

```

/*
 * Dirichlet_precision.c
 *
 * This file provides the functions
 *
 * void precise_generators(MatrixPairList *gen_list);
 * void precise_matrix(O3lMatrix m);
 * Boolean precise_double(double *x);
 *
 * which address the nasty issue of roundoff error in O3lMatrices.
 *
 * The root of the problem is that large numbers may occur in the O(3,1)
 * matrices, and when you multiply two large numbers x and y with
 * respective errors dx and dy, you get (x + dx)(y + dy) = xy + ydx + xdy
 * + (ignore second order term). The errors ydx and xdy aren't so bad
 * compared to the product xy (if x and y are in the thousands, xy will be
 * in the millions). But when you do the matrix multiplication you end up
 * summing four such terms, and the four xy's largely cancel, so you're
 * left with a sum of reasonable magnitude (same magnitude as x and y).
 * But the error is still of magnitude ydx or xdy. In other words, each
 * time you do a matrix multiplication, the error is multiplied by the
 * magnitude of the entries! For matrices with large entries (10^5 or
 * 10^6 is not extreme), these errors become completely unacceptable after
 * only a few matrix multiplications.
 *
 * In general, there is no satisfactory solution to avoiding the roundoff
 * error. Fortunately, 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).
 * (I'm still investigating this, and working on proofs. I might add to
 * this documentation later.) 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.
 *
 * precise_o3l_product() multiplies two O3lMatrices with an eye to
 * recognizes nice numbers. precise_generators() tries to recognize
 * nice numbers in an initial list of generators.
 */

#include "kernel.h"
#include "Dirichlet.h"

#define EPSILON (1e2 * DBL_EPSILON)
#define DEFAULT_EPSILON (1e6 * DBL_EPSILON)

#define ROOT2 1.41421356237309504880
#define ROOT3 1.73205080756887729352

static void precise_matrix(O3lMatrix m);
static Boolean precise_trace(O3lMatrix m);
static Boolean precise_double(double *x, double epsilon);

void precise_o3l_product(
    O3lMatrix a,
    O3lMatrix b,
    O3lMatrix product)
{
    int i,
        j,
        k;

    double sum,
            abs_sum,
            term;

    O3lMatrix temp;

    /*
     * If a and b don't have precise traces, then we don't waste our time
     * looking for precise entries.
     */
    if (precise_trace(a) == FALSE || precise_trace(b) == FALSE)
    {
        o3l_product(a, b, product);
    }
}

```

```

    return;
}

/*
 * As explained at the top of this file, the product of two numbers
 * x and y with respective roundoff errors dx and dy will be
 * (x + dx)(y + dy) = xy + xdy + ydx + (ignore second order term).
 * If the original factors are known to maximum precision, then
 * dx ~ x*DBL_EPSILON and dy ~ y*DBL_EPSILON, so the expected error
 * in the product will be about x*y*DBL_EPSILON. That is, the
 * product will be known to (almost) maximum precision when the two
 * factors are.
 *
 * The error in a matrix multiplication lies not in the
 * multiplication of the entries, but in the addition of the products
 * of entries. Several large numbers may partially cancel each other
 * when added, giving a sum whose absolute value is much less than that
 * of any of the factors. Unfortunately, the errors don't cancel so
 * nicely, so we sometimes end up with a small sum and a large error.
 */

for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
    {
        sum = 0.0;
        abs_sum = 0.0;
        for (k = 0; k < 4; k++)
        {
            term = a[i][k] * b[k][j];
            sum += term;
            abs_sum += fabs(term);
        }
        precise_double(&sum, abs_sum*EPSILON);
        temp[i][j] = sum;
    }

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

void precise_generators(
    MatrixPairList *gen_list)
{
    MatrixPair *matrix_pair;

    for (matrix_pair = gen_list->begin.next;
         matrix_pair != &gen_list->end;
         matrix_pair = matrix_pair->next)
    {
        precise_matrix(matrix_pair->m[0]);
        o3l_invert(matrix_pair->m[0], matrix_pair->m[1]);
    }
}

static void precise_matrix(
    O3lMatrix m)
{
    int i,
        j;

    /*
     * First check the trace of m.
     * If the trace isn't a nice number, give up now.
     */

    if (precise_trace(m) == FALSE)
        return;

    /*
     * Look at each entry in the O3lMatrix m, and see whether it's a nice
     * number. If so, write in the exact value to full precision. We don't

```

```

    * know where this matrix came from, so use the DEFAULT_EPSILON.
    */

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            (void) precise_double(&m[i][j], fabs(m[i][j])*DEFAULT_EPSILON);
}

static Boolean precise_trace(
    O3lMatrix m)
{
    int i;
    double trace,
           sum_abs;

    trace = 0.0;
    sum_abs = 0.0;

    for (i = 0; i < 4; i++)
    {
        trace += m[i][i];
        sum_abs += fabs(m[i][i]);
    }

    return precise_double(&trace, sum_abs*DEFAULT_EPSILON);
}

static Boolean precise_double(
    double *x,
    double epsilon)
{
    double x4,
           x_int,
           x_root2,
           x_root2_int,
           x_root3,
           x_root3_int;

    /*
     * If *x is nonzero, then the given value of epsilon will be fine.
     * But it may not work so well for recognizing zero, so we check
     * for zero as a special case.
     */

    if (fabs(*x) < DEFAULT_EPSILON)
    {
        *x = 0.0;
        return TRUE;
    }

    /*
     * We're interested in quarter integer multiples of 1, sqrt(2) and
     * sqrt(3), so multiply *x by 4.
     */
    x4 = 4 * (*x);

    /*
     * Is x4 an integer?
     */

    x_int = floor(x4 + 0.5);

    if (fabs(x4 - x_int) < epsilon)
    {
        *x = x_int / 4.0;
        return TRUE;
    }

    /*
     * Is x4 an integer multiple of sqrt(2)?

```

```
    */

    x_root2      = x4 / ROOT2;
    x_root2_int = floor(x_root2 + 0.5);

    if (fabs(x_root2 - x_root2_int) < epsilon)
    {
        *x = (x_root2_int / 4.0) * ROOT2;
        return TRUE;
    }

    /*
    * Is x4 an integer multiple of sqrt(3)?
    */

    x_root3      = x4 / ROOT3;
    x_root3_int = floor(x_root3 + 0.5);

    if (fabs(x_root3 - x_root3_int) < epsilon)
    {
        *x = (x_root3_int / 4.0) * ROOT3;
        return TRUE;
    }

    return FALSE;
}
```