

```

/*
 * matrix_conversion.c
 *
 * This file provides the two functions
 *
 *     void Moebius_to_O3l(MoebiusTransformation *A, O3lMatrix B);
 *     void O3l_to_Moebius(O3lMatrix B, MoebiusTransformation *A);
 *
 * which convert matrices back and forth between SL(2,C) and O(3,1),
 * as well as the functions
 *
 *     void Moebius_array_to_O3l_array(MoebiusTransformation  arrayA[],
 *                                     O3lMatrix                arrayB[],
 *                                     int                        num_matrices);
 *     void O3l_array_to_Moebius_array(O3lMatrix                arrayB[],
 *                                     MoebiusTransformation  arrayA[],
 *                                     int                        num_matrices);
 *
 * which do the same for arrays of matrices.
 *
 * As an add-on, this file also provides
 *
 *     Boolean O3l_determinants_OK(    O3lMatrix  arrayB[],
 *                                     int          num_matrices,
 *                                     double        epsilon);
 *
 * which returns TRUE if all the O3lMatrices in the array have determinants
 * within epsilon of plus or minus one, and FALSE otherwise.
 *
 * The algorithm in Moebius_to_O3l() is based on an explanation provided
 * by Craig Hodgson of a program written by Diane Hoffoss which in turn
 * was based on an algorithm explained to her by Bill Thurston.
 * One would expect a more straightforward algorithm (i.e. something
 * which makes a direct correspondence between the Minkowski space
 * model and the upper half space model), but two problems arise:
 * (1) Whenever you work with points on the sphere at infinity, you
 *     encounter the problem that there is no good way to normalize
 *     the lengths of vectors on the light cone. The most promising
 *     approach to solving this problem is to work with the basis
 *     which is dual to a basis of lightlike vectors, but this is
 *     at best a nuisance.
 * (2) The computations required to pass from one model to the
 *     other appear, superficially, more complicated than the simple
 *     calculations used in Thurston's algorithm, but obviously they
 *     have to simplify down to the same thing eventually.
 * So . . . for now I'll just stick with Thurston's algorithm.
 */

```

```

#include "kernel.h"

```

```

void Moebius_array_to_O3l_array(
    MoebiusTransformation  arrayA[],
    O3lMatrix              arrayB[],
    int                    num_matrices)
{
    int i;

    for (i = 0; i < num_matrices; i++)
        Moebius_to_O3l(&arrayA[i], arrayB[i]);
}

```

```

void O3l_array_to_Moebius_array(
    O3lMatrix              arrayB[],
    MoebiusTransformation  arrayA[],
    int                    num_matrices)
{
    int i;

    for (i = 0; i < num_matrices; i++)
        O3l_to_Moebius(arrayB[i], &arrayA[i]);
}

