

```

/*
 * matrix_generators.c
 *
 * This file provides the function
 *
 * void matrix_generators( Triangulation      *manifold,
 *                        MoebiusTransformation generators[],
 *                        Boolean              centroid_at_origin);
 *
 * which computes the MoebiusTransformations representing the action
 * of the generators of a manifold's fundamental group on the sphere at
 * infinity. matrix_generators() writes the MoebiusTransformations
 * to the array generators[], which it assumes has already been allocated.
 */

#include "kernel.h"

static void compute_one_generator(Tetrahedron *tet, FaceIndex f, MoebiusTransformation *mt)
{

void matrix_generators(
    Triangulation      *manifold,
    MoebiusTransformation generators[],
    Boolean              centroid_at_origin)
{
    Boolean      *already_computed;
    int          i;
    FaceIndex    f;
    Tetrahedron  *tet;

    /*
     * Compute the locations of the ideal vertices of the Tetrahedra
     * on the sphere at infinity.
     */

    choose_generators(manifold, TRUE, centroid_at_origin);

    /*
     * Keep track of which generators we've already computed,
     * to avoid unnecessary duplication of effort.
     */

    already_computed = NEW_ARRAY(manifold->num_generators, Boolean);
    for (i = 0; i < manifold->num_generators; i++)
        already_computed[i] = FALSE;

    /*
     * Search through all the faces of all the Tetrahedra looking
     * for generators. Compute those not already computed.
     */

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

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

            if (tet->generator_status[f] == outbound_generator
                && already_computed[tet->generator_index[f]] == FALSE)
            {
                compute_one_generator(tet, f, &generators[tet->generator_index[f]]);
                already_computed[tet->generator_index[f]] = TRUE;
            }

    /*
     * We're done.
     * Free the array and go home.
     */

    my_free(already_computed);
}

```

```

static void compute_one_generator(
    Tetrahedron      *tet,
    FaceIndex        f,
    MoebiusTransformation *mt)
{
    int      i,
            count;
    Complex a[3],
            b[3],
            k,
            blk,
            normalization;

    /*
     * First read out the locations of the corners of this face, and
     * also its mate elsewhere on the boundary of the fundamental domain.
     *
     * There are two possible interpretations for the matrix corresponding
     * to a given generator.
     *
     * (1) Think of the matrix as defining a face pairing isometry on
     * a fundamental domain. The isometry takes the given face to
     * its mate.
     *
     * (2) Think of the matrix as giving a representation of the abstract
     * fundamental group into the group of covering transformations.
     * The isometry takes the mate to the given face.
     *
     * Here we adopt interpretation (2). If you preferred (1) you'd need
     * to switch the definitions of a[] and b[] to
     *
     *     a[count] = tet->corner[i];
     *     b[count] = tet->neighbor[f]->corner[EVALUATE(tet->gluing[f], i)];
     *
     * (But don't switch them here! You'll foul up the matrix representations
     * in fundamental_group.c.)
     */

    count = 0;

    for (i = 0; i < 4; i++)
    {
        if (i == f)
            continue;

        a[count] = tet->neighbor[f]->corner[EVALUATE(tet->gluing[f], i)];
        b[count] = tet->corner[i];

        count++;
    }

    /*
     * Note the parity of MoebiusTransformation.
     */

    mt->parity = tet->generator_parity[f];

    /*
     * If the MoebiusTransformation is orientation_reversing, we want
     * to compute a function of z-bar, as explained in the documentation
     * accompanying the definition of a MoebiusTransformation in SnapPea.h.
     */

    if (mt->parity == orientation_reversing)
        for (i = 0; i < 3; i++)
            a[i] = complex_conjugate(a[i]);

    /*
     * The formula for the Moebius transformation taking the a[] to the b[]
     * is simple enough:
     *
     * 
$$f(z) = \frac{(b_1 k - b_0) * z + (b_0 a_1 - b_1 a_0 k)}{(k - 1) * z + (a_1 - k a_0)}$$

     */

```

```

*   where
*
*       k = [(b2-b0)/(b2-b1)] * [(a2-a1)/(a2-a0)]
*
*   Even though one of the a[] and/or one of the b[] could be infinite,
*   we will bravely push forward with the computation. The justification
*   for such boldness is that the Complex constant Infinity is not actually
*   infinite, but is just a large number (1e34). Its square is less than
*   the assumed value of DBL_MAX, so we can safely compute squares
*   of "infinite" numbers and expect them to cancel properly with other
*   infinite numbers. In the normalization step we also assume the fourth
*   power of "infinity" is less than DBL_MAX. (DBL_MAX is 1.2e+4932
*   on a Mac and 1.8e+308 on a Sun or NeXT, so we shouldn't run into
*   trouble (knock on wood).)
*
*   It would be more rigorous to consider separately the cases with
*   a0, a1 or a2; and/or b0, b1 or b2 are infinite, and take the limit of
*   the above formula by hand, but I didn't feel up to it.
*/

k = complex_div(
    complex_mult(complex_minus(b[2],b[0]), complex_minus(a[2],a[1])),
    complex_mult(complex_minus(b[2],b[1]), complex_minus(a[2],a[0]))
);
blk = complex_mult(b[1], k);
normalization = complex_sqrt(
    complex_div(
        One,
        complex_mult(k,
            complex_mult(
                complex_minus(a[1],a[0]),
                complex_minus(b[1],b[0])
            )
        )
    )
);

mt->matrix[0][0] = complex_mult(
    normalization,
    complex_minus(blk, b[0])
);
mt->matrix[0][1] = complex_mult(
    normalization,
    complex_minus(
        complex_mult(b[0], a[1]),
        complex_mult(blk, a[0])
    )
);
mt->matrix[1][0] = complex_mult(
    normalization,
    complex_minus(k, One)
);
mt->matrix[1][1] = complex_mult(
    normalization,
    complex_minus(
        a[1],
        complex_mult(k,a[0])
    )
);
}

```