

```

/*
 * hyperbolic_structure.c
 *
 * This file contains the following functions which the kernel
 * provides for the UI:
 *
 * SolutionType    find_complete_hyperbolic_structure(Triangulation *manifold);
 * SolutionType    do_Dehn_filling(Triangulation *manifold);
 * SolutionType    remove_Dehn_fillings(Triangulation *manifold);
 *
 * Their use is described in SnapPea.h.
 *
 * This file also provides the following functions for use
 * within the kernel
 *
 * void    remove_hyperbolic_structures(Triangulation *manifold);
 * void    polish_hyperbolic_structures(Triangulation *manifold);
 *
 * remove_hyperbolic_structures() frees the TetShapes (if any) pointed to
 * by each tet->shape[] and sets manifold->solution_type[complete] and
 * manifold->solution_type[filled] to not_attempted.
 *
 * polish_hyperbolic_structures() attempts to increase the accuracy of
 * both the complete and the Dehn filled hyperbolic structures already
 * present in *manifold. It's designed to be called following
 * retriangulation operations which diminish the accuracy of the TetShapes.
 *
 * SnapPea uses Newton's method to solve the gluing equations (see
 * Thurston's notes for an explanation of the gluing equations). The
 * linear equations generated at each iteration of Newton's method are
 * solved using Gaussian elimination with partial pivoting.
 *
 * The number of gluing equations is (number of tetrahedra + number of cusps),
 * while the number of variables is just the number of tetrahedra.
 * Unlike previous versions of SnapPea, this version does not select out
 * a linearly independent subset of the gluing equations, but rather
 * solves the whole system. My hope is that in cases where the gluing
 * equations are degenerate (or nearly so) the pivoting will tend to
 * select a more robust subset of the equations. In any case, once the
 * equations have been solved, there will be some rows of zeros at the
 * bottom, one for each cusp. The constants on the right hand side of
 * these zero rows provide a measure of how accurately the equations were
 * solved. For example, in the case of the Whitehead link complement,
 * which has four tetrahedra and two cusps, the matrix will reduce to
 *
 *          1   0   0   0   a  <- solution
 *          0   1   0   0   b
 *          0   0   1   0   c
 *          0   0   0   1   d
 *          0   0   0   0   e  <- should be zero
 *          0   0   0   0   f
 *
 * where the constants a - d represent the solution to the equations,
 * and the constants e - f (which will be close to zero) measure the
 * solution's accuracy.
 *
 * The coordinate systems used to parameterize the shapes of the
 * tetrahedra are chosen dynamically so as to avoid singularities.
 * The comment preceding the function choose_coordinate_system()
 * (see below) explains the underlying mathematics.
 *
 * The gluing equations are written in terms of complex variables,
 * namely the edge parameters of the tetrahedra. If the manifold is
 * oriented, they are analytic functions of these variables, and
 * Newton's method is applied directly. If the manifold is unoriented,
 * they are almost analytic, but not quite: they are analytic functions
 * of the variables and their complex conjugates. (Reversing the
 * orientation of a tetrahedron replaces its edge parameter with the
 * inverse of its complex conjugate.) Newton's method is applied by
 * writing the n x m system of complex equations as a 2n x 2m system of
 * real equations.
 *
 * [One could of course use real equations for oriented manifolds as

```

```

* well, but the speed suffers. The arithmetic involved in the row
* operations (multiplying an entry in one row by a constant and adding
* it to the corresponding entry in another row) is four times faster
* for real numbers than for complex numbers, but a  $2n \times 2m$  real system
* requires eight times as many such steps as does an  $n \times m$  complex system.
* Hence the speed decreases by a factor of two. This is why SnapPea
* handles oriented and unoriented manifolds differently. Other than
* loss of speed, there is no harm in passing an unoriented (but
* orientable) manifold, with manifold->orientability ==
* unknown_orientability).]
*
* do_Dehn_filling() computes the shape of each unfilled cusp and
* stores it in the field cusp->cusp_shape[current].
* find_complete_hyperbolic_structure(), after calling do_Dehn_filling(),
* copies cusp->cusp_shape[current] to cusp->cusp_shape[initial].
*/

#include "kernel.h"

const static ComplexWithLog regular_shape = {
    {0.5, ROOT_3_OVER_2},
    {0.0, PI_OVER_3}
};

/*
* RIGHT_BALLPARK must be set fairly large to allow for degenerate
* solutions, which cannot be computed to great accuracy.
*/

#define RIGHT_BALLPARK 1e-2
#define QUADRATIC_THRESHOLD 1e-4

/*
* If the solution is degenerate and Newton's method has been
* iterated at least DEGENERACY_ITERATIONS times, then
* do_Dehn_filling() will keep going iff the distance to
* the solution decreases by a factor of at least DEGENERACY_RATIO
* each time.
*/

#define DEGENERACY_ITERATIONS 10
#define DEGENERACY_RATIO 0.9

/*
* If we haven't converged and aren't making progress after
* ITERATION_LIMIT iterations, we give up.
*/

#define ITERATION_LIMIT 101

/*
* The CuspInfo and ChernSimonsInfo data structures are
* used only in polish_hyperbolic_structures().
*/

typedef struct
{
    Boolean is_complete;
    double m,
    l;
} CuspInfo;

typedef struct
{
    Boolean CS_value_is_known,
    CS_fudge_is_known;
    double CS_value[2],
    CS_fudge[2];
} ChernSimonsInfo;

static void allocate_cusp_status_arrays(Triangulation *manifold, Boolean **
is_complete_array, double **m_array, double **l_array);