```

```

void Moebius_to_O31(
    MoebiusTransformation *A,
    O31Matrix B)
{
    /*
    * The trick here is to consider the Minkowski space E(3,1)
    * not just as an abstract space, but as the space of all
    * 2 x 2 Hermitian matrices. Roughly speaking, "Hermitian"
    * is the complex version of "symmetric".
    *
    * Definition. The adjoint M* of a complex matrix is
    * the transpose of the complex conjugate of M.
    *
    * Definition. A complex matrix is Hermitian iff M* = M.
    *
    * A 2 x 2 complex matrix is Hermitian matrix iff it has the form
    *
    *          a + 0i      c + di
    *          c - di      b + 0i
    *
    * Such matrices form a 4-dimensional real vector space V.
    * We choose the following basis for V:
    *
    *      M0 = 1  0      M1 = 1  0      M2 = 0  1      M3 = 0  i
    *           0  1           0 -1          1  0          -i  0
    *
    * The determinant -det() defines a (squared) norm on V.
    * (We use -det() instead of det() so that we end up with
    * a metric of signature (-+++)) instead of (+---).)
    * This norm leads us to an inner product <, > on V.
    * We want the inner product to satisfy
    *
    *      <M+N, M+N> = <M, M> + 2<M, N> + <N, N>
    *
    * which is equivalent to
    *
    *      -det(M+N) = -det(M) + 2<M, N> - det(N),
    *
    * so we define the inner product to be
    *
    *      <M, N> = (1/2)(det(M) + det(N) - det(M+N)).
    *
    * Relative to this inner product, the vectors {M0, M1, M2, M3}
    * are mutually orthogonal and have squared norms {-1, +1, +1, +1}.
    * I.e. this is the usual metric for the Minkowski space E(3,1).
    *
    * Assume for the moment that the MoebiusTransformation we want
    * to convert to O(3,1) represents an orientation_preserving isometry.
    *
    * We will let a matrix A in SL(2,C) act on the Minkowski
    * space V, and compute the matrix B in O(3,1) which describes
    * the action. The action of the matrix A will be denoted f(A).
    * Thus, f(A) is itself a function which acts on elements of V.
    * To define f(A), we must say how it acts on each element M of V:
    *
    *      [f(A)](M) = A M A*
    *
    * It's trivial to see that f(A) is a linear function.
    * f(A) preserves norms [det(A)det(M)det(A*) = det(M),
    * because det(A) = 1], hence it also preserves the inner product;
    * therefore f(A) is an isometry of V.
    *
    * To show that f() is a homomorphism from SL(2,C) to Isom(V),
    * we must show that the isometries f(A) o f(A') and f(AA')
    * are equal (where "o" denotes composition of functions).
    * That is, we must show that [f(A) o f(A')](M) = f(AA')(M)
    * for all M in V.
    *
    *      [f(A) o f(A')](M) = f(A)( f(A')(M) )
    *                          = f(A)( A' M A'* )
    *                          = A A' M A'* A*
    *                          = (AA') M (AA')*
    *                          = f(AA')(M)
    */
}

```

```

*
* It's easy to check that f(A) fixes the basis {M0, M1, M2, M3}
* iff A is plus or minus the identity. That is, the kernel
* of f() is {+-I}, and we may think of f() as an injective
* map from PSL(2,C) into Isom(V).
*
* Exercise for the reader: prove that f() maps PSL(2,C)
* onto the set of isometries of V which preserve the "time
* direction". I.e. the set of all isometries which don't
* interchange the past and future light cones.
*
* So far we have only established the correspondence between
* isometries in the upper half space model (as represented
* by SL(2,C) matrices) and isometries in the Minkowski space
* model (as represented by O(3,1) matrices). We haven't made
* a direct correspondence between the points of the upper half
* space model and the points of the Minkowski space model.
* For some purposes (e.g. dealing with orientation_reversing
* isometries) we need to know the precise correspondence between
* the spheres at infinity in the two models. We deduce the
* correspondence by converting several carefully chosen matrices
* from one model to the other, and comparing their fixed points
* on the sphere at infinity in each model. Each isometry is
* a translation along a geodesic (without rotation).
*
*      Upper Half Space                      Minkowski Space
*
*      matrix      axis                      matrix      axis
*
*      2      0      from 0                  17/8 15/8  0    0      from
*      0      1/2    towards                  15/8 17/8  0    0      (1,-1, 0, 0)
*      infinity                                     0    0    1    0      towards
*      infinity                                     0    0    0    1      (1, 1, 0, 0)
*
*      17/8  0    15/8  0      from
*      5/4   3/4   from -1      0    1    0    0      (1, 0,-1, 0)
*      3/4   5/4   towards 1    15/8  0    17/8  0      towards
*      0    0    0    1      (1, 0, 1, 0)
*
*      17/8  0    0    15/8      from
*      5/4   3i/4   from -i      0    1    0    0      (1, 0, 0,-1)
*      -3i/4  5/4   towards i    0    0    1    0      towards
*      15/8  0    0    17/8      (1, 0, 0, 1)
*
* Comparison of the above fixed points reveals the correspondence
*
*      0      <-> (1,-1, 0, 0)
*      infinity <-> (1, 1, 0, 0)
*      -1      <-> (1, 0,-1, 0)
*      1       <-> (1, 0, 1, 0)
*      -i      <-> (1, 0, 0,-1)
*      i       <-> (1, 0, 0, 1)
*
* As a corollary, we may deduce the matrix in Minkowski space
* which corresponds to complex conjugation in the sphere at
* infinity in the upper half space model. In the upper half
* space model, complex conjugation fixes 0, infinity, -1 and 1,
* and interchanges -i and i. Therefore in the Minkowski space
* model it must fix (1,-1, 0, 0), (1, 1, 0, 0), (1, 0,-1, 0)
* and (1, 0, 1, 0), and interchange (1, 0, 0,-1) and (1, 0, 0, 1).
* The matrix which does this is
*
*      1  0  0  0
*      0  1  0  0
*      0  0  1  0
*      0  0  0 -1
*
* Therefore, to convert an orientation_reversing MoebiusTransformation
* to SO(3,1), we first convert the SL(2,C) matrix (as if the isometry
* were orientation_preserving), and then multiply the resulting O(3,1)
* matrix on the right by the matrix shown above, to account for the
* complex conjugation.
*
* After that long-winded documentation, the code itself is very

