

```

/*
 * triangulation.c
 *
 * This file contains the following functions which the kernel
 * provides for the UI:
 *
 *      void      data_to_triangulation(  TriangulationData  *data,
 *      void      triangulation_to_data(  TriangulationData  **manifold_ptr);
 *      void      triangulation_to_data(  TriangulationData  **data_ptr,
 *      void      free_triangulation_data(TriangulationData  *manifold);
 *      void      free_triangulation(    TriangulationData  *data);
 *      void      copy_triangulation(    TriangulationData  *manifold);
 *      void      copy_triangulation(    Triangulation      *source,
 *      void      copy_triangulation(    Triangulation      **destination);
 *
 * Their use is described in SnapPea.h.
 *
 * This file also provides the functions
 *
 *      void      initialize_triangulation(Triangulation *manifold);
 *      void      initialize_tetrahedron(Tetrahedron *tet);
 *      void      initialize_cusp(Cusp *cusp);
 *      void      initialize_edge_class(EdgeClass *edge_class);
 *
 * which kernel functions use to do generic initializations. Much of
 * what they do is not really necessary, but it seems like a good idea
 * to at least write NULL into pointers which have not been set.
 *
 * This file also includes
 *
 *      FuncResult check_Euler_characteristic_of_boundary(Triangulation *manifold);
 *
 * which returns func_OK if the Euler characteristic of the total
 * boundary of the manifold is zero. Otherwise it returns func_failed.
 *
 * Furthermore, this file includes
 *
 *      void      number_the_tetrahedra(Triangulation *manifold);
 *
 * which sets each Tetrahedron's index field equal to its position in
 * the linked list. Indices range from 0 to (num_tetrahedra - 1).
 *
 * In addition, we have
 *
 *      void free_tetrahedron(Tetrahedron *tet);
 *
 * which frees a Tetrahedron and all attached data structures, but does NOT
 * remove the Tetrahedron from any doubly linked list it may be on, and
 *
 *      void clear_shape_history(Tetrahedron *tet);
 *      void copy_shape_history(ShapeInversion *source, ShapeInversion **dest);
 *      void clear_one_shape_history(Tetrahedron *tet, FillingStatus which_history);
 *
 * which do what you'd expect (please see the code for details).
 */

#include "kernel.h"
#include <stdio.h> /* for sprintf() in check_neighbors_and_gluings() */

static void check_neighbors_and_gluings(Triangulation *manifold);
static int count_the_edge_classes(Triangulation *manifold);

void data_to_triangulation(
    TriangulationData  *data,
    Triangulation      **manifold_ptr)
{
    /*
     * We assume the UI has done some basic error checking
     * on the data, so we don't repeat it here.
     */

    Triangulation      *manifold;
    Tetrahedron        **tet_array;

```

```

Cusp          **cusp_array;
Boolean        cusps_are_given;
int            i,
               j,
               k,
               l,
               m;
Boolean        all_peripheral_curves_are_zero,
               finite_vertices_are_present;

/*
 * Initialize *manifold_ptr to NULL.
 * We'll do all our work with manifold, and then copy
 * manifold to *manifold_ptr at the very end.
 */
*manifold_ptr = NULL;

/*
 * Allocate and initialize the Triangulation structure.
 */
manifold = NEW_STRUCT(Triangulation);
initialize_triangulation(manifold);

/*
 * Allocate and copy the name.
 */
manifold->name = NEW_ARRAY(strlen(data->name) + 1, char);
strcpy(manifold->name, data->name);

/*
 * Set up the global information.
 *
 * The hyperbolic structure is included in the file only for
 * human readers; here we recompute it from scratch.
 */
manifold->num_tetrahedra      = data->num_tetrahedra;
manifold->solution_type[complete] = not_attempted;
manifold->solution_type[ filled ] = not_attempted;
manifold->orientability      = data->orientability;
manifold->num_or_cusps       = data->num_or_cusps;
manifold->num_nonor_cusps    = data->num_nonor_cusps;
manifold->num_cusps          = manifold->num_or_cusps
                             + manifold->num_nonor_cusps;

/*
 * Allocate the Tetrahedra.
