

```

/*
 * o3l_matrices.c
 *
 * This file provides the following functions for working with O3lMatrices:
 *
 * void o3l_copy(O3lMatrix dest, O3lMatrix source);
 * void o3l_invert(O3lMatrix m, O3lMatrix m_inverse);
 * FuncResult gl4R_invert(GL4RMatrix m, GL4RMatrix m_inverse);
 * double gl4R_determinant(GL4RMatrix m);
 * void o3l_product(O3lMatrix a, O3lMatrix b, O3lMatrix product);
 * Boolean o3l_equal(O3lMatrix a, O3lMatrix b, double epsilon);
 * double o3l_trace(O3lMatrix m);
 * double o3l_deviation(O3lMatrix m);
 * void o3l_GramSchmidt(O3lMatrix m);
 * void o3l_conjugate(O3lMatrix m, O3lMatrix t, O3lMatrix Tmt);
 * double o3l_inner_product(O3lVector u, O3lVector v);
 * void o3l_matrix_times_vector(O3lMatrix m, O3lVector v, O3lVector product);
 * void o3l_constant_times_vector(double r, O3lVector v, O3lVector product);
 * void o3l_copy_vector(O3lVector dest, O3lVector source);
 * void o3l_vector_sum(O3lVector a, O3lVector b, O3lVector sum);
 * void o3l_vector_diff(O3lVector a, O3lVector b, O3lVector diff);
 */

#include "kernel.h"

/*
 * gl4R_invert will consider a matrix to be singular iff one of the
 * absolute value of one of the pivots is less than SINGULAR_MATRIX_EPSILON.
 */
#define SINGULAR_MATRIX_EPSILON 1e-2

#define COLUMN_PRODUCT(m, i, j) \
    (-m[0][i]*m[0][j] + m[1][i]*m[1][j] + m[2][i]*m[2][j] + m[3][i]*m[3][j])

O3lMatrix O3l_identity = {
    {1.0, 0.0, 0.0, 0.0},
    {0.0, 1.0, 0.0, 0.0},
    {0.0, 0.0, 1.0, 0.0},
    {0.0, 0.0, 0.0, 1.0}
};

void o3l_copy(
    O3lMatrix dest,
    O3lMatrix source)
{
    int i,
        j;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            dest[i][j] = source[i][j];
}

void o3l_invert(
    O3lMatrix m,
    O3lMatrix m_inverse)
{
    /*
     * The inverse of an O(3,1) matrix may be computed by taking
     * the transpose and then negating both the zeroth row and the
     * zeroth column. The proof follows easily from the fact that
     * multiplying an O(3,1) matrix by its transpose is almost the
     * same thing as computing the inner product of each pair of
     * columns. (For O(4) matrices, the transpose is precisely
     * the inverse, because there are no minus sign in the metric
     * to fuss over.)
     *
     * We first write the inverse into the O3lMatrix temp, so that if
     * the parameters m and m_inverse are the same O3lMatrix, we don't
     * overwrite m[j][i] before computing m[i][j].
     */
}

```

```

    int        i,
               j;
    O3lMatrix   temp;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            temp[i][j] = ((i == 0) == (j == 0)) ? m[j][i] : -m[j][i];

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            m_inverse[i][j] = temp[i][j];
}

```

```

FuncResult gl4R_invert(
    GL4RMatrix m,
    GL4RMatrix m_inverse)
{
    double      row[4][8];
    double      *mm[4],
               *temp_row,
               multiple;

    int         i,
               j,
               k;

    for (i = 0; i < 4; i++)
        mm[i] = row[i];

    /*
     * Copy m -- don't alter the original.
     */
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            mm[i][j] = m[i][j];

    /*
     * Initialize the four right hand columns to the identity.
     */
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            mm[i][4 + j] = (i == j) ? 1.0 : 0.0;

    /*
     * Forward elimination.
     */
    for (j = 0; j < 4; j++)
    {
        /*
         * Partial pivoting.
         */
        for (i = j+1; i < 4; i++)
            if (fabs(mm[j][j]) < fabs(mm[i][j]))
            {
                temp_row    = mm[j];
                mm[j]        = mm[i];
                mm[i]        = temp_row;
            }

        /*
         * Is the matrix singular?
         */
        if (fabs(mm[j][j]) < SINGULAR_MATRIX_EPSILON)
            return func_bad_input;

        /*
         * Divide through to get a 1.0 on the diagonal.
         */
        multiple = 1.0 / mm[j][j];
        for (k = j; k < 8; k++)
            mm[j][k] *= multiple;

        /*
         * Clear out that column.
         */
    }
}

