

```

/*
 * terse_triangulation.c
 *
 * This file provides the functions
 *
 *     TerseTriangulation *tri_to_terse(Triangulation *manifold);
 *     Triangulation      *terse_to_tri(TerseTriangulation *tt);
 *     void                free_terse_triangulation(TerseTriangulation *tt);
 *
 * tri_to_terse() accepts a pointer to a Triangulation, computes the
 *     corresponding TerseTriangulation, and returns a pointer to it.
 *
 * terse_to_tri() accepts a pointer to a TerseTriangulation, expands it
 *     to a full Triangulation, and returns a pointer to it.
 *
 * free_terse_triangulation() releases the memory used to store a
 *     TerseTriangulation.
 */

#include "kernel.h"

#define DAFAULT_NAME      "unknown"

/*
 * If you are not familiar with SnapPea's "Extra" field in
 * the Tetrahedron data structure, please see the explanation
 * preceding the Extra typedef in kernel_typedefs.h.
 *
 * tri_to_terse() attaches an Extra field to each old Tetrahedron
 * to keep track of the Tetrahedron's role in the TerseTriangulation.
 */

struct extra
{
    /*
     * Has this Tetrahedron been incorporated in the TerseTriangulation?
     */
    Boolean    in_use;

    /*
     * The remaining fields will be defined iff in_use == TRUE.
     */

    /*
     * What is this Tetrahedron's index in the TerseTriangulation?
     */
    int        index;

    /*
     * The Permutation convert_tri_to_terse_indices takes a VertexIndex
     * or FaceIndex in the Triangulation data structure and "returns"
     * the TerseTriangulation's index for that same vertex or face.
     * That is,
     *
     *     (index in TerseTriangulation)
     *       = EVALUATE(convert_tri_to_terse_indices, index in Triangulation);
     *
     * The Permutation convert_terse_to_tri_indices does just the opposite.
     */
    Permutation convert_tri_to_terse_indices,
                convert_terse_to_tri_indices;

    /*
     * Which of the four faces have been glued in the TerseTriangulation?
     * The indices are those of the TerseTriangulation, not the
     * original Triangulation.
     */
    Boolean    face_is_glued[4];
};

static Boolean    better_terse(TerseTriangulation *challenger, TerseTriangulation *
    defender);
static void        attach_extra(Triangulation *manifold);

```

```

static void      free_extra(Triangulation *manifold);
static void      initialize_extra(Triangulation *manifold);
static Triangulation *bare_bones_triangulation(TerseTriangulation *tt);

```

```

TerseTriangulation *tri_to_canonical_terse(
    Triangulation *manifold,
    Boolean      respect_orientation)
{
    /*
     * Compute all TerseTriangulations, ranging over all possible
     * choices of base_tetrahedron and base_permutation, and
     * return the one that is "lexicographically least".
     */

    TerseTriangulation *defender,
                      *challenger;
    Tetrahedron *tet;
    int i;
    Permutation p;

    defender = tri_to_terse(manifold);

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

        for (i = 0; i < 24; i++)
        {
            p = permutation_by_index[i];

            if (manifold->orientability == oriented_manifold
                && respect_orientation == TRUE
                && parity[p] == 1 /* odd permutation */)
                continue;

            challenger = tri_to_terse_with_base(manifold, tet, p);

            if (better_terse(challenger, defender) == TRUE)
            {
                free_terse_triangulation(defender);
                defender = challenger;
                challenger = NULL;
            }
            else
            {
                free_terse_triangulation(challenger);
                challenger = NULL;
            }
        }

    return defender;
}

```

```

static Boolean better_terse(
    TerseTriangulation *challenger,
    TerseTriangulation *defender)
{
    int i;

    if (challenger->num_tetrahedra != defender->num_tetrahedra)
        uFatalError("better_terse", "terse_triangulation");

    for (i = 0; i < 2*challenger->num_tetrahedra; i++)
    {
        if (challenger->glues_to_old_tet[i] == FALSE
            && defender->glues_to_old_tet[i] == TRUE)
            return TRUE;

        if (challenger->glues_to_old_tet[i] == TRUE
            && defender->glues_to_old_tet[i] == FALSE)
            return FALSE;
    }
}