```

```

static void      free_cusp_status_arrays(Boolean *is_complete_array, double *m_array,
double *l_array);
static void      record_cusp_status(Triangulation *manifold, Boolean is_complete_array[],
double m_array[], double l_array[]);
static void      restore_cusp_status(Triangulation *manifold, Boolean is_complete_array
[], double m_array[], double l_array[]);
static void      copy_tet_shapes(Triangulation *manifold, FillingStatus source,
FillingStatus dest);
static void      copy_cusp_shapes(Triangulation *manifold, FillingStatus source,
FillingStatus dest);
static void      verify_coefficients(Triangulation *manifold);
static void      allocate_equations(Triangulation *manifold, Complex ***
complex_equations, double ***real_equations, int *num_rows, int *num_columns);
static void      free_equations(Triangulation *manifold, Complex **complex_equations,
double **real_equations, int num_rows);
static void      allocate_complex_equations(Triangulation *manifold, Complex ***
complex_equations, int *num_rows, int *num_columns);
static void      allocate_real_equations(Triangulation *manifold, double ***
real_equations, int *num_rows, int *num_columns);
static void      free_complex_equations(Complex **complex_equations, int num_rows);
static void      free_real_equations(double **real_equations, int num_rows);
static void      associate_complex_eqns_to_edges_and_cusps(Triangulation *manifold,
Complex **complex_equations);
static void      associate_real_eqns_to_edges_and_cusps(Triangulation *manifold, double
**real_equations);
static void      dissociate_eqns_from_edges_and_cusps(Triangulation *manifold);
static void      choose_coordinate_system(Triangulation *manifold);
static Boolean   check_convergence(Orientability orientability, Complex **
complex_equations, double **real_equations, int num_rows, int num_columns, double *
distance_to_solution, Boolean *convergence_is_quadratic, double *distance_ratio);
static double    compute_distance_complex(Complex **complex_equations, int num_rows, int
num_columns);
static double    compute_distance_real(double **real_equations, int num_rows, int
num_columns);
static FuncResult solve_equations(Orientability orientability, Complex **
complex_equations, double **real_equations, int num_rows, int num_columns, Complex *
solution);
static void      convert_solution(double *real_solution, Complex *solution, int
num_columns);
static void      save_chern_simons(Triangulation *manifold, ChernSimonsInfo *
chern_simons_info);
static void      restore_chern_simons(Triangulation *manifold, ChernSimonsInfo *
chern_simons_info);
static void      allocate_arrays(Triangulation *manifold, TetShape **save_shapes,
CuspInfo **save_cusp_info);
static void      save_filled_solution(Triangulation *manifold, TetShape *save_shapes,
CuspInfo *save_cusp_info);
static void      restore_filled_solution(Triangulation *manifold, TetShape *save_shapes,
CuspInfo *save_cusp_info);
static void      validate_null_history(Triangulation *manifold);
static void      free_arrays(TetShape *save_shapes, CuspInfo *save_cusp_info);
static void      copy_ultimate_to_penultimate(Triangulation *manifold);
static void      suppress_imaginary_parts(Triangulation *manifold);

```

```

SolutionType find_complete_hyperbolic_structure(
    Triangulation *manifold)
{
    Boolean *is_complete_array;
    double *m_array,
           *l_array;

    /*
     * Set all Tetrahedra to be regular ideal tetrahedra.
     * Allocate the TetShapes if necessary.
     * Clear the shape_histories if necessary.
     */
    initialize_tet_shapes(manifold);

    /*
     * We don't want to destroy any preexisting Dehn filling
     * coefficients, so copy them out to arrays.
     */
    allocate_cusp_status_arrays(manifold, &is_complete_array, &m_array, &l_array);
    record_cusp_status(manifold, is_complete_array, m_array, l_array);
}

```

```

/*
 * Complete all the cusps.
 */
complete_all_cusps(manifold);

/*
 * Call do_Dehn_filling().
 * In general it thinks it's finding a filled hyperbolic structure,
 * but since all the cusps are complete it's really finding the
 * complete hyperbolic structure.
 */
do_Dehn_filling(manifold);

/*
 * Copy the "filled solution" (which is really the complete
 * solution) to where the complete solution belongs.
 */
copy_solution(manifold, filled, complete);

/*
 * Restore the preexisting Dehn filling coefficients.
 */
restore_cusp_status(manifold, is_complete_array, m_array, l_array);
free_cusp_status_arrays(is_complete_array, m_array, l_array);

/*
 * Done.
 */
return manifold->solution_type[complete];
}

void initialize_tet_shapes(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i, j;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        for (i = 0; i < 2; i++) /* i = complete, filled */
        {
            if (tet->shape[i] == NULL)
                tet->shape[i] = NEW_STRUCT(TetShape);

            for (j = 0; j < 3; j++)
                tet->shape[i]->cwl[ultimate][j] = regular_shape;
        }

        clear_shape_history(tet);
    }
}

static void allocate_cusp_status_arrays(
    Triangulation *manifold,
    Boolean **is_complete_array,
    double **m_array,
    double **l_array)
{
    *is_complete_array = NEW_ARRAY(manifold->num_cusps, Boolean);
    *m_array = NEW_ARRAY(manifold->num_cusps, double);
    *l_array = NEW_ARRAY(manifold->num_cusps, double);
}

static void free_cusp_status_arrays(
    Boolean *is_complete_array,
    double *m_array,
    double *l_array)

