

```

/*
 * normal_surface_construction.c
 *
 * FuncResult find_normal_surfaces(      Triangulation      *manifold,
 *                                     NormalSurfaceList    **surface_list);
 *
 *      tries to find connected, embedded normal surfaces of nonnegative
 *      Euler characteristic.  If spheres or projective planes are found,
 *      then tori and Klein bottles aren't reported, because from the point
 *      of view of the Geometrization Conjecture, one wants to cut along
 *      spheres and projective planes first.  Surfaces are guaranteed
 *      to be connected.  They aren't guaranteed to be incompressible,
 *      although typically they are.  There is no guarantee that all such
 *      normal surfaces will be found.  Returns its result as a
 *      NormalSurfaceList.  The present implementation works only for
 *      cusped manifolds.  Returns func_bad_input for closed manifolds,
 *      or non-manifolds (e.g. for orbifolds or noninteger Dehn fillings).
 *
 * int number_of_normal_surfaces_on_list(NormalSurfaceList *surface_list);
 *
 *      returns the number of normal surfaces contained in the list.
 *
 * Boolean normal_surface_is_orientable(  NormalSurfaceList  *surface_list,
 *                                     int                    index);
 * Boolean normal_surface_is_two_sided(   NormalSurfaceList  *surface_list,
 *                                     int                    index);
 * int normal_surface_Euler_characteristic(NormalSurfaceList *surface_list,
 *                                     int                    index);
 *
 *      return information about a given normal surface on the list.
 *
 * void free_normal_surfaces(NormalSurfaceList *surface_list);
 *
 *      frees an array of NormalSurfaceLists.
 */

/*
 * The Algorithm
 *
 * Normal surfaces consist of a collection of squares and triangles,
 * as described in normal_surfaces.h.  At present we assume the
 * manifold has no filled cusps, although a later version of SnapPea
 * may include an algorithm for the filled case as well.
 *
 * When SnapPea tries to find a hyperbolic structure for a manifold
 * containing an incompressible sphere, projective plane, torus or
 * Klein bottle, the tetrahedra containing the squares of the normal
 * surface description tend to "pinch off" at the square and become
 * degenerate.  That is, the complex edge parameters of the edges
 * parallel to the square tend to one, while the parameters of the
 * remaining edges tend to zero and infinity.  This tells us where to
 * find the squares for the normal surface, and saves us from blindly
 * trying all  $3^{(\text{num tetrahedra})}$  possibilities.  I don't know how to
 * prove that the tetrahedra will always degenerate in this way, but
 * empirically this is what happens.
 *
 * Even though we know the position of the squares, we must still decide
 * how many parallel copies belong in each tetrahedron.  Fortunately
 * we can set up a system of linear equations to do this.  To understand
 * the meaning of the equations, it's helpful to (temporarily!) install
 * an infinite stack of triangles at each ideal vertex, beginning near
 * the fat part of the ideal tetrahedron, and marching off towards
 * the cusp.  With the infinite stack of triangles in place, we
 * can assign any number of squares we want to each tetrahedron without
 * affecting how the surface (squares and triangles together) intersects
 * the faces of the tetrahedra.  [Oh how I wish I could draw pictures
 * to illustrate this.  It is so simple and clear.  But let's push on
 * in ASCII...]  That is, each face of each ideal tetrahedron intersects
 * the surface in three infinite stacks of line segments, one stack
 * going towards each ideal vertex.  So no matter how we assign squares
 * to tetrahedra, the surface will match up across the faces.
 *
 * The question, then, is how does the surface look in the neighborhood
 * of an edge?  By "edge" I mean an edge in the ideal triangulation,

```

\* where the edges of several tetrahedra come together. Consider a  
 \* regular neighborhood of the edge; its boundary is an infinite  
 \* cylinder. Look at the paths the normal surface traces out on the  
 \* cylinder. Where the cylinder intersects a 2-cell of the ideal  
 \* triangulation, the paths may naturally be divided into two sets,  
 \* according to which stack of line segments they pass through (cf. above);  
 \* in other words, according to which ideal vertex they are near.  
 \* Triangles (of the normal surface) define arcs (of paths) which stay  
 \* near the same ideal vertex. Squares define arcs which go from being  
 \* near one ideal vertex to being near the other. If, in going once  
 \* around the cylinder, the total number of square-defined arcs going  
 \* from the "lower" ideal vertex to the "upper" ideal vertex (I'm  
 \* imagining the edge to be vertical) equals the total number going  
 \* from the "upper" to the "lower", then the paths will all be circles;  
 \* otherwise they will be a finite set of helices. The normal surface  
 \* extends nicely across the edge iff the paths are circles.