 * Keep pointers to them on a temporary array, so we can
 * find them by their indices.
 */
tet_array = NEW_ARRAY(manifold->num_tetrahedra, Tetrahedron *);
for (i = 0; i < manifold->num_tetrahedra; i++)
{
    tet_array[i] = NEW_STRUCT(Tetrahedron);
    initialize_tetrahedron(tet_array[i]);
    INSERT_BEFORE(tet_array[i], &manifold->tet_list_end);
}

/*
 * If num_or_cusps or num_nonor_cusps is nonzero, allocate the Cusps.
 * Keep pointers to them on temporary arrays, so we can find them
 * by their indices.
 * Otherwise we will create arbitrary Cusps later.
 */
cusps_are_given = (data->num_or_cusps != 0) || (data->num_nonor_cusps != 0);
if (cusps_are_given == TRUE)
{
    cusp_array = NEW_ARRAY(manifold->num_cusps, Cusp *);
    for (i = 0; i < manifold->num_cusps; i++)
    {
        cusp_array[i] = NEW_STRUCT(Cusp);
        initialize_cusp(cusp_array[i]);
        INSERT_BEFORE(cusp_array[i], &manifold->cusp_list_end);
    }
}

```

```

else
    cusp_array = NULL;

/*
 * Set up the Tetrahedra.
 */

all_peripheral_curves_are_zero = TRUE;
finite_vertices_are_present    = FALSE;

for (i = 0; i < manifold->num_tetrahedra; i++)
{
    for (j = 0; j < 4; j++)
        tet_array[i]->neighbor[j] = tet_array[data->tetrahedron_data[i].neighbor_index
[j]];

    for (j = 0; j < 4; j++)
        tet_array[i]->gluing[j] = CREATE_PERMUTATION(
            0, data->tetrahedron_data[i].gluing[j][0],
            1, data->tetrahedron_data[i].gluing[j][1],
            2, data->tetrahedron_data[i].gluing[j][2],
            3, data->tetrahedron_data[i].gluing[j][3]);

    if (cusps_are_given == TRUE)
    {
        for (j = 0; j < 4; j++)
        {
            if (data->tetrahedron_data[i].cusp_index[j] >= 0)
                /* assign a real cusp */
                tet_array[i]->cusp[j] = cusp_array[data->tetrahedron_data[i].cusp_index
[j]];

            else
            {
                /* mark a finite vertex with NULL */
                tet_array[i]->cusp[j] = NULL;
                finite_vertices_are_present = TRUE;
            }
        }

        for (j = 0; j < 2; j++)
            for (k = 0; k < 2; k++)
                for (l = 0; l < 4; l++)
                    for (m = 0; m < 4; m++)
                    {
                        tet_array[i]->curve[j][k][l][m] = data->tetrahedron_data[i].
curve[j][k][l][m];

                        if (data->tetrahedron_data[i].curve[j][k][l][m] != 0)
                            all_peripheral_curves_are_zero = FALSE;
                    }
    }
}

/*
 * 97/12/8 Check that the neighbors and gluings are consistent.
 * For SnapPea's own files this isn't necessary, but it's a big
 * help for people who write files by hand. It catches the most
 * obvious errors and provides a useful diagnosis (as opposed to,
 * say, having the program hang when inconsistent neighbors and/or
 * gluings send create_edge_classes() into an infinite loop).
 * Even in the typical case of reading SnapPea's own files,
 * check_neighbors_and_gluings() is very quick.
 */
check_neighbors_and_gluings(manifold);

/*
 * Set up the EdgeClasses.
 */
create_edge_classes(manifold);
orient_edge_classes(manifold);

/*
 * If the Cusps were specified explicitly, copy in the data.
 * Otherwise create arbitrary Cusps now.

```

```

    */
    if (cusps_are_given == TRUE)
    {
        for (i = 0; i < manifold->num_cusps; i++)
        {
            cusp_array[i]->topology      = data->cusp_data[i].topology;
            cusp_array[i]->m             = data->cusp_data[i].m;
            cusp_array[i]->l             = data->cusp_data[i].l;
            cusp_array[i]->is_complete  = (data->cusp_data[i].m == 0.0
            && data->cusp_data[i].l == 0.0);

            cusp_array[i]->index        = i;
        }

        /*
        * If finite vertices are present they will be marked with NULL.