```

```

{
    my_free(is_complete_array);
    my_free(m_array);
    my_free(l_array);
}

static void record_cusp_status(
    Triangulation *manifold,
    Boolean       is_complete_array[],
    double        m_array[],
    double        l_array[])
{
    Cusp *cusp;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)
    {
        is_complete_array[cusp->index] = cusp->is_complete;
        m_array[cusp->index]           = cusp->m;
        l_array[cusp->index]           = cusp->l;
    }
}

static void restore_cusp_status(
    Triangulation *manifold,
    Boolean       is_complete_array[],
    double        m_array[],
    double        l_array[])
{
    Cusp *cusp;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)
    {
        cusp->is_complete = is_complete_array[cusp->index];
        cusp->m           = m_array[cusp->index];
        cusp->l           = l_array[cusp->index];
    }
}

void complete_all_cusps(
    Triangulation *manifold)
{
    Cusp *cusp;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)
    {
        cusp->is_complete = TRUE;
        cusp->m           = 0.0;
        cusp->l           = 0.0;
    }
}

void copy_solution(
    Triangulation *manifold,
    FillingStatus source, /* complete or filled */
    FillingStatus dest)  /* filled or complete */
{
    copy_tet_shapes(manifold, source, dest);
    copy_cusp_shapes(manifold, source, dest);
    manifold->solution_type[dest] = manifold->solution_type[source];
}

static void copy_tet_shapes(
    Triangulation *manifold,

```

```

    FillingStatus    source,      /* complete or filled */
    FillingStatus    dest)      /* filled or complete */
{
    Tetrahedron *tet;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        *tet->shape[dest] = *tet->shape[source];

        clear_one_shape_history(tet, dest);
        copy_shape_history(tet->shape_history[source], &tet->shape_history[dest]);
    }
}

```

```

static void copy_cusp_shapes(
    Triangulation    *manifold,
    FillingStatus    source,      /* complete/initial or filled/current */
    FillingStatus    dest)      /* filled/current or complete/initial */
{
    Cusp    *cusp;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)
    {
        cusp->cusp_shape[dest] = cusp->cusp_shape[source];
        cusp->shape_precision[dest] = cusp->shape_precision[source];
    }
}

```

```

/*
 * do_Dehn_filling() uses complex gluing equations for oriented
 * manifolds and real gluing equations for unoriented manifolds.
 * To keep the structure of its algorithm as clear as possible,
 * do_Dehn_filling() passes variables for both the complex equations
 * and real equations to the lower level routines, and lets the lower
 * level routines sort out which is the correct one to use for the
 * given manifold.
 */

```

```

SolutionType do_Dehn_filling(
    Triangulation *manifold)
{
    Complex **complex_equations,
             *delta;
    double **real_equations,
            distance_to_solution,
            distance_ratio;
    int    num_rows,
            num_columns,
            iterations,
            result;
    Boolean convergence_is_quadratic,
            solution_was_found,
            iteration_limit_exceeded;

    /*
     * Notify the UI that a potentially long computation is beginning.
     * The user may abort the computation if desired.
     */
    uLongComputationBegins("Computing hyperbolic structure . . .", TRUE);

    /*
     * Check that the Dehn filling coefficients are valid.
     */
    verify_coefficients(manifold);

    /*
     * Number the Tetrahedra. This implicitly assigns each Tetrahedron
     * to one of the complex variables.
     */
}

```

```

    */
    number_the_tetrahedra(manifold);

    /*
    * The following call to compute_holonomies() will rarely be needed,
    * but it guarantees holonomy[penultimate][] will be correct
    * even if Newton's method terminates after only one iteration.
    */
    compute_holonomies(manifold);

    /*
    * allocate_equations() not only allocates the appropriate
    * set of equations, it also associates each equation to an edge
    * or cusp in the manifold. This is why the equations are not
    * explicitly passed to compute_equations().
    */
    allocate_equations( manifold,
                        &complex_equations,
                        &real_equations,
                        &num_rows,
                        &num_columns);

    /*
    * Allocate an array to hold the changes to the Tetrahedron shapes
    * specified by Newton's method.
    */
    delta = NEW_ARRAY(manifold->num_tetrahedra, Complex);

    /*
    * distance_to_solution is initialized to RIGHT_BALLPARK
    * to get the proper behavior the first time through the loop.
    */
    distance_to_solution      = RIGHT_BALLPARK;
    convergence_is_quadratic  = FALSE;
    iterations                = 0;
    iteration_limit_exceeded  = FALSE;

do
{
    choose_coordinate_system(manifold);

    compute_gluing_equations(manifold);

    /*
    * We're done if either
    *
    * (1) the solution has converged, or
    *
    * (2) the solution is degenerate (in which case it
    *     would take a long, long time to converge).
    */
    if
    (   check_convergence(      manifold->orientability,
                                complex_equations,
                                real_equations,
                                num_rows,
                                num_columns,
                                &distance_to_solution,
                                &convergence_is_quadratic,
                                &distance_ratio)

        ||
        (   solution_is_degenerate(manifold)
            && iterations > DEGENERACY_ITERATIONS
            && distance_ratio > DEGENERACY_RATIO
        )
    )
    {
        solution_was_found = TRUE;
        break; /* break out of the do {} while (TRUE) loop */
    }

    /*
    * iterations almost never exceeds ITERATION_LIMIT.
    * In fact, SnapPea was used for years without this check, and

