

```

/*
 * solve_equations.c
 *
 * This file provides the functions
 *
 *      FuncResult solve_complex_equations(Complex **complex_equations,
 *                                          int num_rows, int num_columns, Complex *solution);
 *      FuncResult solve_real_equations(double **real_equations,
 *                                       int num_rows, int num_columns, double *solution);
 *
 * which do_Dehn_filling() in hyperbolic_structure.c calls to
 * solve num_rows linear equations in num_columns variables.  Gaussian
 * elimination with partial pivoting is used.
 *
 * The equations are stored as an array of num_rows pointers, each
 * of which points to an array of (num_columns + 1) entries.  The
 * last entry in each row is the constant on the right hand side of
 * the equation.
 *
 * num_rows is assumed to be greater than or equal to num_columns.
 * The equations are assumed to have rank exactly equal to num_columns.
 * Thus, even though there may be more equations than variables, the
 * equations are assumed to be consistent.
 *
 * Even though these routines make no assumption about the origin or
 * purpose of the equations, their main use is, of course, to find
 * hyperbolic structures.  My hope in including all the equations
 * (rather than just a linearly independent subset) is that we will
 * get more accurate solutions, particularly in degenerate or
 * nearly degenerate situations.
 *
 * These functions assume an array of num_columns elements has already
 * been allocated for the solution.
 *
 * Technical detail:  While doing the Gaussian elimination, these functions
 * do not actually compute or write values which are guaranteed to be one
 * or zero, but they know where they are.  So don't worry that in the end
 * the matrix does not contain all ones and zeros; the matrix will be a
 * mess, but the solution will be correct.
 */

```

```

#include "kernel.h"

```

```

FuncResult solve_complex_equations(
    Complex **complex_equations,
    int      num_rows,
    int      num_columns,
    Complex *solution)
{
    /*
     * The following register variables are used in the n^3 bottleneck.
     * (See below.)
     */

    register double    factor_real,
                      factor_imag;
    register Complex   *row_r,
                      *row_c;
    register int        count;

    /*
     * The remaining variables are used in less critical places.
     */

    int    r,
           c,
           cc,
           pivot_row = -1;
    double max_modulus,
           this_modulus,
           max_error,
           error;
    Complex *temp,

```

```

        factor;

/*
 * Forward elimination.
 */

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

    max_modulus = 0.0;

    for (r = c; r < num_rows; r++)
    {
        this_modulus = complex_modulus(complex_equations[r][c]);
        if (this_modulus > max_modulus)
        {
            max_modulus = this_modulus;
            pivot_row = r;
        }
    }

    if (max_modulus == 0.0)        /* In the old snappea, max_modulus */
        return func_failed;      /* was was never below 1e-100, even */
                                /* in degenerate cases. */

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

    temp = complex_equations[c];
    complex_equations[c] = complex_equations[pivot_row];
    complex_equations[pivot_row] = temp;

    /*
     * Multiply the pivot row through by 1.0/(pivot value).
     */

    factor = complex_div(One, complex_equations[c][c]);

    for (cc = c + 1; cc <= num_columns; cc++)
        complex_equations[c][cc] = complex_mult(
            factor,
            complex_equations[c][cc]
        );

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

    for (r = c + 1; r < num_rows; r++)
    {
        /*
         * The following loop is the bottleneck for computing
         * hyperbolic structures. It is executed n^3 times to solve
         * an n x n system of equations, and no other n^3 algorithms
         * are used. For this reason, I've written the loop to
         * maximize speed at the expense of readability.
         *
         * Here's the loop in pseudocode:
         *
         for (cc = c + 1; cc <= num_columns; cc++)
             complex_equations[r][cc]
                 -= complex_equations[r][c] * complex_equations[c][cc]

         *
         * Here's a version that will actually run:
         *
         for (cc = c + 1; cc <= num_columns; cc++)
             complex_equations[r][cc] = complex_minus(