        * Assign Cusp structures.
        */
        if (finite_vertices_are_present == TRUE)
            create_fake_cusps(manifold);
    }
else
    {
        create_cusps(manifold);
        finite_vertices_are_present = mark_fake_cusps(manifold);
    }

    /*
    * Provide peripheral curves if necessary.
    * This automatically records the CuspTopologies.
    * (Note: all_peripheral_curves_are_zero is TRUE whenever
    * cusps_are_given is FALSE.)
    */
    if (all_peripheral_curves_are_zero == TRUE)
        peripheral_curves(manifold);

    /*
    * If the given triangulation includes finite vertices, remove them.
    */
    if (finite_vertices_are_present == TRUE)
        remove_finite_vertices(manifold);

    /*
    * Count the Cusps if necessary, noting how many have each topology.
    */
    if (cusps_are_given == FALSE)
        count_cusps(manifold);

    /*
    * Free the temporary arrays.
    */
    my_free(tet_array);
    if (cusp_array != NULL)
        my_free(cusp_array);

    /*
    * Typically the manifold's orientability will already be known,
    * but if it isn't, try to orient it now.
    */
    if (manifold->orientability == unknown_orientability)
    {
        orient(manifold);
        if (manifold->orientability == oriented_manifold)
        {
            if (all_peripheral_curves_are_zero == FALSE)
                uAcknowledge("Meridians may be reversed to insure right-handed {M,L} pairs.❧");
            fix_peripheral_orientations(manifold);
        }
    }

    /*
    * Compute the complete and filled hyperbolic structures.
    *
    * (The Dehn fillings should be nontrivial only if the data

```

```

    * provided the peripheral curves.)
    */
    find_complete_hyperbolic_structure(manifold);
    do_Deihn_filling(manifold);

    /*
    * If we provided the basis and the manifold is hyperbolic,
    * replace it with a shortest basis.
    */
    if (all_peripheral_curves_are_zero == TRUE
        && ( manifold->solution_type[complete] == geometric_solution
            || manifold->solution_type[complete] == nongeometric_solution))
        install_shortest_bases(manifold);

    /*
    * If the Chern-Simons invariant is present, compute the fudge factor.
    * Then recompute the value from the fudge factor, to restore the
    * uncertainty between the ultimate and penultimate values.
    */
    manifold->CS_value_is_known      = data->CS_value_is_known;
    manifold->CS_value[ultimate]     = data->CS_value;
    manifold->CS_value[penultimate]  = data->CS_value;
    compute_CS_fudge_from_value(manifold);
    compute_CS_value_from_fudge(manifold);

    /*
    * Done.
    */
    *manifold_ptr = manifold;
}

static void check_neighbors_and_gluings(
    Triangulation *manifold)
{
    Tetrahedron *tet,
                *nbr;
    FaceIndex    f,
                nbr_f;
    Permutation  this_gluing;
    char         scratch[256];

    number_the_tetrahedra(manifold);

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

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

            if (nbr->neighbor[nbr_f] != tet)
            {
                sprintf(scratch, "inconsistent neighbor data, tet %d face %d to tet %d face %d",
                        tet->index, f, nbr->index, nbr_f);
                uAcknowledge(scratch);
                uFatalError("check_neighbors_and_gluings", "triangulations");
            }

            if (nbr->gluing[nbr_f] != inverse_permutation[this_gluing])
            {
                sprintf(scratch, "inconsistent gluing data, tet %d face %d to tet %d face %d",
                        tet->index, f, nbr->index, nbr_f);
                uAcknowledge(scratch);
                uFatalError("check_neighbors_and_gluings", "triangulations");
            }
        }
    }
}

```

```

void triangulation_to_data(
    Triangulation      *manifold,
    TriangulationData  **data_ptr)
{
    /*
     * Allocate the TriangulationData and write in the data describing
     * the manifold. Set *data_ptr to point to the result. The UI
     * should call free_triangulation_data() when it's done with the
     * TriangulationData.