```

```

    * it always found solutions. The first examples where the
    * solutions didn't converge were the meridional Dehn fillings
    * on the nonorientable 6-tetrahedron census manifolds
    * x045, x048, x063, x084 and x175. For further comments,
    * please see the file "failure to solve gluing eqns".
    */
    if (iterations > ITERATION_LIMIT
        && distance_ratio >= 1.0)
    {
        iteration_limit_exceeded = TRUE;
        solution_was_found = FALSE;
        break; /* break out of the do {} while (TRUE) loop */
    }

    result = solve_equations( manifold->orientability,
                              complex_equations,
                              real_equations,
                              num_rows,
                              num_columns,
                              delta);

    if (result == func_cancelled
        || result == func_failed)
    {
        solution_was_found = FALSE;
        break; /* break out of the do {} while (TRUE) loop */
    }

    update_shapes(manifold, delta);

    iterations++;
}
while (TRUE); /* The loop terminates in one of the break statements. */

/*
 * In the rare case that distance_to_solution is exactly zero,
 * copy the ultimate solution to the penultimate one, to indicate
 * that we've solved the equations to full accuracy.
 */
if (distance_to_solution == 0.0)
    copy_ultimate_to_penultimate(manifold);

free_equations(manifold, complex_equations, real_equations, num_rows);
my_free(delta);

if (solution_was_found == TRUE)
    identify_solution_type(manifold);
else if (iteration_limit_exceeded == TRUE)
    manifold->solution_type[filled] = no_solution;
else switch (result)
{
    case func_cancelled:
        manifold->solution_type[filled] = not_attempted;
        break;
    case func_failed:
        manifold->solution_type[filled] = no_solution;
        break;
}

/*
 * 96/1/12 Craig has requested that for flat solutions SnapPea's
 * complex length function provide consistent signs for rotation
 * angles of elliptic isometries (see complex_length.c). I was
 * concerned about distinguishing flat solutions from almost flat
 * solutions, so here we check whether the solution is provably flat,
 * and if so set the imaginary parts of all tet shapes to zero.
 *
 * Proposition. If a solution (to the gluing equations) is
 * almost flat and the Dehn filling coefficients are all integers,
 * then the solution obtained by setting the imaginary parts
 * of all tetrahedron shapes to zero is stable, in the sense that
 * Newton's method would keep all imaginary parts zero.
 *
 * Proof. In Newton's method, both the derivative matrix and the
 * "right hand side" would be real, so the computed array "delta"

```



```

    * would also be real.  QED
    */
    if (manifold->solution_type[filled] == flat_solution
        && all_Deihn_coefficients_are_integers(manifold) == TRUE)
        suppress_imaginary_parts(manifold);

    compute_cusp_shapes(manifold, current);

    compute_CS_value_from_fudge(manifold);

    uLongComputationEnds();

    return manifold->solution_type[filled];
}

/*
 * verify_coefficients() alerts the user and exits if the current set
 * of Dehn filling coefficients includes
 *
 *      (0,0) Dehn filling on any cusp, or
 *
 *      (p,q) Dehn filling, with q != 0, on a nonorientable cusp.
 *
 * set_cusp_info() should have already checked the coefficients
 * for errors, so verify_coefficients() should be unnecessary.  It is
 * included to guard against programming errors (e.g. passing a manifold
 * whose coefficients have not been set at all), not user errors.
 */

static void verify_coefficients(
    Triangulation *manifold)
{
    Cusp      *cusp;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)

        if (
            cusp->is_complete ?
            cusp->m != 0.0 || cusp->l != 0.0 :
            (cusp->m == 0.0 && cusp->l == 0.0) || (cusp->topology == Klein_cusp && cusp->l != 0.0)
        )

            uFatalError("verify_coefficients", "hyperbolic_structure");
}

/*
 * allocate_equations() allocates space for the equations as a matrix,
 * and also associates each equation to an edge or cusp in the manifold.
 */

static void allocate_equations(
    Triangulation *manifold,
    Complex      ***complex_equations,
    double       ***real_equations,
    int          *num_rows,
    int          *num_columns)
{
    if (manifold->orientability == oriented_manifold)
    {
        real_equations = NULL;
        allocate_complex_equations(manifold, complex_equations, num_rows, num_columns);
        associate_complex_eqns_to_edges_and_cusps(manifold, *complex_equations);
    }
    else
    {
        complex_equations = NULL;
        allocate_real_equations(manifold, real_equations, num_rows, num_columns);
        associate_real_eqns_to_edges_and_cusps(manifold, *real_equations);
    }
}

```

```

}

static void free_equations(
    Triangulation *manifold,
    Complex **complex_equations,
    double **real_equations,
    int num_rows)
{
    if (manifold->orientability == oriented_manifold)
        free_complex_equations(complex_equations, num_rows);
    else
        free_real_equations(real_equations, num_rows);

    dissociate_eqns_from_edges_and_cusps(manifold);
}

/*
 * allocate_complex_equations() sets *num_rows and *num_columns,
 * and allocates memory for a complex matrix of dimensions
 * (*num_rows) x (*num_columns + 1). The extra column will
 * hold the constant on the right hand side of the equations.
 */

static void allocate_complex_equations(
    Triangulation *manifold,
    Complex ***complex_equations,
    int *num_rows,
    int *num_columns)
{
    int i;

    /*
     * We'll have an equation for each edge, and also an equation
     * for each cusp. The number of edges in an ideal triangulation
     * equals the number of tetrahedra, by an Euler characteristic
     * argument.
     */

    *num_rows = manifold->num_tetrahedra + manifold->num_cusps;

    /*
     * We'll have one complex variable for each ideal tetrahedron.
     */

    *num_columns = manifold->num_tetrahedra;

    /*
     * The matrix is stored as an array of row pointers.
     */

    *complex_equations = NEW_ARRAY(*num_rows, Complex *);

    for (i = 0; i < *num_rows; i++)
        (*complex_equations)[i] = NEW_ARRAY(*num_columns + 1, Complex);
}

/*
 * allocate_real_equations() sets *num_rows and *num_columns,
 * and allocates memory for a real matrix of dimensions
 * 2*(*num_rows) x 2*(*num_columns + 1). The extra column will
 * hold the constant on the right hand side of the equations.
 */

static void allocate_real_equations(
    Triangulation *manifold,
    double ***real_equations,
    int *num_rows,
    int *num_columns)
{
    int i;