\* We also want to make sure that the surface is well behaved in the  
 \* neighborhood of each cusp. That is, we want the infinite stacks  
 \* for triangles to piece together to form infinite stacks of  
 \* boundary-parallel tori and Klein bottles, not infinite surfaces  
 \* wrapping around the cusp. The picture to keep in mind is similar  
 \* to the picture for the edge neighborhoods. As you trace a meridian  
 \* or longitude around the cusp, squares will "come up from below" or  
 \* "drop down out of sight". If the total number coming up equals the  
 \* total number going down, then (all but a finite number of) the  
 \* triangles will piece together to form boundary parallel tori and  
 \* Klein bottles. To obtain the final surface, discard all the  
 \* boundary parallel tori and Klein bottles, and keep the  
 \* non-boundary-parallel piece(s) which remain(s). In practice,  
 \* of course, we don't construct infinite stacks of surfaces.  
 \* We construct the squares, and then add a minimal number of triangles  
 \* to extend the squares to the closed surface.

\* The algorithm is as follows. First we use the degenerate hyperbolic  
 \* structure to decide which way to position the squares in each  
 \* tetrahedron. Then we set up and solve a system of linear, integer  
 \* equations to determine how many squares belong in each tetrahedron.  
 \* There is one variable for each tetrahedron, saying how many (parallel)  
 \* squares it contains. There is one equation for each edge, saying  
 \* that the surface passes nicely through the edge without spiraling  
 \* (cf. two paragraph back). There are two equations for each cusp  
 \* (for the meridian and longitude), saying that the surface doesn't  
 \* spiral around the cusp (cf. the preceding paragraph). For example,  
 \* for the square knot these equations are as follows. (A vector  
 \* c0 c1 c2 c3 denotes the equation  $c_0x_0 + c_1x_1 + c_2x_2 + c_3x_3 = 0$ .)

```

*           edge equations
*           0  1 -1  0
*           0 -1  1  0
*           0 -1  1  0
*           0  1 -1  0

```

```

*           cusp equations
*           0  0  0  0
*           0  1  1  0

```

\* These equations may be simplified over the integers to

```

*           0  1  0  0
*           0  0  1  0

```

\* They say that tetrahedron #0 and tetrahedron #3 may have any  
 \* nonnegative number of squares (independently of one another),  
 \* while tetrahedron #1 and tetrahedron #2 may have no squares at all.  
 \* In other words, the equations admit two independent solutions

```

*           (x0 x1 x2 x3) = (1 0 0 0)
*           (x0 x1 x2 x3) = (0 0 0 1)

```

\* Each solution defines a surface (one is a torus following the first  
 \* trefoil summand of the square knot, and the other is a torus following  
 \* the second trefoil summand).

```

* m051()(1) provides a more interesting example. It's equations
* simplify to
*
*          -2  0  0  1  0
*          0  1  0 -1  0
*          0  0  1  0  0
*
* The first three tetrahedra define "dependent variables", whose
* value are completely determined by the values of the variables
* which follow it. In this case
*
*          x0 = x3 / 2
*          x1 = x3
*          x2 = 0
*
* The last two tetrahedra define "independent variables", whose
* values may be chosen freely, subject only to the constraint that
* the values which depend on them be integers. So in this case,
* x3 must be even. By the way, it was accidental that in this example
* the two independent variables came last. The equations could just
* as well have been
*
*          -2  0  1  0  0
*          0  1 -1  0  0
*          0  0  0  1  0
*
* The code in simplify_equations() shows that any set of equations
* may be brought into this form.
*
* Definition. A "nonnegative solution" is one for which all variables
* have nonnegative values.
*
* Definition. The "surface defined by a nonnegative solution" is
* the surface obtained by (1) constructing the specified number of
* squares, (2) constructing an infinite stack of triangles at each
* ideal vertex of each tetrahedron, and (3) removing all boundary
* parallel components of the resulting surface.
*
* Proposition. A surface defined by a nonnegative solution is finite.
*
* Proof. The edge and cusp equations guarantee the existence of
* infinitely many boundary parallel tori and Klein bottles. Q.E.D.
*
* Comment. A surface defined by a nonnegative solution may or may
* not be connected. The code below checks explicitly, and rejects
* nonconnected surfaces.
*/

/*
* Simpler proof???          (This is mainly a note to myself.
*                          Feel free to ignore it.)
*
* There may be a simpler justification of the edge and cusp equations.
* The basic idea is that around each edge (of the manifold's triangulation),
* the number of "upward sloping square" must equal the number of
* "downward sloping squares" if the number of upper and lower edges
* (of squares and triangles) is to balance out. Similar considerations
* apply to cusps. The details appear in the notes for my CAM3DT talk.
* Richard Rannard says this approach is well-known, and called
* "Q normal surface theory" ('Q' stands for "quadrilateral").
* I haven't revised the above documentation, because I still need
* to think through whether it's truly obvious that after choosing
* the number of squares, one can extend to a closed surface with a
* finite number of triangles. (I.e. whether one can justify that
* without falling back on the image of infinite stacks of triangles
* at the ideal vertices.)
*/

/*
* Closed Manifolds
*
* The present algorithm works only for cusped manifolds. Eventually
* it may be possible to extend the algorithm to closed manifolds.
* Detecting the normal surface is no problem: instead of insisting that
* the equations for the meridian and longitude both be satisfied, insist
* only that the equation for the Dehn filling curve be satisfied, and use