     */

    TriangulationData  *data;
    Cusp               *cusp;
    Tetrahedron        *tet;
    int                i,
                      j,
                      k,
                      l,
                      m;

    *data_ptr = NULL;

    data = NEW_STRUCT(TriangulationData);

    if (manifold->name != NULL)
    {
        data->name = NEW_ARRAY(strlen(manifold->name) + 1, char);
        strcpy(data->name, manifold->name);
    }
    else
        data->name = NULL;

    data->num_tetrahedra = manifold->num_tetrahedra;
    data->solution_type = manifold->solution_type[filled];
    data->volume = volume(manifold, NULL);
    data->orientability = manifold->orientability;
    data->CS_value_is_known = manifold->CS_value_is_known;
    data->num_or_cusps = manifold->num_or_cusps;
    data->num_nonor_cusps = manifold->num_nonor_cusps;
    if (manifold->CS_value_is_known == TRUE)
        data->CS_value = manifold->CS_value[ultimate];

    data->cusp_data = NEW_ARRAY(manifold->num_cusps, CuspData);
    for (i = 0; i < manifold->num_cusps; i++)
    {
        cusp = find_cusp(manifold, i);

        data->cusp_data[i].topology = cusp->topology;
        data->cusp_data[i].m = cusp->m;
        data->cusp_data[i].l = cusp->l;
    }

    number_the_tetrahedra(manifold);
    data->tetrahedron_data = NEW_ARRAY(manifold->num_tetrahedra, TetrahedronData);
    for (tet = manifold->tet_list_begin.next, i = 0;
        tet != &manifold->tet_list_end;
        tet = tet->next, i++)
    {
        for (j = 0; j < 4; j++)
            data->tetrahedron_data[i].neighbor_index[j] = tet->neighbor[j]->index;

        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                data->tetrahedron_data[i].gluing[j][k] = EVALUATE(tet->gluing[j], k);

        /*
         * All negative cusp indices (for finite vertices) map to -1.
         * This could be changed if desired, but at the moment it's
         * consistent with TriangulationFileFormat.
         */
        for (j = 0; j < 4; j++)
            data->tetrahedron_data[i].cusp_index[j] =
                ((tet->cusp[j]->index >= 0) ? tet->cusp[j]->index : -1);
    }
}

```

```

        for (j = 0; j < 2; j++)
            for (k = 0; k < 2; k++)
                for (l = 0; l < 4; l++)
                    for (m = 0; m < 4; m++)
                        data->tetrahedron_data[i].curve[j][k][l][m] = tet->curve[j][k][l]
[m];

        data->tetrahedron_data[i].filled_shape =
            (tet->shape[filled] != NULL) ?
            tet->shape[filled]->cwl[ultimate][0].rect :
            Zero;
    }

    *data_ptr = data;
}

void free_triangulation_data(
    TriangulationData *data)
{
    /*
     * If the kernel allocates a TriangulationData structure,
     * the kernel must free it.
     */
    if (data != NULL)
    {
        if (data->name != NULL)
            my_free(data->name);

        if (data->cusp_data != NULL)
            my_free(data->cusp_data);

        if (data->tetrahedron_data != NULL)
            my_free(data->tetrahedron_data);

        my_free(data);
    }
}

void free_triangulation(
    Triangulation *manifold)
{
    Tetrahedron *dead_tet;
    EdgeClass *dead_edge;
    Cusp *dead_cusp;

    if (manifold != NULL)
    {
        if (manifold->name != NULL)
            my_free(manifold->name);

        while (manifold->tet_list_begin.next != &manifold->tet_list_end)
        {
            dead_tet = manifold->tet_list_begin.next;
            REMOVE_NODE(dead_tet);
            free_tetrahedron(dead_tet);
        }

        while (manifold->edge_list_begin.next != &manifold->edge_list_end)
        {
            dead_edge = manifold->edge_list_begin.next;
            REMOVE_NODE(dead_edge);
            my_free(dead_edge);
        }

        while (manifold->cusp_list_begin.next != &manifold->cusp_list_end)
        {
            dead_cusp = manifold->cusp_list_begin.next;
            REMOVE_NODE(dead_cusp);
            my_free(dead_cusp);
        }
    }
}

```

```

        my_free(manifold);
    }
}

void free_tetrahedron(
    Tetrahedron *tet)
{
    /*
     * This function does NOT remove the Tetrahedron from
     * any doubly linked list it may be on, but does remove
     * all data structures attached to the Tetrahedron.
