirichlet domain should be
     * the square  $\text{Max}(|x|, |y|) \leq 1/2$ . Say we propose to compute it by
     * starting with the square  $\text{Max}(|x|, |y|) \leq 3$  and slicing it with the
    */
}

```

```

* halfplanes defined by the generators alpha: (x,y) -> (x+2, y+1) and
* beta: (x,y) -> (x+5, x+3). Once we've sliced the cube by alpha,
* there'll be nothing left for beta to slice off, and beta will be
* lost. To avoid this problem, we simplify our initial set of
* generators. Whenever the product alpha*beta of two generators
* moves the basepoint less than one of the factors (alpha or beta),
* we replace that factor with the product. The procedure is similar
* in spirit to simplifying the above generators of  $Z + Z$  by repeatedly
* subtracting one from the other:  $\{(2,1), (5,3)\} \rightarrow \{(2,1), (1,1)\}$ 
*  $\rightarrow \{(1,0), (1,1)\} \rightarrow \{(1,0), (0,1)\}$ .
*
* Note, by the way, that the hyperbolic case is even worse than the
* Euclidean case. In the Euclidean case, the halfplane corresponding
* to one primitive generator cannot be a subset of the halfplane
* corresponding to another, but in the hyperbolic case it can.
*/

MatrixPair *aA,
           *bB,
           *best_aA;
O3lMatrix *best_aA_factor,
          *best_bB_factor;
double max_improvement,
       improvement;
int i,
    j;

/*
* Definition: We'll say that one MatrixPair is "greater than"
* another iff the value of its height field is greater than that
* of the other.
*/

/*
* Roughly speaking, the idea is to look for MatrixPairs {a, A} and
* {b, B} whose product is less than {a, A}, and replace {a, A} with
* the product. (Technically speaking, there are four different ways
* to form the product, as explained in the code below.) Unfortunately,
* the roundoff error in the product is roughly the sum of the
* roundoff errors in {a, A} and {b, B}, so we want to minimize the
* number of matrix multiplications we do. For this reason we look
* for the {a, A} and {b, B} whose product is less than {a, A} by the
* greatest amount possible (this will be the max_improvement
* variable below).
*
* We keep replacing generators by products until no further
* improvement is possible. Whenever the identity matrix occurs,
* we remove it.
*/

/*
* We'll break out of the while() loop when no further
* progress is possible.
*/

while (TRUE)
{
    /*
    * Find the MatrixPairs {a, A} and {b, B} which offer the
    * greatest possible height decrease.
    */

    max_improvement = 0.0;

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

        for (bB = gen_list->begin.next;
             bB != &gen_list->end;
             bB = bB->next)
        {
            /*
            * We certainly don't want to replace a MatrixPair

```



```

        * with its square, so skip the case {a, A} == {b, B}.
    */
    if (aA == bB)
        continue;

    /*
    * We want aA to be the larger of the two. If bB is larger,
    * skip it, knowing that {a, A} and {b, B} will eventually
    * show up in the opposite order, if they haven't already.
    * (If they happen to be exactly equal, we consider them.
    * Better to consider a pair twice than not at all!)
    */
    if (aA->height < bB->height)
        continue;

    /*
    * There are four possible products to consider:
    * {ab, BA}, {aB, bA}, {Ab, Ba} and {AB, ba}.
    * For computational efficiency we check the height of each
    * possibility without doing the full matrix multiplication.
    */
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
        {
            improvement = aA->height - product_height(aA->m[i], bB->m[j]);
            if (improvement > max_improvement)
            {
                max_improvement = improvement;
                best_aA = aA;
                /* Cater to a DEC compiler error that chokes on &(array)[i] */
                /* best_aA_factor = &aA->m[i]; */
                /* best_bB_factor = &bB->m[j]; */
                /* best_aA_factor = aA->m + i; */
                /* best_bB_factor = bB->m + j; */
            }
        }

    /*
    * If no improvement is possible, we're done.
    */

    if (max_improvement < SIMPLIFY_EPSILON)
        break;

    /*
    * Replace the contents of best_aA with the appropriate product.
    */

    precise_o3l_product(*best_aA_factor, *best_bB_factor, best_aA->m[0]);
    o3l_invert(best_aA->m[0], best_aA->m[1]);
    best_aA->height = best_aA->m[0][0][0];

    /*
    * If best_aA is the identity, remove it.
    */

    if (o3l_equal(best_aA->m[0], O3l_identity, MATRIX_EPSILON) == TRUE)
    {
        REMOVE_NODE(best_aA);
        my_free(best_aA);
    }
}

static Boolean generator_fixes_basepoint(
    MatrixPairList *gen_list)
{
    MatrixPair *matrix_pair;

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

```

```

        if (matrix_pair->m[0][0][0] < 1.0 + FIXED_BASEPOINT_EPSILON)

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

                return TRUE;

    return FALSE;
}

static double product_height(
    O3lMatrix  a,
    O3lMatrix  b)
{
    /*
     * We don't need the whole product of a and b, just the [0][0] entry.
     */

    double  height;
    int     i;

    height = 0.0;

    for (i = 0; i < 4; i++)
        height += a[0][i] * b[i][0];

    return height;
}

void change_basepoint(
    WEPolyhedron      **polyhedron,
    Triangulation      *manifold,
    O3lMatrix          *generators,
    int                num_generators,
    double             displacement[3],
    double             vertex_epsilon,
    Boolean            centroid_at_origin,
    DirichletInteractivity  interactivity,
    Boolean            maximize_injectivity_radius)
{
    O3lMatrix  *gen;
    int        num_gen;

    /*
     * If polyhedron is not NULL, the plan is to read the generators
     * from the polyhedron, displace them, and recompute the
     * Dirichlet domain.
     *
     * If *polyhedron is NULL (as would be the case if
     * compute_Dirichlet_domain() failed with the previous basepoint),
     * then we must compute the generators directly from the
     * Triangulation if one is present, or otherwise from the
     * explicitly provided generators.
     */

    if (*polyhedron != NULL)
    {
        generators_from_polyhedron(*polyhedron, &gen, &num_gen);

        free_Dirichlet_domain(*polyhedron);

        (*polyhedron) = Dirichlet_from_generators_with_displacement(
            gen, num_gen, displacement, vertex_epsilon, interactivity,
            maximize_injectivity_radius);

        my_free(gen);
    }
    else
    {
        if (manifold != NULL)
            (*polyhedron) = Dirichlet_with_displacement(
                manifold, displacement, vertex_epsilon, centroid_at_origin,

```

```
        interactivity, maximize_injectivity_radius);
    else if (generators != NULL && num_generators > 0)
        (*polyhedron) = Dirichlet_from_generators_with_displacement(
            generators, num_generators, displacement, vertex_epsilon,
            interactivity, maximize_injectivity_radius);
    else
        uFatalError("change_basepoint", "Dirichlet");
}

static void generators_from_polyhedron(
    WEPolyhedron *polyhedron,
    O3lMatrix **generators,
    int *num_generators)
{
    WEFace *face;
    int i;

    *num_generators = polyhedron->num_faces;

    *generators = NEW_ARRAY(*num_generators, O3lMatrix);

    for (face = polyhedron->face_list_begin.next,
        i = 0;
        face != &polyhedron->face_list_end;
        face = face->next,
        i++)

        o3l_copy((*generators)[i], *face->group_element);

    if (i != *num_generators)
        uFatalError("generators_from_polyhedron", "Dirichlet");
}
```