```

```

for (i = 0; i < challenger->num_tetrahedra + 1; i++)
{
    if (challenger->which_old_tet[i] < defender->which_old_tet[i])
        return TRUE;
    if (challenger->which_old_tet[i] > defender->which_old_tet[i])
        return FALSE;
}

for (i = 0; i < challenger->num_tetrahedra + 1; i++)
{
    if (challenger->which_gluing[i] < defender->which_gluing[i])
        return TRUE;
    if (challenger->which_gluing[i] > defender->which_gluing[i])
        return FALSE;
}

return FALSE; /* challenger and defender are identical, except perhaps CS */
}

```

```

TerseTriangulation *tri_to_terse(
    Triangulation *manifold)
{
    /*
     * Pick an arbitrary base tetrahedron.
     *
     * Note: It is essential that the convert_tri_to_terse_indices and
     * convert_terse_to_tri_indices Permutations be orientation_preserving.
     * Together with the properties of terse_to_tri() and orient(), this
     * insures that when an oriented Triangulation is converted to a
     * TerseTriangulation and then back to a regular Triangulation, the
     * Orientation will be preserved (the proof is that all functions
     * preserve the Orientation of the base Tetrahedron).
     */
    return tri_to_terse_with_base( manifold,
                                   manifold->tet_list_begin.next,
                                   IDENTITY_PERMUTATION);
}

```

```

TerseTriangulation *tri_to_terse_with_base(
    Triangulation *manifold,
    Tetrahedron *base_tetrahedron,
    Permutation base_permutation)
{
    TerseTriangulation *tt;
    int count_glues_to_old_tet,
        count_which_old_tet,
        count_which_gluing,
        tet_count,
        tet_index;
    Tetrahedron **tet_list;
    FaceIndex terse_f,
              tri_f,
              nbr_terse_f,
              nbr_tri_f;
    Tetrahedron *tet,
               *nbr;

    /*
     * We assume the user wants to compress the complete manifold.
     * If Dehn fillings are present, something has gone wrong.
     * (This code could easily call fill_reasonable_cusps()
     * to handle partially filled manifolds, but at present
     * tri_to_terse() isn't used by the standard UI at all,
     * so I won't both making modifications until the need arises.)
     */
    if (all_cusps_are_complete(manifold) == FALSE)
        uFatalError("tri_to_terse", "terse_triangulation");

    /*
     * Attach an Extra field to each old Tetrahedron to keep
     * track of its role in the TerseTriangulation.
     */
}

```

```

    */
    attach_extra(manifold);

    /*
     * Initialize the Extra fields to show that initially no
     * Tetrahedra are in use.
     */
    initialize_extra(manifold);

    /*
     * Allocate space for the TerseTriangulation.
     */
    tt = alloc_terse(manifold->num_tetrahedra);

    /*
     * Set the number of Tetrahedra.
     */
    tt->num_tetrahedra = manifold->num_tetrahedra;

    /*
     * Set the Chern-Simons invariant, if it's present.
     * (Note that this assumes the current value is that of the
     * complete structure. I.e. no Dehn fillings are present.)
     */
    tt->CS_is_present = manifold->CS_value_is_known;
    tt->CS_value = manifold->CS_value[ultimate];

    /*
     * Keep track of how many entries have been written into
     * each of the TerseTriangulation's arrays.
     */
    count_glues_to_old_tet = 0;
    count_which_old_tet = 0;
    count_which_gluing = 0;

    /*
     * Keep track of how many Tetrahedra have been
     * incorporated into the TerseTriangulation.
     */
    tet_count = 0;

    /*
     * Keep an array which tells us where to find the Tetrahedra which
     * have been incorporated into the TerseTriangulation. The array
     * is indexed by a Tetrahedron's index in the TerseTriangulation,
     * not the original Triangulation. tet_count tells the number of
     * elements presently on the array.
     */
    tet_list = NEW_ARRAY(manifold->num_tetrahedra, Tetrahedron *);

    /*
     * Strictly speaking it shouldn't be necessary, but as a guard
     * against errors let's initialize the tet_list to all NULL's.
     */
    for (tet_index = 0; tet_index < manifold->num_tetrahedra; tet_index++)
        tet_list[tet_index] = NULL;

    /*
     * Initialize the base Tetrahedron.
     */
    base_tetrahedron->extra->in_use = TRUE;
    base_tetrahedron->extra->index = 0;
    base_tetrahedron->extra->convert_tri_to_terse_indices = base_permutation;
    base_tetrahedron->extra->convert_terse_to_tri_indices = inverse_permutation
    [base_permutation];
    tet_list[tet_count++] = base_tetrahedron;

    /*
     * Go through the faces of the Tetrahedra on the tet_list,
     * noting where each is glued and creating the TerseTriangulation.
     * Don't worry that the tet_list initially contains only one element.
     * The connectedness of the manifold implies that the remaining
     * Tetrahedra will all arrive on time.
     */