     */

    int i;

    for (i = 0; i < 2; i++) /* i = complete, filled */
        if (tet->shape[i] != NULL)
            my_free(tet->shape[i]);

    clear_shape_history(tet);

    if (tet->cross_section != NULL)
        my_free(tet->cross_section);

    if (tet->canonize_info != NULL)
        my_free(tet->canonize_info);

    if (tet->cusp_nbhd_position != NULL)
        my_free(tet->cusp_nbhd_position);

    if (tet->extra != NULL)
        my_free(tet->extra);

    my_free(tet);
}

void clear_shape_history(
    Tetrahedron *tet)
{
    int i;

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

        clear_one_shape_history(tet, i);
}

void clear_one_shape_history(
    Tetrahedron *tet,
    FillingStatus which_history) /* filled or complete */
{
    ShapeInversion *dead_shape_inversion;

    while (tet->shape_history[which_history] != NULL)
    {
        dead_shape_inversion = tet->shape_history[which_history];
        tet->shape_history[which_history] = tet->shape_history[which_history]->next;
        my_free(dead_shape_inversion);
    }
}

void copy_triangulation(
    Triangulation *source,
    Triangulation **destination_ptr)
{
    Triangulation *destination;
    Tetrahedron **new_tet;
    EdgeClass **new_edge;
    Cusp **new_cusp;
    Tetrahedron *tet;
    EdgeClass *edge;

```



```

Cusp          *cusp;
int           num_edge_classes,
             min_cusp_index,
             max_cusp_index,
             num_potential_cusps,
             i,
             j;

/*
 * Allocate space for the new Triangulation.
 */

*destination_ptr = NEW_STRUCT(Triangulation);

/*
 * Give it a local name.
 */

destination = *destination_ptr;

/*
 * Copy the global information.
 * In a moment we'll overwrite the fields involving pointers.
 */

*destination = *source;

/*
 * Allocate space for the name, and copy it it.
 */

destination->name = NEW_ARRAY(strlen(source->name) + 1, char);
strcpy(destination->name, source->name);

/*
 * Initialize the doubly linked lists.
 */

destination->tet_list_begin.prev = NULL;
destination->tet_list_begin.next = &destination->tet_list_end;
destination->tet_list_end.prev = &destination->tet_list_begin;
destination->tet_list_end.next = NULL;

destination->edge_list_begin.prev = NULL;
destination->edge_list_begin.next = &destination->edge_list_end;
destination->edge_list_end.prev = &destination->edge_list_begin;
destination->edge_list_end.next = NULL;

destination->cusp_list_begin.prev = NULL;
destination->cusp_list_begin.next = &destination->cusp_list_end;
destination->cusp_list_end.prev = &destination->cusp_list_begin;
destination->cusp_list_end.next = NULL;

/*
 * Assign consecutive indices to source's Tetrahedra and EdgeClasses.
 * The Cusps will already be numbered.
 *
 * While we're at it, count the EdgeClasses.
 * If no finite vertices are present, the number of EdgeClasses
 * will equal the number of Tetrahedra.
 */

number_the_tetrahedra(source);
number_the_edge_classes(source);
num_edge_classes = count_the_edge_classes(source);

/*
 * Find the largest and smallest Cusp indices.
 */
min_cusp_index = source->cusp_list_begin.next->index;
max_cusp_index = source->cusp_list_begin.next->index;
for ( cusp = source->cusp_list_begin.next;
      cusp != &source->cusp_list_end;
      cusp = cusp->next)

```

```

{
    if (cusp->index < min_cusp_index)
        min_cusp_index = cusp->index;
    if (cusp->index > max_cusp_index)
        max_cusp_index = cusp->index;
}
num_potential_cusps = max_cusp_index - min_cusp_index + 1;

/*
 * Allocate the new Tetrahedra, EdgeClasses and Cusps.