```

```

* the equation for a transverse curve to count how many times the normal
* surface intersects the core geodesic. The messy part is checking the
* Euler characteristic, and, worse still, splitting along the surface
* once we've found it.
*/

#include "kernel.h"
#include "normal_surfaces.h"

#define NO_DEFINING_ROW -1

/*
* Due to the quirks of C syntax, we can't say NEW_ARRAY(n, int [4])
* directly, but we can make the following typedef and then say
* NEW_ARRAY(n, ArrayInt4).
*/
typedef int ArrayInt4[4];

static void      create_equations(Triangulation *manifold, int ***equations, int *
    num_equations, int *num_variables);
static void      simplify_equations(int **equations, int num_equations, int num_variables);
static void      find_defining_rows(int **equations, int num_equations, int num_variables,
    int **defining_row);
static int       count_independent_variables(int *defining_row, int num_variables);
static void      solve_equations(int **equations, int num_variables, int *defining_row, int
    index, int *solution);
static Boolean    solution_is_nonnegative(int num_variables, int *solution);
static void      create_squares(Triangulation *manifold, int *solution);
static void      create_triangles(Triangulation *manifold);
static int       count_surface_edges(Tetrahedron *tet, FaceIndex f, VertexIndex v);
static void      copy_normal_surface(Triangulation *manifold, NormalSurface *surface);
static Boolean    contains_positive_Euler_characteristic(NormalSurface *normal_surface_list);
static void      remove_zero_Euler_characteristic(NormalSurface **normal_surface_list, int *
    num_surfaces);
static void      transfer_list_to_array(NormalSurface **temporary_linked_list,
    NormalSurfaceList *permanent_surface_list);
static void      free_equations(int **equations, int num_equations);

FuncResult find_normal_surfaces(
    Triangulation      *manifold,
    NormalSurfaceList  **surface_list)
{
    int                **equations,
        num_equations,
        num_variables,
        *defining_row,
        num_independent_variables,
        loop_stopper,
        index,
        *solution;
    NormalSurface      *normal_surface_list,
        *new_entry;
    Boolean             connected,
        orientable,
        two_sided;
    int                Euler_characteristic;

    /*
    * Allocate and initialize the NormalSurfaceList.
    */
    *surface_list = NEW_STRUCT(NormalSurfaceList);
    (*surface_list)->triangulation      = NULL;
    (*surface_list)->num_normal_surfaces = 0;
    (*surface_list)->list               = NULL;

    /*
    * If the space isn't a manifold, or is a manifold with no cusps,
    * return func_bad_input. (Eventually it may be possible to
    * extend the algorithm to closed manifolds -- see above.)
    */
    if (all_Dehn_coefficients_are_relatively_prime_integers(manifold) == FALSE
        || all_cusps_are_filled(manifold) == TRUE)