```

```

    * simple. The (i,j)-th entry of the O(3,1) matrix B is the
    * i-th component (relative to the basis {m[0], m[1], m[2], m[3]})
    * of A m[j] A*.
    */

SL2CMatrix ad_A, /* A* = adjoint of A */
           fAmj, /* f(A)(m[j]) = A m[j] A* */
           temp;
int i, /* which row of B */
    j; /* which column of B */

CONST static SL2CMatrix m[4] =
{
    {{{ 1.0, 0.0},{ 0.0, 0.0}},
     {{ 0.0, 0.0},{ 1.0, 0.0}}},

    {{{ 1.0, 0.0},{ 0.0, 0.0}},
     {{ 0.0, 0.0},{-1.0, 0.0}}},

    {{{ 0.0, 0.0},{ 1.0, 0.0}},
     {{ 1.0, 0.0},{ 0.0, 0.0}}},

    {{{ 0.0, 0.0},{ 0.0, 1.0}},
     {{ 0.0,-1.0},{ 0.0, 0.0}}}
};

/*
 * First convert A->matrix to SO(3,1), without
 * worrying about A->parity.
 */

/*
 * For each basis vector m[j] . . .
 */

for (j = 0; j < 4; j++)
{
    /*
     * . . . compute f(A)(m[j]) = A m[j] A* . . .
     */
    sl2c_adjoint(A->matrix, ad_A);
    sl2c_product(A->matrix, m[j], temp);
    sl2c_product(temp, ad_A, fAmj);

    /*
     * . . . and find its components relative to the basis m[].
     */
    B[0][j] = 0.5 * (fAmj[0][0].real + fAmj[1][1].real);
    B[1][j] = 0.5 * (fAmj[0][0].real - fAmj[1][1].real);
    B[2][j] = fAmj[0][1].real;
    B[3][j] = fAmj[0][1].imag;
}

/*
 * If A->parity is orientation_reversing, multiply on the
 * right by
 *
 *      1  0  0  0
 *      0  1  0  0
 *      0  0  1  0
 *      0  0  0 -1
 */

if (A->parity == orientation_reversing)
    for (i = 0; i < 4; i++)
        B[i][3] = - B[i][3];
}

void O31_to_Moebius(
    O31Matrix B,
    MoebiusTransformation *A)
{
    /*
     * We want to invert the transformation described in Moebius_to_O31().