 * For the Cusps we want to allow for the possibility
 * that there'll be gaps in the indexing scheme.
 */

new_tet = NEW_ARRAY(source->num_tetrahedra, Tetrahedron *);
for (i = 0; i < source->num_tetrahedra; i++)
    new_tet[i] = NEW_STRUCT(Tetrahedron);

new_edge = NEW_ARRAY(num_edge_classes, EdgeClass *);
for (i = 0; i < num_edge_classes; i++)
    new_edge[i] = NEW_STRUCT(EdgeClass);

new_cusp = NEW_ARRAY(num_potential_cusps, Cusp *);
for (i = 0; i < num_potential_cusps; i++)
    new_cusp[i] = NULL;
for (cusp = source->cusp_list_begin.next;
     cusp != &source->cusp_list_end;
     cusp = cusp->next)
    new_cusp[cusp->index - min_cusp_index] = NEW_STRUCT(Cusp);

/*
 * Copy the fields of each Tetrahedron in source
 * to the corresponding fields in new_tet[i].
 */

for (tet = source->tet_list_begin.next, i = 0;
     tet != &source->tet_list_end;
     tet = tet->next, i++)
{
    /*
     * Copy all fields,
     * then overwrite the ones involving pointers.
     */

    *new_tet[i] = *tet;

    for (j = 0; j < 4; j++)
    {
        new_tet[i]->neighbor[j] = new_tet[tet->neighbor[j]->index];
        new_tet[i]->gluing[j] = tet->gluing[j];
        new_tet[i]->cusp[j] = new_cusp[tet->cusp[j]->index - min_cusp_index];
    }

    for (j = 0; j < 6; j++)
        new_tet[i]->edge_class[j] = new_edge[tet->edge_class[j]->index];

    for (j = 0; j < 2; j++) /* j = complete, filled */
        if (tet->shape[j] != NULL)
        {
            new_tet[i]->shape[j] = NEW_STRUCT(TetShape);
            *new_tet[i]->shape[j] = *tet->shape[j];
        }

    for (j = 0; j < 2; j++) /* j = complete, filled */
        copy_shape_history(tet->shape_history[j], &new_tet[i]->shape_history[j]);

    if (tet->cusp_nbhd_position != NULL)
    {
        new_tet[i]->cusp_nbhd_position = NEW_STRUCT(CuspNbhdPosition);
        *new_tet[i]->cusp_nbhd_position = *tet->cusp_nbhd_position;
    }

    /*
     * Just to be safe.

```

```

    */
    new_tet[i]->cross_section = NULL;
    new_tet[i]->canonize_info = NULL;
    new_tet[i]->extra = NULL;

    INSERT_BEFORE(new_tet[i], &destination->tet_list_end);
}

/*
 * Copy the fields of each EdgeClass in source
 * to the corresponding fields in new_edge[i].
 */

for (edge = source->edge_list_begin.next, i = 0;
     edge != &source->edge_list_end;
     edge = edge->next, i++)
{
    /*
     * Copy all fields,
     * then overwrite the ones involving pointers.
     */

    *new_edge[i] = *edge;

    new_edge[i]->incident_tet = new_tet[edge->incident_tet->index];

    INSERT_BEFORE(new_edge[i], &destination->edge_list_end);
}

/*
 * Copy the fields of each Cusp in source
 * to the corresponding fields in new_cusp[i].
 */

for (cusp = source->cusp_list_begin.next;
     cusp != &source->cusp_list_end;
     cusp = cusp->next)
{
    /*
     * Copy all fields,
     * then overwrite the ones involving pointers.
     */

    *new_cusp[cusp->index - min_cusp_index] = *cusp;

    INSERT_BEFORE(new_cusp[cusp->index - min_cusp_index], &destination->cusp_list_end);
}

/*
 * Free the arrays of pointers.