```

```

    return func_bad_input;

/*
 * Retriangulate the manifold to removed the filled cusps, if any.
 */
(*surface_list)->triangulation = fill_reasonable_cusps(manifold);
if ((*surface_list)->triangulation == NULL)
    return func_failed;

/*
 * Number the Triangulation's Tetrahedra and EdgeClasses,
 * so they indices may be used to index the rows and columns
 * in the equation matrix.
 */
number_the_tetrahedra((*surface_list)->triangulation);
number_the_edge_classes((*surface_list)->triangulation);

/*
 * Carry out the algorithm described at the top of this file.
 * Create the equations, simplify them, and decide which variables
 * are defined in terms of the others. (If c is the index of a
 * dependent variable, then defining_row[c] is the index of the
 * equation which defines it in terms of the independent variables.
 * If c is the index of an independent variable, then defining_row[c]
 * is set to NO_DEFINING_ROW.)
 */
create_equations((*surface_list)->triangulation, &equations, &num_equations, &
num_variables);
simplify_equations(equations, num_equations, num_variables);
find_defining_rows(equations, num_equations, num_variables, &defining_row);

/*
 * As we find NormalSurfaces, add them to the NULL-terminated
 * singly linked normal_surface_list. Once we know how many
 * there are, we'll transfer them to an array.
 */
normal_surface_list = NULL;

/*
 * How many independent variables are there?
 */
num_independent_variables = count_independent_variables(defining_row, num_variables);

/*
 * We'll examine all solutions (excluding the trivial one) in which
 * each independent variable takes the value 0 or 1. For example,
 * if there are two independent variables, the potential solutions
 * will be parameterized as
 *
 *           0 0    <- exclude as trivial
 *           0 1
 *           1 0
 *           1 1
 *
 * An unsigned int serves well to parameterize such solutions.
 *
 * Eventually, of course, the solutions may have to be scaled
 * to insure that the dependent variables take integer values.
 */

/*
 * It's almost inconceivable we'd have 32 independent variables,
 * but we should check just to be safe.
 */
if (num_independent_variables >= 8 * sizeof(int))
    uFatalError("find_normal_surfaces", "normal_surface_construction");

/*
 * Allocate space for a solution.
 */
solution = NEW_ARRAY(num_variables, int);

/*
 * Loop through the solutions, as explained above.

```

```

    */
    loop_stopper = 1 << num_independent_variables;
    for (index = 1; index < loop_stopper; index++)
    {
        /*
        * Solve the equations to find the number of squares
        * assigned to each Tetrahedron. Find the smallest
        * solution such that independent variable c is positive,
        * and all other independent variables are zero.
        */
        solve_equations(equations,
                        num_variables,
                        defining_row,
                        index,
                        solution);

        /*
        * Ignore solutions in which one or more dependent variables
        * are negative.
        */
        if (solution_is_nonnegative(num_variables, solution) == TRUE)
        {
            /*
            * Construct (in the Tetrahedron data structure itself)
            * the number of squares specified by the solution.
            */
            create_squares((*surface_list)->triangulation, solution);

            /*
            * Construct (in the Tetrahedron data structure itself)
            * the minimal set of triangles required to extend
            * the aforementioned squares to a closed surface.
            * (The fact that the squares satisfy the equations
            * implies that such a set of triangles exists.)
            */
            create_triangles((*surface_list)->triangulation);

            /*
            * What have we got?
            */
            recognize_embedded_surface((*surface_list)->triangulation, &connected, &
orientable, &two_sided, &Euler_characteristic);

            /*
            * Keep only connected surfaces of nonnegative Euler
            * characteristic, because these are the only ones we
            * need to split along.
            */
            if (connected == TRUE && Euler_characteristic >= 0)
            {
                new_entry = NEW_STRUCT(NormalSurface);
                (*surface_list)->num_normal_surfaces++;

                new_entry->is_connected          = connected;
                new_entry->is_orientable          = orientable;
                new_entry->is_two_sided           = two_sided;
                new_entry->Euler_characteristic    = Euler_characteristic;

                copy_normal_surface((*surface_list)->triangulation, new_entry);

                new_entry->next      = normal_surface_list;
                normal_surface_list = new_entry;
            }
        }
    }

    /*
    * If spheres and/or projective planes were found, don't report
    * tori or Klein bottles, since according to the Geometrization
    * Conjecture we should cut along spheres and projective planes first.
    */
    if (contains_positive_Euler_characteristic(normal_surface_list) == TRUE)
        remove_zero_Euler_characteristic(&normal_surface_list, &(*surface_list)->
num_normal_surfaces);