```

```

*
* If the isometry is orientation_reversing (i.e. if det(B) == -1),
* we first factor it as
*
*      ( original )      ( new ) (1 0 0 0)
*      ( matrix )   =   ( matrix ) (0 1 0 0)
*      ( B )         ( B ) (0 0 1 0)
*      ( )           ( ) (0 0 0 -1)
*
* We then convert the "new matrix B" to an SL(2,C) matrix to get
* A->matrix, and we set A->parity to orientation_reversing.
* The matrix on the far right (= diag(1, 1, 1, -1)) corresponds
* to complex conjugation, as explained in the documentation in
* Moebius_to_O31() above.
*
* First write out what Moebius_to_O31() does to a typical Moebius
* transformation A. Assume for the moment that the Moebius
* transformation is orientation_preserving, and denote the matrix
* by A (rather than A->matrix) to save space. As usual, the entries
* of A are
*
*      a  b
*      c  d
*
* and the complex conjugate of a number z is written z' (read "z-bar").
* You should also imagine a single set of parentheses around each
* 2 x 2 matrix, in spite of the limitations of this text-only file.
* Each of the following lines computes A M A* for one of the four
* basis vectors {M0, M1, M2, M3}.
*
* (a b) ( 1 0) (a' c') = ( aa' + bb'   ac' + bd' )
* (c d) ( 0 1) (b' d')   ( a'c + b'd   cc' + dd' )
*
* (a b) ( 1 0) (a' c') = ( aa' - bb'   ac' - bd' )
* (c d) ( 0 -1) (b' d')   ( a'c - b'd   cc' - dd' )
*
* (a b) ( 0 1) (a' c') = ( ab' + a'b   ad' + bc' )
* (c d) ( 1 0) (b' d')   ( a'd + b'c   cd' + c'd )
*
* (a b) ( 0 i) (a' c') = i ( ab' - a'b   ad' - bc' )
* (c d) (-i 0) (b' d')   (-a'd + b'c   cd' - c'd )
*
* The right side of each of the above equations is a
* linear combination of the {M0, M1, M2, M3}. The coefficients
* are the entries of the O(3,1) matrix B. For the j-th line above,
*
* A M[j] A*
*
* = B[0][j] M0 + B[1][j] M1 + B[2][j] M2 + B[3][j] M3
*
* = B[0][j] (1 0) + B[1][j] (1 0) + B[2][j] (0 1) + B[3][j] ( 0 i)
*           (0 1)           (0 -1)           (1 0)           (-i 0)
*
* = ( B[0][j] + B[1][j]   B[2][j] + i B[3][j] )
*   ( B[2][j] - i B[3][j]   B[0][j] + B[1][j] )
*
* Comparing matrix entries in the two computations of A M[j] A* gives
* relations like aa' + bb' = B[0][0] + B[1][0], etc.
* There is no need to write out all 16 relations -- we'll get the
* ones we need later on, as we need them.
*
* It's not so easy to compute the matrix
*
*      a  b
*      c  d
*
* from the above relations, but it is easy to compute
*
*      2a'a  2a'b   or   2b'a  2b'b
*      2a'c  2a'd       2b'c  2b'd
*
* (The details are in the code below.)
* Each of the latter two matrices, if nonzero, can be normalized
* to give the former. At least one of them will be nonzero,

```

```

* because if a and b were both zero, the determinant would
* be zero.
*
* So . . . the algorithm is to decide whether a or b is
* nonzero, then compute one of the above two matrices, and
* normalize it to give the matrix A.
*/

int      i;
double  AM0A_00,      /* The (0, 0) entry of A M0 A*      */
        AM1A_00,      /* The (0, 0) entry of A M1 A*      */
        aa,           /* 2 * |a|^2                        */
        bb;           /* 2 * |b|^2                        */

/*
* Now deal with the orientation, as explained at the beginning
* of this function's documentation.  gl4R_determinant(B) will be
*
*      +1 if the isometry is orientation_preserving
*
*      -1 if the isometry is orientation_reversing
*/

if (gl4R_determinant(B) > 0.0)
    A->parity = orientation_preserving;
else
{
    A->parity = orientation_reversing;

    /*
    * Factor out diag(1, 1, 1, -1), as explained above.
    * At the end of the function we'll restore B to its
    * original condition.
    */

    for (i = 0; i < 4; i++)
        B[i][3] = - B[i][3];
}

/*
* From above,
*
*      (A M0 A*)[0][0] = aa' + bb' = B[0][0] + B[1][0]
*      (A M1 A*)[0][0] = aa' - bb' = B[0][1] + B[1][1]
*
* =>
*      2aa' = (A M0 A*)[0][0] + (A M1 A*)[0][0]
*            = (B[0][0] + B[1][0]) + (B[0][1] + B[1][1])
*
*      2bb' = (A M0 A*)[0][0] - (A M1 A*)[0][0]
*            = (B[0][0] + B[1][0]) - (B[0][1] + B[1][1])
*/

AM0A_00 = B[0][0] + B[1][0];
AM1A_00 = B[0][1] + B[1][1];
aa = AM0A_00 + AM1A_00;
bb = AM0A_00 - AM1A_00;

if (aa > bb)      /* |a| > |b| */
{
    A->matrix[0][0].real = aa;          /* 2a'a */
    A->matrix[0][0].imag = 0;

    /*
    * (A M2 A*)[0][0] = ab' + a'b
    *                  = 2 Re(a'b)
    *                  = B[0][2] + B[1][2]
    *
    * (A M3 A*)[0][0] = i (ab' - a'b)
    *                  = i (-2i Im(a'b))
    *                  = 2 Im(a'b)
    *                  = B[0][3] + B[1][3]
    */
}