 */

my_free(new_tet);
my_free(new_edge);
my_free(new_cusp);
}

void copy_shape_history(
    ShapeInversion *source,
    ShapeInversion **dest)
{
    while (source != NULL)
    {
        *dest = NEW_STRUCT(ShapeInversion);

        (*dest)->wide_angle = source->wide_angle;

        source = source->next;
        dest = &(*dest)->next;
    }

    *dest = NULL;
}

```

```

void initialize_triangulation(
    Triangulation *manifold)
{
    manifold->name = NULL;
    manifold->num_tetrahedra = 0;
    manifold->solution_type[complete] = not_attempted;
    manifold->solution_type[filled] = not_attempted;
    manifold->orientability = unknown_orientability;
    manifold->num_cusps = 0;
    manifold->num_or_cusps = 0;
    manifold->num_nonor_cusps = 0;
    manifold->num_generators = 0;
    manifold->CS_value_is_known = FALSE;
    manifold->CS_fudge_is_known = FALSE;
    manifold->CS_value[ultimate] = 0.0;
    manifold->CS_value[penultimate] = 0.0;
    manifold->CS_fudge[ultimate] = 0.0;
    manifold->CS_fudge[penultimate] = 0.0;

    initialize_tetrahedron(&manifold->tet_list_begin);
    initialize_tetrahedron(&manifold->tet_list_end);

    manifold->tet_list_begin.prev = NULL;
    manifold->tet_list_begin.next = &manifold->tet_list_end;
    manifold->tet_list_end.prev = &manifold->tet_list_begin;
    manifold->tet_list_end.next = NULL;

    initialize_edge_class(&manifold->edge_list_begin);
    initialize_edge_class(&manifold->edge_list_end);

    manifold->edge_list_begin.prev = NULL;
    manifold->edge_list_begin.next = &manifold->edge_list_end;
    manifold->edge_list_end.prev = &manifold->edge_list_begin;
    manifold->edge_list_end.next = NULL;

    initialize_cusp(&manifold->cusp_list_begin);
    initialize_cusp(&manifold->cusp_list_end);

    manifold->cusp_list_begin.prev = NULL;
    manifold->cusp_list_begin.next = &manifold->cusp_list_end;
    manifold->cusp_list_end.prev = &manifold->cusp_list_begin;
    manifold->cusp_list_end.next = NULL;
}

void initialize_tetrahedron(
    Tetrahedron *tet)
{
    int h,
        i,
        j,
        k;

    for (i = 0; i < 4; i++)
    {
        tet->neighbor[i] = NULL;
        tet->gluing[i] = 0;
        tet->cusp[i] = NULL;
        tet->generator_status[i] = unassigned_generator;
        tet->generator_index[i] = -1;
        tet->generator_parity[i] = -1;
        tet->corner[i] = Zero;
        tet->tilt[i] = -1.0e17;
    }

    for (h = 0; h < 2; h++)
        for (i = 0; i < 2; i++)
            for (j = 0; j < 4; j++)
                for (k = 0; k < 4; k++)
                    tet->curve[h][i][j][k] = 0;

    for (i = 0; i < 6; i++)

```

```

{
    tet->edge_class[i]      = NULL;
    tet->edge_orientation[i] = -1;
}

for (i = 0; i < 2; i++)
{
    tet->shape[i]      = NULL;
    tet->shape_history[i] = NULL;
}

tet->generator_path      = -2;
tet->cross_section       = NULL;
tet->canonize_info        = NULL;
tet->cusp_nbhd_position  = NULL;
tet->extra                = NULL;
tet->prev                = NULL;
tet->next                = NULL;
}

void initialize_cusp(
    Cusp      *cusp)
{
    cusp->topology          = unknown_topology;
    cusp->is_complete       = TRUE;
    cusp->m                 = 0.0;
    cusp->l                 = 0.0;
    cusp->holonomy[ultimate][M] = Zero;
    cusp->holonomy[ultimate][L] = Zero;
    cusp->holonomy[penultimate][M] = Zero;
    cusp->holonomy[penultimate][L] = Zero;
    cusp->complex_cusp_equation = NULL;
    cusp->real_cusp_equation_re = NULL;
    cusp->real_cusp_equation_im = NULL;
    cusp->cusp_shape[initial]    = Zero;
    cusp->cusp_shape[current]    = Zero;
    cusp->shape_precision[initial] = 0;
    cusp->shape_precision[current] = 0;
    cusp->index                  = 255;
    cusp->displacement            = 0.0;
    cusp->displacement_exp        = 1.0;