```

```

        complex_equations[r][cc],
        complex_mult(
            complex_equations[r][c],
            complex_equations[c][cc]
        )
    );

    *
    * And here's the fancy, built-for-speed version:
    */

    factor_real = - complex_equations[r][c].real;
    factor_imag = - complex_equations[r][c].imag;

    if (factor_real || factor_imag)
    {
        row_r = complex_equations[r] + c + 1;
        row_c = complex_equations[c] + c + 1;

        for (count = num_columns - c; --count >= 0; )
        {
            if (row_c->real || row_c->imag)
            {
                row_r->real +=
                    factor_real * row_c->real
                    - factor_imag * row_c->imag;
                row_r->imag +=
                    factor_real * row_c->imag
                    + factor_imag * row_c->real;
            }
            row_r++;
            row_c++;
        }
    }

    /*
    * With all the THINK C compiler's optimization options on,
    * the fancy version runs 8 times faster than the plain
    * version. With all THINK C optimization options off,
    * it runs 9 times faster. The THINK C optimizer increases
    * the speed of the fancy code by only 6%.
    */

    /*
    * Yield some time to the window system, and check
    * whether the user has cancelled this computation.
    */

    if (uLongComputationContinues() == func_cancelled)
        return func_cancelled;
    }
}

/*
 * Back substitution.
 */

for (c = num_columns; --c > 0; ) /* Do columns (num_columns - 1) to 1, */
                                /* but skip column 0. */
    for (r = c; --r >= 0; ) /* Do rows (c - 1) to 0. */
        complex_equations[r][num_columns] = complex_minus(
            complex_equations[r][num_columns],
            complex_mult(
                complex_equations[r][c],
                complex_equations[c][num_columns]
            )
        );

/*
 * Check "extra" rows for consistency.
 * That is, in each of the last (num_rows - num_columns) rows,
 * check that the constant on the right hand side is zero.
 */

```

```

    * This will give us a measure of the accuracy of the solution.
    * I still haven't decided what to do with this number.
    */

    max_error = 0.0;

    for (r = num_columns; r < num_rows; r++)
    {
        error = complex_modulus(complex_equations[r][num_columns]);
        if (error > max_error)
            max_error = error;
    }

    /*
    * Record the solution.
    */

    for (r = 0; r < num_columns; r++)
        solution[r] = complex_equations[r][num_columns];

    return func_OK;
}

FuncResult solve_real_equations(
    double **real_equations,
    int     num_rows,
    int     num_columns,
    double *solution)
{
    /*
    * The following register variables are used in the n^3 bottleneck.
    * (See below.)
    */

    register double factor,
                   *row_r,
                   *row_c;
    register int    count;

    /*
    * The remaining variables are used in less critical places.
    */

    int     r,
           c,
           cc,
           pivot_row = -1;
    double  max_abs,
           this_abs,
           max_error,
           error,
           *temp;

    /*
    * Forward elimination.
    */

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

        max_abs = 0.0;

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

```

```

    }

    if (max_abs == 0.0)
        return func_failed;

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

    temp = real_equations[c];
    real_equations[c] = real_equations[pivot_row];
    real_equations[pivot_row] = temp;

    /*
     * Multiply the pivot row through by 1.0/(pivot value).
     */

    factor = 1.0 / real_equations[c][c];

    for (cc = c + 1; cc <= num_columns; cc++)
        real_equations[c][cc] *= factor;

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

    for (r = c + 1; r < num_rows; r++)
    {
        factor = - real_equations[r][c];

        /*
         * The following loop is the bottleneck for computing
         * hyperbolic structures. It is executed n^3 times to solve
         * an n x n system of equations, and no other n^3 algorithms
         * are used. For this reason, I've written the loop to
         * maximize speed at the expense of readability.
         *
         * Here's the loop in its humanly comprehensible form:
         */

        if (factor)
            for (cc = c + 1; cc <= num_columns; cc++)
                real_equations[r][cc] += factor * real_equations[c][cc];

        /*
         * Here's the optimized version of the same thing:
         */

        if (factor)
        {
            row_r = real_equations[r] + c + 1;
            row_c = real_equations[c] + c + 1;
            for (count = num_columns - c; --count >= 0; )
                *row_r++ += factor * *row_c++;
        }

        /*
         * Yield some time to the window system, and check
         * whether the user has cancelled this computation.
         */

        if (uLongComputationContinues() == func_cancelled)
            return func_cancelled;
    }
}

/*
 * Back substitution.
 */

for (c = num_columns; --c > 0; ) /* Do columns (num_columns - 1) to 1, */
    /* but skip column 0. */
    for (r = c; --r >= 0; ) /* Do rows (c - 1) to 0. */

```

```
        real_equations[r][num_columns] -= real_equations[r][c] * real_equations[c]
[num_columns];

/*
 * Check "extra" rows for consistency.
 * That is, in each of the last (num_rows - num_columns) rows,
 * check that the constant on the right hand side is zero.
 * This will give us a measure of the accuracy of the solution.
 * I still haven't decided what to do with this number.
 */

max_error = 0.0;

for (r = num_columns; r < num_rows; r++)
{
    error = fabs(real_equations[r][num_columns]);
    if (error > max_error)
        max_error = error;
}

/*
 * Record the solution.
 */

for (r = 0; r < num_columns; r++)
    solution[r] = real_equations[r][num_columns];

return func_OK;
}
```