```

```

/*
 * Transfer the NormalSurfaces from the linked list to an array.
 */
transfer_list_to_array(&normal_surface_list, *surface_list);

/*
 * Free local storage.
 */
free_equations(equations, num_equations);
my_free(defining_row);
my_free(solution);

/*
 * All done!
 */
return func_OK;
}

static void create_equations(
    Triangulation *manifold,
    int ***equations,
    int *num_equations,
    int *num_variables)
{
    int i,
        j;
    Tetrahedron *tet;
    ComplexWithLog *z;
    double min_modulus;
    EdgeIndex min_modulus_index;
    int edge_value[6],
        value;
    VertexIndex v;
    FaceIndex initial_side,
        terminal_side;
    PeripheralCurve c;
    Orientation h;

    /*
     * Set up the equations as explained in the documentation at
     * the top of this file.
     */

    /*
     * For now let's allow a square to (potentially) intersect each ideal
     * tetrahedron. Eventually we may want to restrict to degenerate
     * tetrahedra only, to speed up the algorithm. (Actually, it seems
     * plenty fast as it is, and treating all tetrahedra equally keeps
     * the code simple.)
     */

    *num_equations = manifold->num_tetrahedra + 2*manifold->num_cusps;
    *num_variables = manifold->num_tetrahedra;

    *equations = NEW_ARRAY(*num_equations, int *);
    for (i = 0; i < *num_equations; i++)
        (*equations)[i] = NEW_ARRAY(*num_variables, int);

    for (i = 0; i < *num_equations; i++)
        for (j = 0; j < *num_variables; j++)
            (*equations)[i][j] = 0;

    /*
     * If a tetrahedron is degenerate, the complex edge angles will be
     * approaching 0, 1 and infinity. Note which angle is approaching 0.
     * (If a tetrahedron is nondegenerate, then it shouldn't matter which
     * angle is selected, because the corresponding square cross section
     * will be found to have multiplicity zero in the desired surface.)
     */
    for (tet = manifold->tet_list_begin.next, i = 0;
         tet != &manifold->tet_list_end;
         tet = tet->next, i++)

```

```

{
    /*
     * The tetrahedra have already been numbered.
     */
    if (tet->index != i)
        uFatalError("create_equations", "normal_surface_construction");

    z = tet->shape[filled]->cwl[ultimate];

    /*
     * min_modulus_index is the index of the edge whose complex
     * edge parameter is closest to zero. tet->parallel_edge
     * is the index of the edge whose complex edge parameter is
     * closest to one. It's called the "parallel edge" because
     * it's parallel to the square cross section.
     */
    min_modulus_index = 0;
    min_modulus = z[0].log.real;
    for (j = 1; j < 3; j++)
        if (z[j].log.real < min_modulus)
        {
            min_modulus_index = j;
            min_modulus = z[j].log.real;
        }
    tet->parallel_edge = (min_modulus_index + 1) % 3;

    /*
     * The squares may sit in the tetrahedron in one of three positions,
     * according to the value of tet->parallel_edge.
     *
     * parallel_edge = 0    parallel_edge = 1    parallel_edge = 2
     *
     *
     *      0              0              0
     *      /\            /\            /\
     *     /\            /\            /\
     *    3/  5 \4       5/  4 \3       4/  3 \5
     *   /###|###\     /###|###\     /###|###\
     *  /###|###\     /###|###\     /###|###\
     * 3---###|###---2  1---###|###---3  2---###|###---1
     *   \###|###/     \###|###/     \###|###/
     *    1\      /2       2\      /0       0\      /1
     *     \    /         \    /         \    /
     *      1             2             3
     *
     * For each position, it's easy to look at the diagram and
     * see for which edges the square "passes from the lower vertex
     * to the upper vertex", for which edges it does the opposite,
     * and which edges it doesn't intersect at all. For full details,
     * please see the documentation at the top of this file, in
     * particular the discussion of the paths on the cylinder.
     */
    switch (tet->parallel_edge)
    {
        case 0:
            edge_value[0] = edge_value[5] = 0;
            edge_value[1] = edge_value[4] = -1;
            edge_value[2] = edge_value[3] = +1;
            break;

        case 1:
            edge_value[0] = edge_value[5] = +1;
            edge_value[1] = edge_value[4] = 0;
            edge_value[2] = edge_value[3] = -1;
            break;

        case 2:
            edge_value[0] = edge_value[5] = -1;
            edge_value[1] = edge_value[4] = +1;
            edge_value[2] = edge_value[3] = 0;
            break;

        default:
    
```



```

        uFatalError("create_equations", "normal_surface_construction");
    }

    /*
     * Add this tetrahedron's contributions to the edge equations.
     * Note that in a nonorientable manifold, the edge class may
     * see some tetrahedra with reversed orientations.
     */
    for (j = 0; j < 6; j++)
        (*equations)[tet->edge_class[j]->index][i]
            += tet->edge_orientation[j] == right_handed ?
                +edge_value[j] :
                -edge_value[j];

    /*
     * Add this tetrahedron's contributions to the cusp equations.
     */
    for (v = 0; v < 4; v++)
        for (initial_side = 0; initial_side < 4; initial_side++)
        {
            if (initial_side == v)
                continue;

            terminal_side = remaining_face[v][initial_side];

            value = edge_value[edge_between_faces[initial_side][terminal_side]];

            for (c = 0; c < 2; c++) /* c = M, L */
                for (h = 0; h < 2; h++) /* h = right_handed, left_handed */
                    (*equations)[manifold->num_tetrahedra + 2*tet->cusp[v]->index + c]
[i]
                        += value * FLOW(tet->curve[c][h][v][initial_side], tet->curve
[c][h][v][terminal_side]);
        }
    }
}

static void simplify_equations(
    int **equations,
    int num_equations,
    int num_variables)
{
    int r,
        c,
        rr,
        cc,
        mult,
        *temp,
        g;

    r = 0; /* row */
    c = 0; /* column */
    while (r < num_equations && c < num_variables)
    {
        /*
         * Look for a nonzero entry at or below position (r,c).
         */
        for (rr = r; rr < num_equations; rr++)
            if (equations[rr][c] != 0)
                break;

        /*
         * If no nonzero entry is found, move one space to the right
         * and continue.
         */
        if (rr == num_equations)
        {
            c++;
            continue;
        }

        /*
         * Swap rows r and rr, so that the new entry (r,c) is nonzero.