    cusp->is_finite               = FALSE;
    cusp->matching_cusp          = NULL;
    cusp->prev                   = NULL;
    cusp->next                   = NULL;
}

void initialize_edge_class(
    EdgeClass *edge_class)
{
    edge_class->order          = 0;
    edge_class->incident_tet    = NULL;
    edge_class->incident_edge_index = -1;
    edge_class->num_incident_generators = -1;
    edge_class->complex_edge_equation = NULL;
    edge_class->real_edge_equation_re = NULL;
    edge_class->real_edge_equation_im = NULL;
    edge_class->prev            = NULL;
    edge_class->next            = NULL;
}

FuncResult check_Euler_characteristic_of_boundary(
    Triangulation *manifold)
{
    int      num_edges;
    EdgeClass *edge;

    /*
     * check_Euler_characteristic_of_boundary() returns
     * func_OK if the Euler characteristic of the total
     * boundary is zero, and returns func_failed otherwise.
     */
}

```

```

* Note that (so far) all functions which call
* check_Euler_characteristic_of_boundary() are testing
* whether curves on the (intended) boundary components
* have been pinched off. In these cases the Euler
* characteristic of the affected boundary component
* will increase, but never decrease. So knowing that
* the Euler characteristic of the total boundary is
* zero implies that the Euler characteristic of each
* component is also zero.
*
* Checking the Euler characteristic of the boundary
* is trivially easy -- you just check whether the
* number of EdgeClasses in the manifold equals the
* number of Tetrahedra. Here's the proof:
*
* Let v, e and f be the number of vertices, edges, and
* faces in the triangulation of the boundary.
* Let E, F and T be the number of edges, faces and
* tetrahedra in the ideal triangulation of the
* manifold.
*
*  $v = 2E$ 
*  $e = 3F = 6T$ 
*  $f = 4T$ 
*
*  $v - e + f = 2E - 6T + 4T = 2(E - T)$ 
*
* So the boundary topology will be correct iff  $E == T$ .
*/

/*
* Count the number of edge classes.
*/

num_edges = 0;

for (edge = manifold->edge_list_begin.next;
     edge != &manifold->edge_list_end;
     edge = edge->next)

    num_edges++;

/*
* Compare the number of edges to the number of tetrahedra.
*/

if (num_edges != manifold->num_tetrahedra)
    return func_failed;
else
    return func_OK;
}

/*
* number_the_tetrahedra() fills in the index field of the Tetrahedra
* according to their order in the manifold's doubly-linked list.
* Indices range from 0 to (num_tetrahedra - 1).
*/

void number_the_tetrahedra(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int count;

    count = 0;

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

        tet->index = count++;
}

```

```
/*
 * number_the_edge_classes() fills in the index field of the EdgeClasses
 * according to their order in the manifold's doubly-linked list.
 * Indices range from 0 to (number of EdgeClasses) - 1.
 */
```

```
void number_the_edge_classes(
    Triangulation *manifold)
{
    EdgeClass *edge;
    int count;

    count = 0;

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)

        edge->index = count++;
}
```

```
static int count_the_edge_classes(
    Triangulation *manifold)
{
    EdgeClass *edge;
    int count;

    count = 0;

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)

        count++;

    return count;
}
```

```
/*
 * compose_permutations() returns the composition of two permutations.
 * Permutations are composed right-to-left: the composition p1 o p0
 * is what you get by first doing p0, then p1.
 */
```

```
Permutation compose_permutations(
    Permutation p1,
    Permutation p0)
{
    Permutation result;
    int i;

    result = 0;

    for (i = 4; --i >= 0; )
    {
        result <= 2;
        result += EVALUATE(p1, EVALUATE(p0, i));
    }

    return result;
}
```