```

```

/*
 * Cf. allocate_complex_equations() above.
 */

*num_rows      = 2 * (manifold->num_tetrahedra + manifold->num_cusps);
*num_columns   = 2 * manifold->num_tetrahedra;

*real_equations = NEW_ARRAY(*num_rows, double *);

for (i = 0; i < *num_rows; i++)
    (*real_equations)[i] = NEW_ARRAY(*num_columns + 1, double);
}

/*
 * free_complex_equations() frees the memory allocated
 * in allocate_complex_equations().
 */

static void free_complex_equations(
    Complex **complex_equations,
    int      num_rows)
{
    int i;

    for (i = 0; i < num_rows; i++)
        my_free(complex_equations[i]);

    my_free(complex_equations);
}

/*
 * free_real_equations() frees the memory allocated
 * in allocate_real_equations().
 */

static void free_real_equations(
    double **real_equations,
    int      num_rows)
{
    int i;

    for (i = 0; i < num_rows; i++)
        my_free(real_equations[i]);

    my_free(real_equations);
}

/*
 * associate_complex_eqns_to_edges_and_cusps() associates the first
 * num_tetrahedra equations to edge classes, and the remaining
 * num_cusps equations to cusps.
 */

static void associate_complex_eqns_to_edges_and_cusps(
    Triangulation *manifold,
    Complex       **complex_equations)
{
    EdgeClass *edge;
    Cusp      *cusp;

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)
    {
        edge->complex_edge_equation = *complex_equations++;
        edge->real_edge_equation_re = NULL;
        edge->real_edge_equation_im = NULL;
    }

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;

```

```

        cusp = cusp->next)
    {
        cusp->complex_cusp_equation = *complex_equations++;
        cusp->real_cusp_equation_re = NULL;
        cusp->real_cusp_equation_im = NULL;
    }
}

/*
 * associate_real_eqns_to_edges_and_cusps() associates the first
 * 2*num_tetrahedra equations to edge classes, and the remaining
 * 2*num_cusps equations to cusps.
 */

static void associate_real_eqns_to_edges_and_cusps(
    Triangulation *manifold,
    double **real_equations)
{
    EdgeClass *edge;
    Cusp *cusp;

    for ( edge = manifold->edge_list_begin.next;
          edge != &manifold->edge_list_end;
          edge = edge->next)
    {
        edge->complex_edge_equation = NULL;
        edge->real_edge_equation_re = *real_equations++;
        edge->real_edge_equation_im = *real_equations++;
    }

    for (cusp = manifold->cusp_list_begin.next;
          cusp != &manifold->cusp_list_end;
          cusp = cusp->next)
    {
        cusp->complex_cusp_equation = NULL;
        cusp->real_cusp_equation_re = *real_equations++;
        cusp->real_cusp_equation_im = *real_equations++;
    }
}

/*
 * dissociate_eqns_from_edges_and_cusps() dissociates the gluing
 * equations from the edges and cusps. Note that this function
 * works for both complex and real equations.
 */

static void dissociate_eqns_from_edges_and_cusps(
    Triangulation *manifold)
{
    EdgeClass *edge;
    Cusp *cusp;

    for ( edge = manifold->edge_list_begin.next;
          edge != &manifold->edge_list_end;
          edge = edge->next)
    {
        edge->complex_edge_equation = NULL;
        edge->real_edge_equation_re = NULL;
        edge->real_edge_equation_im = NULL;
    }

    for (cusp = manifold->cusp_list_begin.next;
          cusp != &manifold->cusp_list_end;
          cusp = cusp->next)
    {
        cusp->complex_cusp_equation = NULL;
        cusp->real_cusp_equation_re = NULL;
        cusp->real_cusp_equation_im = NULL;
    }
}

```

```

/*
 * The shape of an ideal tetrahedron is traditionally parameterized
 * by one of the three forms of its cross ratio. Cross ratios of
 * 0, 1 and infinity represent degenerate tetrahedra. Near these
 * points, bad things happen. The two main problems are that (1) some
 * of the entries in the derivative matrix (used in Newton's method)
 * approach infinity, and (2) incrementing the solution can move it
 * too close to a singularity, resulting in wild swings in the arguments
 * of the cross ratios. Switching the coordinates from the cross
 * ratio to the log of the cross ratio helps a bit. Rather than having
 * two singularities (0 and 1) embedded in the parameter space, you
 * have only one (the singularity which used to be at 1 is now at 0,
 * but the singularity which used to be at 0 has been happily pushed
 * out to infinity).
 *
 * This scheme can be further improved by choosing a (logarithmic)
 * coordinate system based on the current shape of
 * the tetrahedron. The coordinate system is chosen so that the
 * current shape of the tetrahedron stays away from the singularity
 * in the parameter space. Specifically, let
 *
 *      z0 = z
 *
 *      z1 = -----
 *             1 - z
 *
 *      z2 = -----
 *             z
 *
 * and divide the complex plane into three regions:
 *
 *      region A: |z-1| > 1  &&  Re(z) < 1/2
 *      region B: |z| > 1  &&  Re(z) > 1/2
 *      region C: |z-1| < 1  &&  |z| < 1
 *
 * Viewed on the Riemann sphere, the singularities are equally
 * spaced points on the equator, and the regions are separated
 * by meridians spaced 120 degrees apart. The points along the
 * boundaries may be arbitrarily assigned to either neighboring region.
 *
 * In region A, use log(z0) coordinates.
 * In region B, use log(z1) coordinates.
 * In region C, use log(z2) coordinates.
 *
 * Each entry in the derivative matrix used in Newton's method is
 * a linear combination of the derivatives of log(z0), log(z1)
 * and log(z2). The above choice of coordinates implies that each
 * such derivative will have modulus less than or equal to one.
 * Here's the proof. First compute
 *
 *      d(log z0)    1
 *      ----- = -
 *      dz           z
 *
 *      d(log z1)    1
 *      ----- = -----
 *      dz           1 - z
 *
 *      d(log z2)    1
 *      ----- = -----
 *      dz           z(z - 1)
 *
 * Now take ratios of the above to compute
 *
 *      d(log z0)          d(log z0)    1 - z          d(log z0)
 *      ----- = 1          ----- = -----          ----- = z - 1
 *      d(log z0)          d(log z1)    z              d(log z2)
 *
 *      d(log z1)    z          d(log z1)          d(log z1)
 *      ----- = -----          ----- = 1          ----- = -z
 *      d(log z0)    1 - z        d(log z1)          d(log z2)