```

```

    */
    temp          = equations[r];
    equations[r]   = equations[rr];
    equations[rr]  = temp;

    /*
    * Do row operations so that
    * (1) entry (r,c) remains nonzero, and
    * (2) all entries below it (i.e. (rr,c) for rr > r) are zero.
    */
    rr = r + 1;
    while (rr < num_equations)
    {
        if (equations[rr][c] != 0)
        {
            mult = equations[rr][c] / equations[r][c];
            for (cc = c; cc < num_variables; cc++)
                equations[rr][cc] -= mult * equations[r][cc];
            if (equations[rr][c] != 0)
            {
                temp          = equations[r];
                equations[r]   = equations[rr];
                equations[rr]  = temp;
            }
            else
                rr++;
        }
        else
            rr++;
    }

    /*
    * Move one space down and one space to the right, and continue.
    */
    r++;
    c++;
}

/*
* Examine each row, starting at the bottom and working
* our way up.
*/
for (r = num_equations; --r >= 0; )
{
    /*
    * Find the first nonzero entry in row r, if any.
    */
    for (c = 0; c < num_variables; c++)
        if (equations[r][c] != 0)
            break;

    /*
    * If no nonzero entry was found, ignore this row.
    */
    if (c == num_variables)
        continue;

    /*
    * Divide this row by the gcd of its entries.
    */
    g = ABS(equations[r][c]);
    for (cc = c + 1; cc < num_variables; cc++)
        g = gcd(g, equations[r][cc]);
    for (cc = c; cc < num_variables; cc++)
        equations[r][cc] /= g;

    /*
    * Clear out all entries in column c, above row r.
    * (The entries below row r are already zero.)
    */
    for (rr = r; --rr >= 0; )
    {
        /*
        * If equations[rr][c] is already zero,

```

```

        * there is no work to be done.
    */
    if (equations[rr][c] == 0)
        continue;

    /*
     * Multiply row rr through by a constant, if necessary,
     * to ensure that equations[r][c] divides equations[rr][c].
     */
    mult = equations[r][c] / gcd(equations[r][c], equations[rr][c]);
    if (mult != 1 && mult != -1)
        for (cc = 0; cc < num_variables; cc++)
            equations[rr][cc] *= mult;

    /*
     * Add a multiple of row r to row rr to create a zero
     * in position (rr,c).
     */
    mult = equations[rr][c] / equations[r][c];
    for (cc = c; cc < num_variables; cc++)
        equations[rr][cc] -= mult * equations[r][cc];
    }
}

static void find_defining_rows(
    int **equations,
    int num_equations,
    int num_variables,
    int **defining_row)
{
    int r,
        c;

    *defining_row = NEW_ARRAY(num_variables, int);

    for (c = 0; c < num_variables; c++)
        (*defining_row)[c] = NO_DEFINING_ROW;

    for (r = 0; r < num_equations; r++)
        for (c = 0; c < num_variables; c++)
            if (equations[r][c] != 0)
            {
                (*defining_row)[c] = r;
                break;
            }
}

static int count_independent_variables(
    int *defining_row,
    int num_variables)
{
    int c,
        count;

    count = 0;

    for (c = 0; c < num_variables; c++)
        if (defining_row[c] == NO_DEFINING_ROW)
            count++;

    return count;
}

static void solve_equations(
    int **equations,
    /* int num_equations, */
    int num_variables,
    int *defining_row,
    int index,
    int *solution) /* space should already be allocated */