```

```

    */
    for (i = j+1; i < 4; i++)
    {
        multiple = mm[i][j];
        for (k = j; k < 8; k++)
            mm[i][k] -= multiple * mm[j][k];
    }
}

/*
 * Back substitution.
 */
for (j = 4; --j >= 0; )
    for (i = j; --i >= 0; )
        for (k = 4; k < 8; k++)
            mm[i][k] -= mm[i][j] * mm[j][k];

/*
 * Copy out the solution.
 */
for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
        m_inverse[i][j] = mm[i][4 + j];

return func_OK;
}

double gl4R_determinant(
    GL4RMatrix m)
{
    /*
     * Two approaches come to mind for computing a 4 x 4 determinant.
     *
     * (1) Work out the 4! = 24 terms -- each a product of four
     *     matrix entries -- and form their alternating sum.
     *
     * (2) Use Gaussian elimination.
     *
     * I doubt there is much difference in efficiency between the two
     * methods, so I've chosen method (2) on the assumption (which I
     * haven't thoroughly thought out) that it will be numerically more
     * robust. Numerical effects could be noticeable, because O(3,1)
     * matrices tend to have large entries. On the other hand, the
     * determinant will always be plus or minus one, so it's not worth
     * getting too concerned about the precision.
     */

    /*
     * gl4R_determinant() no longer calls uFatalError() when the determinant
     * is something other than +1 or -1. This lets compute_approx_volume()
     * in Dirichlet_extras.c compute the determinant of matrices which
     * are in gl(2,C) but not O(3,1). Note that matrix_io.c already calls
     * O3l_determinants_OK() to validate matrices read from files, and
     * that SnapPea has had no trouble with determinants of internally
     * computed O(3,1) matrices.
     *
     * JRW 94/11/30
     */

    int
        r,
        c,
        cc,
        pivot_row,
        row_swaps;
    double
        max_abs,
        this_abs,
        temp,
        factor,
        det;
    O3lMatrix mm;

    /*
     * First copy the matrix, to avoid destroying it as

```

```

    * we compute its determinant.
    */

o3l_copy(mm, m);

/*
 * Put the matrix in upper triangular form.
 *
 * Count the number of row swaps, so we can get the
 * correct sign at the end.
 *
 * Technical comment: We don't actually write zeros into the
 * lower part of the matrix; we just pretend.
 */

row_swaps = 0;

for (c = 0; c < 4; c++)
{
    /*
     * Find the pivot row.
     */

    max_abs = -1.0;

    for (r = c; r < 4; r++)
    {
        this_abs = fabs(mm[r][c]);
        if (this_abs > max_abs)
        {
            max_abs = this_abs;
            pivot_row = r;
        }
    }

    if (max_abs == 0.0)
    /*
     * The determinant of an O(3,1) matrix should always
     * be plus or minus one, never zero.
     */
    /* uFatalError("gl4R_determinant", "o3l_matrices"); */
    return 0.0; /* JRW 94/11/30 (see explanation above) */

    /*
     * Swap the pivot row into position.
     */

    if (pivot_row != c)
    {
        for (cc = c; cc < 4; cc++)
        {
            temp = mm[c][cc];
            mm[c][cc] = mm[pivot_row][cc];
            mm[pivot_row][cc] = temp;
        }
        row_swaps++;
    }

    /*
     * Eliminate the entries in column c which lie below the pivot.
     */

    for (r = c + 1; r < 4; r++)
    {
        factor = - mm[r][c] / mm[c][c];

        for (cc = c + 1; cc < 4; cc++)
            mm[r][cc] += factor * mm[c][cc];
    }
}

/*
 * The determinant is now the product of the diagonal entries.
 */

```

```

    det = 1.0;

    for (c = 0; c < 4; c++)
        det *= mm[c][c];

    if (row_swaps % 2)
        det = - det;

    /*
     * Do a quick error check, just to be safe.
     * The determinant of an O3l_matrix should be +1 or -1.
     */

    /*
    commented out by JRW 94/11/30 (see explanation above)

        if (fabs(fabs(det) - 1.0) > 0.01)
            uFatalError("gl4R_determinant", "o3l_matrices");
    */

    return det;
}

void o3l_product(
    O3lMatrix  a,
    O3lMatrix  b,
    O3lMatrix  product)
{
    register int    i,
                   j,
                   k;
    register double sum;
    O3lMatrix      temp;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
        {
            sum = 0.0;
            for (k = 0; k < 4; k++)
                sum += a[i][k] * b[k][j];
            temp[i][j] = sum;
        }

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            product[i][j] = temp[i][j];
}

Boolean o3l_equal(
    O3lMatrix  a,
    O3lMatrix  b,
    double      epsilon)
{
    /*
     * There are a number of different ways one could decide whether two
     * O(3,1) matrices are the same or not. The fancier ways, such as
     * computing the sum of the squares of the differences of corresponding
     * entries, are numerically more time consuming. For now let's just
     * check that all entries are equal to within epsilon. This offers the
     * advantage that when scanning down lists, the vast majority of
     * matrices are diagnosed as different after the comparison of a
     * single pair of numbers. The epsilon can be fairly large, since to
     * qualify as equal, two matrices must have ALL their entries equal to
     * within that precision.
     */

    int i,
        j;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)