```

```

for (tet_index = 0; tet_index < manifold->num_tetrahedra; tet_index++)
{
    /*
     * Dereference the Tetrahedron under consideration,
     * and do a quick error check.
     */

    tet = tet_list[tet_index];
    if (tet == NULL || tet->extra->in_use == FALSE)
        uFatalError("tri_to_terse", "terse_triangulation");

    /*
     * Consider each face, in order.
     */

    for (terse_f = 0; terse_f < 4; terse_f++)
    {
        /*
         * If this face is already glued, do nothing.
         */
        if (tet->extra->face_is_glued[terse_f] == TRUE)
            continue;

        /*
         * Otherwise, see what's it should be glued to.
         */

        tri_f    = EVALUATE(tet->extra->convert_terse_to_tri_indices, terse_f);
        nbr       = tet->neighbor[tri_f];

        /*
         * Is the neighbor already part of the TerseTriangulation?
         */

        if (nbr->extra->in_use == TRUE)
        {
            /*
             * The neighbor is already part of the TerseTriangulation.
             */

            /*
             * Make the appropriate entries in the TerseTriangulation.
             * (Note that compose_permutations() composes right to
             * left, so that first we convert from the TerseTriangulation
             * indices on tet to the standard Triangulation indices,
             * then we do the gluing to get the corresponding
             * standard Triangulation indices on nbr, then we convert
             * to the TerseTriangulation indices on nbr.)
             */

            tt->glues_to_old_tet[count_glues_to_old_tet++] = TRUE;
            tt->which_old_tet[count_which_old_tet++] = nbr->extra->index;
            tt->which_gluing[count_which_gluing++]
                = compose_permutations(
                    compose_permutations(
                        nbr->extra->convert_tri_to_terse_indices,
                        tet->gluing[tri_f]
                    ),
                    tet->extra->convert_terse_to_tri_indices
                );

            /*
             * Make the appropriate entries in the Extra fields.
             */

            nbr_tri_f    = EVALUATE(tet->gluing[tri_f], tri_f);
            nbr_terse_f = EVALUATE(nbr->extra->convert_tri_to_terse_indices, nbr_tri_f)✎
        }
        else

```

```

{
    /*
     * The neighbor is not yet part of the TerseTriangulation.
     */
    tt->glues_to_old_tet[count_glues_to_old_tet++] = FALSE;

    /*
     * Set up nbr's Extra fields.
     *
     * We must define nbr->extra->convert_terse_to_tri_indices
     * so that the following diagram commutes:
     *
     *      tet                      nbr
     *      tri  .-----gluing----->.
     *           ^                      ^
     *           |                      |
     *      tet->extra->          nbr->extra->
     *      terse_to_tri        terse_to_tri
     *           |                      |
     *      terse  .-----identity----->.
     */
    nbr->extra->in_use = TRUE;
    nbr->extra->index = tet_count;
    nbr->extra->convert_terse_to_tri_indices =
        compose_permutations(
            tet->gluing[tri_f],
            tet->extra->convert_terse_to_tri_indices
        );
    nbr->extra->convert_tri_to_terse_indices
        = inverse_permutation[
            nbr->extra->convert_terse_to_tri_indices];
    /*
     * initialize_extra() has already initialized the
     * nbr->extra->face_is_glued[] fields.
     */

    /*
     * Enter nbr on the tet_list.
     * Note that tet_count is incremented after setting
     * nbr->extra->index above.
     */
    tet_list[tet_count++] = nbr;

    /*
     * Record that these faces have been glued.
     * Because the gluing is the identity relative to
     * the TerseTriangulation, nbr_terse_f == terse_f.
     */
    tet->extra->face_is_glued[terse_f] = TRUE;
    nbr->extra->face_is_glued[terse_f] = TRUE;
}
}

/*
 * Free the tet_list.
 */
my_free(tet_list);

/*
 * Free the Extra fields.
 */
free_extra(manifold);

/*
 * As a guard against errors, make sure
 * the array lengths came out right.
 */

if (count_glues_to_old_tet != 2 * manifold->num_tetrahedra
    || count_which_old_tet != manifold->num_tetrahedra + 1
    || count_which_gluing != manifold->num_tetrahedra + 1
    || tet_count != manifold->num_tetrahedra)