```

```

{
    int      r,
            c,
            cc,
            numerator,
            denominator,
            mult;

    /*
     * Find a solution in which the independent variables are
     * or are not zero, as specified by the index (please see
     * find_normal_surfaces() for an explanation of the index).
     * If possible, each nonzero independent variable will be
     * assigned the value 1, but sometimes larger values will
     * be assigned so that the dependent variables are integers.
     *
     * Before trying to understand how the equations are being solved,
     * you might want to review simplify_equations() above to see the
     * form the equations have been put in.  A typical set of equations
     * might look like
     *
     *          -2  0  1  0  1
     *           0  1 -1  0  0
     *           0  0  0  1 -2
     * (but with many more rows and columns, of course).  In this example
     * columns 0, 1 and 3 belong to the dependent variables, while
     * columns 2 and 4 belong to the independent variables.
     */

    /*
     * Assign a value to each variable, starting with the last
     * one and working our way back.
     */
    for (c = num_variables; --c >= 0; )
    {
        /*
         * Is the variable c dependent or independent?
         */
        if (defining_row[c] == NO_DEFINING_ROW)
        {
            /*
             * The variable c is independent.
             * Assign a 1 or a 0, as specified by the index.
             */
            solution[c] = (index & 1);
            index >>= 1;
        }
        else
        {
            /*
             * The variable c is dependent.
             *
             * Use the defining row to deduce the value of the variable c
             * in terms of variables which have already been assigned.
             * If equations[r][c] has absolute value greater than one,
             * it may be necessary to multiply the existing partial
             * solution by some integer > 1 so that the value of the
             * new variable is an integer.  The value of the new variable
             * could be negative; but we'll let the calling routine
             * worry about that.
             */

            r = defining_row[c];

            numerator = 0;
            for (cc = c + 1; cc < num_variables; cc++)
                numerator -= equations[r][cc] * solution[cc];

            denominator = equations[r][c];

            if (numerator % denominator != 0)
            {
                mult = ABS(denominator) / gcd(numerator, denominator);

                for (cc = c + 1; cc < num_variables; cc++)

```

```

        solution[cc] *= mult;

        numerator *= mult;
    }

    solution[c] = numerator / denominator;
}
}

static Boolean solution_is_nonnegative(
    int num_variables,
    int *solution)
{
    int c;

    for (c = 0; c < num_variables; c++)
        if (solution[c] < 0)
            return FALSE;

    return TRUE;
}

static void create_squares(
    Triangulation *manifold,
    int *solution)
{
    Tetrahedron *tet;

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

        tet->num_squares = solution[tet->index];
}

static void create_triangles(
    Triangulation *manifold)
{
    /*
     * The documentation at the top of this file proves that once
     * we have a set of squares satisfying the equations, we may
     * add a finite set of triangles to extend the squares to a
     * closed surface.
     *
     * If we wanted, we could write a mathematically sophisticated
     * algorithm which started at one ideal vertex of one ideal
     * tetrahedron, assumed that vertex had 'n' triangles, and
     * recursively examined neighboring ideal vertices deducing how
     * many triangles they must have (e.g. n+1, n-2, etc.) until it
     * examined all ideal vertices incident to a given cusp, at which
     * point it would choose the smallest value of n which makes the
     * number of triangles nonnegative at all ideal vertices incident
     * to that cusp. It would then repeat the whole procedure for
     * each remaining cusp.
     *
     * Such an algorithm would be a nuisance to code up. Instead we'll
     * use a more simple-minded algorithm. Just keep scanning down
     * the list of tetrahedra, and whenever the number of edges
     * (of squares and triangles combined) on a given face of a given
     * ideal tetrahedron near a given ideal vertex exceeds the number
     * on the face it's glued to, add triangles to make up the difference.
     * (Not only is this simple-minded algorithm easier to code,
     * but for simple manifolds it might be quicker at run time as well.)
     */

    Boolean progress;
    Tetrahedron *tet,
                *nbr;
    FaceIndex f,
                ff;

```

```

    VertexIndex v,
                vv;
    Permutation gluing;
    int         our_edges,
                nbr_edges;

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

        for (v = 0; v < 4; v++)

            tet->num_triangles[v] = 0;

    do
    {
        progress = FALSE;

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

            for (f = 0; f < 4; f++)
            {
                nbr      = tet->neighbor[f];
                gluing    = tet->gluing[f];
                ff        = EVALUATE(tet->gluing[f], f);

                for (v = 0; v < 4; v++)
                {
                    if (f == v)
                        continue;

                    vv = EVALUATE(gluing, v);

                    our_edges = count_surface_edges(tet, f, v);
                    nbr_edges = count_surface_edges(nbr, ff, vv);

                    if (our_edges > nbr_edges)
                    {
                        nbr->num_triangles[vv] += our_edges - nbr_edges;
                        progress = TRUE;
                    }
                    if (nbr_edges > our_edges)
                    {
                        tet->num_triangles[v] += nbr_edges - our_edges;
                        progress = TRUE;
                    }
                }
            }
    } while (progress == TRUE);
}

static int count_surface_edges(
    Tetrahedron *tet,
    FaceIndex    f,
    VertexIndex v)
{
    int num_edge_segments;

    num_edge_segments = 0;

    if (edge3_between_faces[f][v] == tet->parallel_edge)
        num_edge_segments += tet->num_squares;

    num_edge_segments += tet->num_triangles[v];

    return num_edge_segments;
}

static void copy_normal_surface(