```

```

        if (fabs(a[i][j] - b[i][j]) > epsilon)
            return FALSE;

    return TRUE;
}

double o3l_trace(
    O3lMatrix  m)
{
    int      i;
    double   trace;

    trace = 0.0;

    for (i = 0; i < 4; i++)
        trace += m[i][i];

    return trace;
}

double o3l_deviation(
    O3lMatrix  m)
{
    /*
     * The matrix m is, in theory, an element of SO(3,1),
     * so the inner product of column i with column j should be
     *
     *           -1      if i = j = 0,
     *           +1      if i = j != 0, or
     *           0       if i != j.
     *
     * Return the greatest deviation from these values, so the
     * calling function has some idea how precise the matrix is.
     *
     * The simplest way to code this is to multiply the matrix times its
     * inverse. Note that this approach relies on the fact that
     * o3l_inverse() transposes the matrix and negates the appropriate
     * entries. If o3l_inverse() did Gaussian elimination to numerically
     * invert the matrix, we'd have to rewrite the following code.
     */

    O3lMatrix  the_inverse,
               the_product;
    double     error,
               max_error;
    int        i,
               j;

    o3l_invert(m, the_inverse);
    o3l_product(m, the_inverse, the_product);

    max_error = 0.0;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
        {
            error = fabs(the_product[i][j] - (i == j ? 1.0 : 0.0));
            if (error > max_error)
                max_error = error;
        }

    return max_error;
}

void o3l_GramSchmidt(
    O3lMatrix  m)
{
    /*
     * Given a matrix m whose columns are almost orthonormal (in the sense
     * of O(3,1), not O(4)), use the Gram-Schmidt process to make small
     * changes to the matrix entries so that the columns become orthonormal

```

```

    * to the highest precision possible.
    */

int    r,
      c,
      cc;
double length,
      length_of_projection;

for (c = 0; c < 4; c++)
{
    /*
     * Adjust column c to have length -1 (if c == 0) or +1 (if c > 0).
     * We are assuming m is already close to being in O(3,1), so
     * it suffices to divide column c by sqrt(fabs(length)).
     */
    length = sqrt(fabs(COLUMN_PRODUCT(m, c, c))); /* no need for safe_sqrt() */
    for (r = 0; r < 4; r++)
        m[r][c] /= length;

    /*
     * We want to make all subsequent columns be orthogonal to column c,
     * so subtract off their components in the direction of column c.
     * Because column c is now a unit vector, the inner product
     * <column c, column cc> gives plus or minus the length of the
     * projection of column cc onto column c, according to whether or
     * not c == 0.
     */
    for (cc = c + 1; cc < 4; cc++)
    {
        length_of_projection = COLUMN_PRODUCT(m, c, cc);
        if (c == 0)
            length_of_projection = - length_of_projection;
        for (r = 0; r < 4; r++)
            m[r][cc] -= length_of_projection * m[r][c];
    }
}

void o3l_conjugate(
    O3lMatrix  m,
    O3lMatrix  t,
    O3lMatrix  Tmt)
{
    /*
     * Replace m with (t^-1) m t.
     */

    O3lMatrix  t_inverse,
              temp;

    o3l_invert(t, t_inverse);
    o3l_product(t_inverse, m, temp);
    o3l_product(temp, t, Tmt);
}

double o3l_inner_product(
    O3lVector  u,
    O3lVector  v)
{
    int    i;
    double sum;

    sum = - u[0]*v[0];

    for (i = 1; i < 4; i++)
        sum += u[i]*v[i];

    return sum;
}

```

```
void o3l_matrix_times_vector(
    O3lMatrix    m,
    O3lVector     v,
    O3lVector     product)
{
    register int    i,
                  j;
    register double sum;
    O3lVector       temp;

    for (i = 0; i < 4; i++)
    {
        sum = 0.0;
        for (j = 0; j < 4; j++)
            sum += m[i][j] * v[j];
        temp[i] = sum;
    }

    for (i = 0; i < 4; i++)
        product[i] = temp[i];
}
```

```
void o3l_constant_times_vector(
    double        r,
    O3lVector     v,
    O3lVector     product)
{
    int           i;

    for (i = 0; i < 4; i++)
        product[i] = r * v[i];
}
```

```
void o3l_copy_vector(
    O3lVector     dest,
    O3lVector     source)
{
    int i;

    for (i = 0; i < 4; i++)
        dest[i] = source[i];
}
```

```
void o3l_vector_sum(
    O3lVector     a,
    O3lVector     b,
    O3lVector     sum)
{
    int i;

    for (i = 0; i < 4; i++)
        sum[i] = a[i] + b[i];
}
```

```
void o3l_vector_diff(
    O3lVector     a,
    O3lVector     b,
    O3lVector     diff)
{
    int i;

    for (i = 0; i < 4; i++)
        diff[i] = a[i] - b[i];
}
```