```

```

        uFatalError("tri_to_terse", "terse_triangulation");

    return tt;
}

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

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * Make sure no other routine is using the "extra"
         * field in the Tetrahedron data structure.
         */
        if (tet->extra != NULL)
            uFatalError("attach_extra", "terse_triangulation");

        /*
         * Attach the locally defined struct extra.
         */
        tet->extra = NEW_STRUCT(Extra);
    }
}

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

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * Free the struct extra.
         */
        my_free(tet->extra);

        /*
         * Set the extra pointer to NULL to let other
         * modules know we're done with it.
         */
        tet->extra = NULL;
    }
}

static void initialize_extra(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    FaceIndex f;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * Really only the in_use and face_is_glued[] fields need to be
         * initialized, but better to be extra safe and do 'em all.
         */

        tet->extra->in_use = FALSE;
        tet->extra->index = -1;
        tet->extra->convert_tri_to_terse_indices = 0x00;
        tet->extra->convert_terse_to_tri_indices = 0x00;
    }
}

```

```

        for (f = 0; f < 4; f++)
            tet->extra->face_is_glued[f] = FALSE;
    }
}

TerseTriangulation *alloc_terse(
    int num_tetrahedra)
{
    /*
     * The global optimizer screws up here.
     * See the file "notes 2.1" for details.
     * The following pragma turns the optimizer off for this function only.
     */
#ifdef __SC__
    #if ( __SC__ >= 0x800 )
    #pragma options(!global_optimizer);
    #endif
#endif

    TerseTriangulation *tt;

    tt = NEW_STRUCT(TerseTriangulation);

    tt->glues_to_old_tet = NEW_ARRAY(2 * num_tetrahedra, Boolean);
    tt->which_old_tet = NEW_ARRAY(num_tetrahedra + 1, int);
    tt->which_gluing = NEW_ARRAY(num_tetrahedra + 1, Permutation);

    return tt;
}

Triangulation *terse_to_tri(
    TerseTriangulation *tt)
{
    Triangulation *manifold;

    /*
     * Begin by setting up the bare bones Triangulation,
     * with only the neighbor and gluing fields set.
     */
    manifold = bare_bones_triangulation(tt);

    /*
     * Attempt to orient the manifold. Note that
     *
     * (1) orient() works fine when only the neighbor
     *     and gluing fields are set.
     *
     * (2) tri_to_terse() followed by terse_to_tri() will yield the
     *     manifold's original orientation. This is because
     *
     *     (A) tri_to_terse() chooses the base tetrahedron to be
     *         manifold->tet_list_begin.next, with its original
     *         VertexIndices (any orientation_preserving Permutation
     *         would do). (The VertexIndices on all other Tetrahedra
     *         may change -- sometimes in orientation_reversing ways --
     *         in passing to the TerseTriangulation, but that's OK.)
     *
     *     (B) bare_bones_triangulation preserves all VertexIndices
     *         in passing from the TerseTriangulation to the
     *         regular Triangulation.
     *
     *     (B) orient() preserves the Vertex indices on
     *         manifold->tet_list_begin.next. (The VertexIndices on
     *         all other Tetrahedra may change.)
     *
     *     Because manifold->tet_list_begin.next has its original
     *     VertexIndices intact, we know the orientation has not changed.
     */
    orient(manifold);

    /*
     * Create the Cusps.