```

```

*   d(log z2)      1           d(log z2)      -1           d(log z2)
*   ----- = -----          ----- = -----          ----- = 1
*   d(log z0)      z - 1       d(log z1)       z           d(log z2)
*
*   Say z lies in region A, and we have chosen log(z0) coordinates
*   as indicated previously. The derivatives in the first column of the
*   above table have modulus less than or equal to 1. This is obvious
*   for the first entry in the column. For the third entry it's an
*   immediate consequence of the condition  $|1 - z| > 1$ . For the second
*   entry, note that
*
*           | Im(z) | = | Im(1 - z) |
*   and
*           | Re(z) | < | Re(1 - z) | iff Re(z) < 1/2
*
*   hence  $|z| < |1-z|$ .
*
*   Similar arguments show that when z lies in region B (resp. region C)
*   the derivatives in the second column (resp. third column) have
*   modulus less than or equal to 1. (In fact, the derivatives all
*   lie in region C, as can be seen from the fact that the two nonconstant
*   derivatives in each column sum to -1. For our purposes, though, it's
*   enough just to know that the derivatives are bounded, so the entries
*   in the derivative matrix used in Newton's method cannot diverge to
*   infinity.)
*
*   Theoretical note: I briefly entertained the idea of finding a
*   single coordinate system which avoids all three singularities.
*   Picard's Little Theorem shows that this is not possible for an
*   analytic function. It might be possible for a nonanalytic function
*   (perhaps a simple function of z and z-bar?) but I haven't pursued
*   this, and in any case such a function wouldn't be conformal.
*   However, each Tetrahedron's shape_history fields record the topological
*   information such a master coordinate system would contain.
*/

static void choose_coordinate_system(
    Triangulation *manifold)
{
    Tetrahedron *tet;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        if (
            tet->shape[filled]->cwl[ultimate][0].log.real < 0.0          /* |z| < 1 */
            && tet->shape[filled]->cwl[ultimate][1].log.real > 0.0          /* |z-1| < 1 */
        )
        {
            tet->coordinate_system = 2; /* region C, log(z2) coordinates */
        }
        else if (tet->shape[filled]->cwl[ultimate][0].rect.real > 0.5) /* Re(z) < 1/2 */
        {
            tet->coordinate_system = 1; /* region B, log(z1) coordinates */
        }
        else
        {
            tet->coordinate_system = 0; /* region A, log(z0) coordinates */
        }
    }
}

/*
*   check_convergence() checks whether Newton's method has converged to
*   a solution. We check for convergence in the range rather than the
*   domain. In other words, we check how precisely the gluing equations
*   are satisfied, without regard to whether the logs of the tetrahedra's
*   edge parameters are converging. The reason for this is that degenerate
*   equations will be satisfied more and more precisely by edge parameters
*   whose logs are diverging to infinity.
*
*   We know Newton's method has converged when it begins making
*   small random changes. We check this by seeing whether
*/

```

```

* (1) it's in the right ballpark (meaning it should be
*     converging quadratically), and
*
* (2) the new distance is greater than the old one.
*
* We also offer a shortcut, to avoid the possibility of having to
* wait through several essentially random iterations of Newton's
* method which just happen to decrease the distance to the solution
* each time. The shortcut is that we note when quadratic convergence
* begins, and then as soon as it ends we know we've converged.
*
* Finally, if the equations are satisfied perfectly, we return TRUE.
* I realize this is not very likely, but it makes the function
* logically correct. (Without this provision a perfect solution
* would cycle endlessly through Newton's method.)
*
* check_convergence() returns TRUE when it considers Newton's method
* to have converged, and FALSE otherwise.
*/

static Boolean check_convergence(
    Orientability    orientability,
    Complex          **complex_equations,
    double           **real_equations,
    int              num_rows,
    int              num_columns,
    double            *distance_to_solution,
    Boolean           *convergence_is_quadratic,
    double            *distance_ratio)
{
    double            old_distance;

    old_distance = *distance_to_solution;

    *distance_to_solution = orientability == oriented_manifold ?
        compute_distance_complex(complex_equations, num_rows, num_columns) :
        compute_distance_real(real_equations, num_rows, num_columns);

    *distance_ratio = *distance_to_solution / old_distance;

    if (*distance_ratio < QUADRATIC_THRESHOLD)
        *convergence_is_quadratic = TRUE;

    return (
        (*distance_to_solution < RIGHT_BALLPARK && *distance_ratio > 1.0)
        ||
        (*convergence_is_quadratic && *distance_ratio > 0.5)
        ||
        (*distance_to_solution == 0.0) /* seems unlikely, but who knows */
    );
}

static double compute_distance_complex(
    Complex **complex_equations,
    int      num_rows,
    int      num_columns)
{
    double distance_squared;
    int    i;

    distance_squared = 0.0;

    for (i = 0; i < num_rows; i++)
        distance_squared += complex_modulus_squared(complex_equations[i][num_columns]);

    return sqrt(distance_squared); /* no need for safe_sqrt() */
}

static double compute_distance_real(
    double **real_equations,
    int      num_rows,
    int      num_columns)