```

```

    */

    A->matrix[0][1].real = B[0][2] + B[1][2];      /* 2a'b */
    A->matrix[0][1].imag = B[0][3] + B[1][3];

    /*
    *      a'c + b'd = (A M0 A*)[1][0] = B[2][0] - i B[3][0]
    *      a'c - b'd = (A M1 A*)[1][0] = B[2][1] - i B[3][1]
    *
    * => 2a'c = (B[2][0] + B[2][1]) - i (B[3][0] + B[3][1])
    */

    A->matrix[1][0].real = B[2][0] + B[2][1];      /* 2a'c */
    A->matrix[1][0].imag = - B[3][0] - B[3][1];

    /*
    *      a'd + b'c = (A M2 A*)[1][0] = B[2][2] - i B[3][2]
    *      -i (a'd - b'c) = (A M3 A*)[1][0] = B[2][3] - i B[3][3]
    *
    * => 2a'd = (B[2][2] + B[3][3]) + i (B[2][3] - B[3][2])
    */

    A->matrix[1][1].real = B[2][2] + B[3][3];      /* 2a'd */
    A->matrix[1][1].imag = B[2][3] - B[3][2];
}
else      /* |b| >= |a| */
{
    /*
    *      (A M2 A*)[0][0] = b'a + ba'
    *                      = 2 Re(b'a)
    *                      = B[0][2] + B[1][2]
    *
    *      (A M3 A*)[0][0] = i (b'a - ba')
    *                      = i ( 2i Im(b'a) )
    *                      = -2 Im(b'a)
    *                      = B[0][3] + B[1][3]
    */

    A->matrix[0][0].real = B[0][2] + B[1][2];      /* 2b'a */
    A->matrix[0][0].imag = - B[0][3] - B[1][3];

    A->matrix[0][1].real = bb;                      /* 2b'b */
    A->matrix[0][1].imag = 0;

    /*
    *      b'c + a'd = (A M2 A*)[1][0] = B[2][2] - i B[3][2]
    *      i (b'c - a'd) = (A M3 A*)[1][0] = B[2][3] - i B[3][3]
    *
    * => 2b'c = (B[2][2] - B[3][3]) - i (B[2][3] + B[3][2])
    */

    A->matrix[1][0].real = B[2][2] - B[3][3];      /* 2b'c */
    A->matrix[1][0].imag = - B[2][3] - B[3][2];

    /*
    *      b'd + a'c = (A M0 A*)[1][0] = B[2][0] - i B[3][0]
    *      - (b'd - a'c) = (A M1 A*)[1][0] = B[2][1] - i B[3][1]
    *
    * => 2b'd = (B[2][0] - B[2][1]) + i (B[3][1] - B[3][0])
    */

    A->matrix[1][1].real = B[2][0] - B[2][1];      /* 2b'd */
    A->matrix[1][1].imag = B[3][1] - B[3][0];
}

/*
 * Normalize A to have determinant one.
 */

sl2c_normalize(A->matrix);

/*

```

```
    * If the isometry is orientation_reversing, multiply back in
    * the diag(1, 1, 1, -1) which we factored out at the beginning.
    */

    if (A->parity == orientation_reversing)
        for (i = 0; i < 4; i++)
            B[i][3] = - B[i][3];
}

Boolean O3l_determinants_OK(
    O3lMatrix    arrayB[],
    int          num_matrices,
    double       epsilon)
{
    int i;

    for (i = 0; i < num_matrices; i++)
        if (fabs(fabs(gl4R_determinant(arrayB[i])) - 1.0) > epsilon)
            return FALSE;

    return TRUE;
}
```