```



```

    */
    create_cusps(manifold);

    /*
     * Create and orient the EdgeClasses.
     */
    create_edge_classes(manifold);
    orient_edge_classes(manifold);

    /*
     * Install an arbitrary set of peripheral curves.
     *
     * Notes:
     *
     * (1) After the hyperbolic structure is in place we'll
     *     replace these arbitrary curves with a canonical set.
     *
     * (2) We call peripheral_curves() after orient(), so that
     *     if the manifold is orientable the peripheral curves
     *     will respect the standard orientation convention.
     *
     * (3) peripheral_curves() will determine the CuspTopology of
     *     each Cusp, and write it into the cusp->topology field.
     */
    peripheral_curves(manifold);

    /*
     * Count the total number of Cusps, and also the number
     * with torus and Klein bottle CuspTopology.
     */
    count_cusps(manifold);

    /*
     * Attempt to compute a hyperbolic structure.
     */
    find_complete_hyperbolic_structure(manifold);

    /*
     * Install the (almost) canonical set of generators.
     */
    install_shortest_bases(manifold);

    /*
     * Install the Chern-Simons invariant, if one is present.
     */
    if (tt->CS_is_present)
        set_CS_value(manifold, tt->CS_value);

    return manifold;
}

static Triangulation *bare_bones_triangulation(
    TerseTriangulation *tt)
{
    Triangulation      *manifold;
    int                count_glues_to_old_tet,
                      count_which_old_tet,
                      count_which_gluing,
                      tet_count,
                      tet_index;
    Tetrahedron        **tet_list;
    FaceIndex          f,
                      nbr_f;
    Tetrahedron        *tet,
                      *nbr;
    Permutation         gluing;
    int                i;

    /*
     * Set up the header structure.
     */
    manifold = NEW_STRUCT(Triangulation);
    initialize_triangulation(manifold);

```

```

/*
 * Set the manifold's name to DAFAULT_NAME.
 */
manifold->name = NEW_ARRAY(strlen(DAFAULT_NAME) + 1, char);
strcpy(manifold->name, DAFAULT_NAME);

/*
 * Record the number of Tetrahedra.
 */
manifold->num_tetrahedra = tt->num_tetrahedra;

/*
 * Allocate and initialize the Tetrahedra, and temporarily
 * record their addresses in the tet_list.
 */
tet_list = NEW_ARRAY(tt->num_tetrahedra, Tetrahedron *);
for (i = 0; i < tt->num_tetrahedra; i++)
{
    tet_list[i] = NEW_STRUCT(Tetrahedron);
    initialize_tetrahedron(tet_list[i]);
    tet_list[i]->index = i;
    INSERT_BEFORE(tet_list[i], &manifold->tet_list_end);
}

/*
 * Keep track of how many entries have been read from
 * each of the TerseTriangulation's arrays.
 */
count_glues_to_old_tet = 0;
count_which_old_tet = 0;
count_which_gluings = 0;

/*
 * Initially we imagine a single Tetrahedron (namely tet_list[0])
 * to be "worked into the system".
 */
tet_count = 1;

/*
 * Go down the list, setting the neighbors and gluings
 * as specified by the TerseTriangulation.
 *
 * This code is conceptually simpler than the corresponding
 * code in tri_to_terse, because here the regular Triangulation
 * indexing system coincides with the TerseTriangulation
 * indexing system.
 */
for (tet_index = 0; tet_index < manifold->num_tetrahedra; tet_index++)
    for (f = 0; f < 4; f++)
        if (tet_list[tet_index]->neighbor[f] == NULL)
        {
            tet = tet_list[tet_index];

            if (tt->glues_to_old_tet[count_glues_to_old_tet++] == TRUE)
            {
                nbr = tet_list[tt->which_old_tet[count_which_old_tet++]];
                gluing = tt->which_gluings[count_which_gluings++];
                nbr_f = EVALUATE(gluings, f);
            }
            else
            {
                nbr = tet_list[tet_count++];
                gluing = IDENTITY_PERMUTATION;
                nbr_f = f;
            }

            tet->neighbor[f] = nbr;
            tet->gluing[f] = gluing;
            nbr->neighbor[nbr_f] = tet;
            nbr->gluing[nbr_f] = inverse_permutation[gluing];
        }

```

```
    }

    /*
     * Free the Tetrahedron address list.
     */
    my_free(tet_list);

    /*
     * As a guard against errors, make sure
     * the array lengths came out right.
     */
    if (count_glues_to_old_tet != 2 * manifold->num_tetrahedra
        || count_which_old_tet != manifold->num_tetrahedra + 1
        || count_which_gluing != manifold->num_tetrahedra + 1
        || tet_count != manifold->num_tetrahedra)

        uFatalError("terse_to_tri", "terse_triangulation");

    return manifold;
}

void free_terse_triangulation(
    TerseTriangulation *tt)
{
    my_free(tt->glues_to_old_tet);
    my_free(tt->which_old_tet);
    my_free(tt->which_gluing);

    my_free(tt);
}
```