```

```

{
    double distance_squared;
    int i;

    distance_squared = 0.0;

    for (i = 0; i < num_rows; i++)
        distance_squared += real_equations[i][num_columns] * real_equations[i][num_columns]
    ;

    return sqrt(distance_squared); /* no need for safe_sqrt() */
}

/*
 * In practice a typecast would suffice to convert the real_solution
 * to the Complex solution, since an array of n Complex numbers is stored
 * as an array of 2n reals. But we do an explicit conversion anyhow,
 * in the interest of good style and robust code (and also in the
 * interest of maintaining solve_real_equations() as a general purpose
 * routine for solving real equations).
 */

static FuncResult solve_equations(
    Orientability orientability,
    Complex **complex_equations,
    double **real_equations,
    int num_rows,
    int num_columns,
    Complex *solution)
{
    double *real_solution;
    FuncResult result;

    if (orientability == oriented_manifold)
        result = solve_complex_equations(complex_equations, num_rows, num_columns,
        solution);
    else
    {
        real_solution = NEW_ARRAY(num_columns, double);
        result = solve_real_equations(real_equations, num_rows, num_columns, real_solution)
    ;
        if (result == func_OK)
            convert_solution(real_solution, solution, num_columns);
        my_free(real_solution);
    }

    return result;
}

static void convert_solution(
    double *real_solution,
    Complex *solution,
    int num_columns)
{
    int count;

    for (count = num_columns/2; --count >= 0; )
    {
        solution->real = *real_solution++;
        solution->imag = *real_solution++;
        solution++;
    }
}

void remove_hyperbolic_structures(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i;

    /*

```



```

    * If TetShapes are present, remove them.
    */

    if (manifold->solution_type[complete] != not_attempted)

        for (tet = manifold->tet_list_begin.next;
             tet != &manifold->tet_list_end;
             tet = tet->next)
        {
            for (i = 0; i < 2; i++)          /* i = complete, filled */
            {
                my_free(tet->shape[i]);
                tet->shape[i] = NULL;
            }

            clear_shape_history(tet);
        }

    /*
    * Set solution_type[complete] and solution_type[filled]
    * to not_attempted.
    */

    for (i = 0; i < 2; i++)                  /* i = complete, filled */

        manifold->solution_type[i] = not_attempted;
}

void polish_hyperbolic_structures(
    Triangulation *manifold)
{
    TetShape      *save_shapes;
    CuspInfo      *save_cusp_info;
    ChernSimonsInfo chern_simons_info;

    if (manifold->solution_type[complete] == not_attempted)
        uFatalError("polish_hyperbolic_structures", "polish_hyperbolic_structures");

    save_chern_simons(manifold, &chern_simons_info);
    allocate_arrays(manifold, &save_shapes, &save_cusp_info);
    save_filled_solution(manifold, save_shapes, save_cusp_info);
    complete_all_cusps(manifold);
    copy_tet_shapes(manifold, complete, filled);
    validate_null_history(manifold);
    do_Dehn_filling(manifold);
    copy_solution(manifold, filled, complete);
    restore_filled_solution(manifold, save_shapes, save_cusp_info);
    validate_null_history(manifold);
    do_Dehn_filling(manifold);
    free_arrays(save_shapes, save_cusp_info);
    restore_chern_simons(manifold, &chern_simons_info);
}

static void save_chern_simons(
    Triangulation *manifold,
    ChernSimonsInfo *chern_simons_info)
{
    /*
    * Why do we need to save and restore the Chern-Simons info?
    *
    * polish_hyperbolic_structures() is called just after a
    * Triangulation has been modified (e.g. by basic_simplification()
    * or randomize_triangulation()). At this point the TetShapes are
    * slightly inaccurate, the CS_value is accurate, and the
    * CS_fudge is completely wrong. We don't want to call
    * compute_CS_fudge_from_value() just yet, because then the
    * CS_fudge would inherit the inaccuracies of the TetShapes.
    * But if we call find_complete_hyperbolic_structure() or
    * do_Dehn_filling() right way, they will recompute the CS_value
    * based on the completely wrong CS_fudge. So we save the
    * CS_value until after we've polished the hyperbolic structure,
    * then we restore it and compute the CS_fudge using the accurate
    */
}