```

```

    Triangulation    *manifold,
    NormalSurface    *surface)
{
    Tetrahedron *tet;
    VertexIndex v;

    surface->parallel_edge = NEW_ARRAY(manifold->num_tetrahedra, EdgeIndex);
    surface->num_squares    = NEW_ARRAY(manifold->num_tetrahedra, int);
    surface->num_triangles  = NEW_ARRAY(manifold->num_tetrahedra, ArrayInt4);

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        surface->parallel_edge[tet->index] = tet->parallel_edge;
        surface->num_squares[tet->index]   = tet->num_squares;
        for (v = 0; v < 4; v++)
            surface->num_triangles[tet->index][v] = tet->num_triangles[v];
    }
}

static Boolean contains_positive_Euler_characteristic(
    NormalSurface    *normal_surface_list)
{
    Boolean          positive_value_found;
    NormalSurface    *surface;

    positive_value_found = FALSE;

    for (surface = normal_surface_list; surface != NULL; surface = surface->next)
        if (surface->Euler_characteristic > 0)
            positive_value_found = TRUE;

    return positive_value_found;
}

static void remove_zero_Euler_characteristic(
    NormalSurface    **normal_surface_list,
    int              *num_surfaces)
{
    NormalSurface    **surface_ptr,
                    *dead_surface;

    surface_ptr = normal_surface_list;

    while (*surface_ptr != NULL)
    {
        if ((*surface_ptr)->Euler_characteristic != 0)
            surface_ptr = &(*surface_ptr)->next;
        else
        {
            dead_surface = *surface_ptr;
            *surface_ptr = (*surface_ptr)->next;
            my_free(dead_surface->parallel_edge);
            my_free(dead_surface->num_squares);
            my_free(dead_surface->num_triangles);
            my_free(dead_surface);
            (*num_surfaces)--;
        }
    }
}

static void transfer_list_to_array(
    NormalSurface    **temporary_linked_list,
    NormalSurfaceList *permanent_surface_list)
{
    int              count;
    NormalSurface    *the_surface;

    permanent_surface_list->list = NEW_ARRAY(permanent_surface_list->num_normal_surfaces,
    NormalSurface);

```

```

    count = 0;

    while (*temporary_linked_list != NULL)
    {
        the_surface = *temporary_linked_list;
        *temporary_linked_list = (*temporary_linked_list)->next;

        permanent_surface_list->list[count] = *the_surface;
        permanent_surface_list->list[count].next = NULL;
        count++;

        my_free(the_surface);
    }

    if (count != permanent_surface_list->num_normal_surfaces)
        uFatalError("transfer_list_to_array", "normal_surface_construction");
}

static void free_equations(
    int **equations,
    int num_equations)
{
    int i;

    for (i = 0; i < num_equations; i++)
        my_free(equations[i]);
    my_free(equations);
}

int number_of_normal_surfaces_on_list(
    NormalSurfaceList *surface_list)
{
    return surface_list->num_normal_surfaces;
}

Boolean normal_surface_is_orientable(
    NormalSurfaceList *surface_list,
    int index)
{
    return surface_list->list[index].is_orientable;
}

Boolean normal_surface_is_two_sided(
    NormalSurfaceList *surface_list,
    int index)
{
    return surface_list->list[index].is_two_sided;
}

int normal_surface_Euler_characteristic(
    NormalSurfaceList *surface_list,
    int index)
{
    return surface_list->list[index].Euler_characteristic;
}

void free_normal_surfaces(
    NormalSurfaceList *surface_list)
{
    int i;

    if (surface_list != NULL)
    {
        if (surface_list->triangulation != NULL)
            free_triangulation(surface_list->triangulation);

        for (i = 0; i < surface_list->num_normal_surfaces; i++)

```



```
    {
        my_free(surface_list->list[i].parallel_edge);
        my_free(surface_list->list[i].num_squares);
        my_free(surface_list->list[i].num_triangles);
    }
    if (surface_list->list != NULL)
        my_free(surface_list->list);

    my_free(surface_list);
}
```