```

```

    *   TetShapes.
    */

/*
 *   Record the Chern-Simons data.
 */

chern_simons_info->CS_value_is_known      = manifold->CS_value_is_known;
chern_simons_info->CS_fudge_is_known      = manifold->CS_fudge_is_known;

chern_simons_info->CS_value[ultimate]     = manifold->CS_value[ultimate];
chern_simons_info->CS_value[penultimate]   = manifold->CS_value[penultimate];

chern_simons_info->CS_fudge[ultimate]     = manifold->CS_fudge[ultimate];
chern_simons_info->CS_fudge[penultimate]   = manifold->CS_fudge[penultimate];

/*
 *   Pretend it's no longer there, to save some useless computations.
 */

manifold->CS_value_is_known = FALSE;
manifold->CS_fudge_is_known = FALSE;
}

static void restore_chern_simons(
    Triangulation *manifold,
    ChernSimonsInfo *chern_simons_info)
{
    manifold->CS_value_is_known      = chern_simons_info->CS_value_is_known;
    manifold->CS_fudge_is_known      = chern_simons_info->CS_fudge_is_known;

    manifold->CS_value[ultimate]     = chern_simons_info->CS_value[ultimate];
    manifold->CS_value[penultimate]   = chern_simons_info->CS_value[penultimate];

    manifold->CS_fudge[ultimate]     = chern_simons_info->CS_fudge[ultimate];
    manifold->CS_fudge[penultimate]   = chern_simons_info->CS_fudge[penultimate];

    /*
     *   It might makes sense to call compute_CS_fudge_from_value() at
     *   this point, but in the interest of modularity I decided not to.
     */
}

static void allocate_arrays(
    Triangulation *manifold,
    TetShape **save_shapes,
    CuspInfo **save_cusp_info)
{
    *save_shapes      = NEW_ARRAY(manifold->num_tetrahedra, TetShape);
    *save_cusp_info   = NEW_ARRAY(manifold->num_cusps, CuspInfo);
}

static void save_filled_solution(
    Triangulation *manifold,
    TetShape *save_shapes,
    CuspInfo *save_cusp_info)
{
    int i;
    Tetrahedron *tet;
    Cusp *cusp;

    /*
     *   Save the Tetrahedron shapes.
     */

    for (i = 0, tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         i++, tet = tet->next)

        save_shapes[i] = *tet->shape[filled];
}

```

```

/*
 * Save the Cusp information.
 */

for (i = 0, cusp = manifold->cusp_list_begin.next;
     cusp != &manifold->cusp_list_end;
     i++, cusp = cusp->next)
{
    save_cusp_info[i].is_complete = cusp->is_complete;
    save_cusp_info[i].m          = cusp->m;
    save_cusp_info[i].l          = cusp->l;
}
}

static void restore_filled_solution(
    Triangulation *manifold,
    TetShape      *save_shapes,
    CuspInfo      *save_cusp_info)
{
    int          i;
    Tetrahedron *tet;
    Cusp         *cusp;

    /*
     * Restore the Tetrahedron shapes.
     */

    for (i = 0, tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         i++, tet = tet->next)

        *tet->shape[filled] = save_shapes[i];

    /*
     * Restore the Cusp information.
     */

    for (i = 0, cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         i++, cusp = cusp->next)
    {
        cusp->is_complete = save_cusp_info[i].is_complete;
        cusp->m           = save_cusp_info[i].m;
        cusp->l           = save_cusp_info[i].l;
    }
}

static void validate_null_history(
    Triangulation *manifold)
{
    /*
     * The basic_simplification() and randomize_triangulation()
     * functions can't be guaranteed to find correct arguments
     * for the logarithmic forms of the TetShapes, let alone produce
     * a valid history for each new Tetrahedron it introduces.
     *
     * validate_null_history() enforces a trivial shape_history
     * for each Tetrahedron. It does this by
     *
     * (1) clearing all shape_histories,
     *
     * (2) making sure all Tetrahedra are positively oriented, and
     *
     * (3) making sure all logs lie in the range (0, pi).
     *
     * In the nice case that these conditions are all already met,
     * validate_null_history() doesn't change anything, and
     * polish_hyperbolic_structures() ends up making only small
     * changes to the hyperbolic structures ("polishing" them).
     *
     * If the conditions are not met, validate_null_history() sets
     * the offending Tetrahedron shapes to something acceptable, and

```

```

    * in effect the hyperbolic structure is recomputed from scratch.
    */

Tetrahedron *tet;
int i;

for (tet = manifold->tet_list_begin.next;
     tet != &manifold->tet_list_end;
     tet = tet->next)
{
    clear_one_shape_history(tet, filled);

    /*
     * The algorithm in update_shapes() guarantees that the
     * three shapes (for edge j = 0, 1, 2) will have the same sign
     * for rect.imag, regardless of roundoff errors, so if.
     *
     * Only the ultimate shapes are relevant. The penultimate
     * shapes are ignored.
     */

    for (i = 0; i < 3; i++)
    {
        if (tet->shape[filled]->cwl[ultimate][i].rect.imag <= 0.0)
            tet->shape[filled]->cwl[ultimate][i] = regular_shape;

        tet->shape[filled]->cwl[ultimate][i].log = complex_log(
            tet->shape[filled]->cwl[ultimate][i].rect, PI_OVER_2);
    }
}

static void free_arrays(
    TetShape *save_shapes,
    CuspInfo *save_cusp_info)
{
    my_free(save_shapes);
    my_free(save_cusp_info);
}

static void copy_ultimate_to_penultimate(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i;

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

        for (i = 0; i < 3; i++)

            tet->shape[filled]->cwl[penultimate][i] = tet->shape[filled]->cwl[ultimate][i];
}

static void suppress_imaginary_parts(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i, j;

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

        for (i = 0; i < 2; i++) /* ultimate, penultimate */

            for (j = 0; j < 3; j++)
            {
                tet->shape[filled]->cwl[i][j].rect.imag = 0.0;
            }
}

```

```
        tet->shape[filled]->cwl[i][j].log = complex_log(
            tet->shape[filled]->cwl[i][j].rect,
            tet->shape[filled]->cwl[i][j].log.imag);
    }
}

extern SolutionType remove_Deht_fillings(Triangulation *manifold)
{
    /*
     * Set all cusps to be unfilled.
     */
    complete_all_cusps(manifold);

    /*
     * Copy the complete solution to the "filled" solution.
     */
    copy_solution(manifold, complete, filled);

    /*
     * Call do_Deht_filling(), to insure that all internal
     * data (such as the Chern-Simons invariant) are updated
     * correctly. This invokes an unnecessary computation,
     * but it keeps the code simple, and guarantees that
     * all internal data will be up-to-date.
     */
    return do_Deht_filling(manifold);
}
```