

```

/*
 * tersest_triangulation.c
 *
 * This file provides the functions
 *
 * void terse_to_tersest( TerseTriangulation *terse,
 *                        TersestTriangulation tersest);
 *
 * void tersest_to_terse( TersestTriangulation tersest,
 *                        TerseTriangulation **terse);
 *
 * void tri_to_tersest( Triangulation *manifold,
 *                      TersestTriangulation tersest);
 *
 * void tersest_to_tri( TersestTriangulation tersest,
 *                      Triangulation **manifold);
 *
 * terse_to_tersest() and tersest_to_terse() convert back and forth
 * between TerseTriangulations and TersestTriangulations.
 * tersest_to_terse() allocates space for its result, but
 * terse_to_tersest() does not.
 *
 * tri_to_tersest() is the composition of tri_to_terse() and terse_to_tersest().
 * tersest_to_tri() is the composition of tersest_to_terse() and terse_to_tri().
 */

```

```

#include "kernel.h"

```

```

void terse_to_tersest(
    TerseTriangulation *terse,
    TersestTriangulation tersest)
{
    int i,
        j;
    double cs,
        integer_part;

    /*
     * The TersestTriangulation is suitable only for Triangulations
     * with 7 or fewer Tetrahedra.
     */
    if (terse->num_tetrahedra > 7)
        uFatalError("terse_to_tersest", "tersest_triangulation");

    /*
     * Compress the glues_to_old_tet[] array.
     */

    for (i = 0; i < 2; i++)
    {
        tersest[i] = 0;
        for (j = 8; --j >= 0; )
        {
            tersest[i] <= 1;
            if (8*i + j < 2 * terse->num_tetrahedra)
                tersest[i] |= terse->glues_to_old_tet[8*i + j];
        }
    }

    /*
     * Compress which_old_tet[] and which_gluing[]
     */

    for (i = 0; i < terse->num_tetrahedra + 1; i++)
    {
        tersest[i+2] = terse->which_old_tet[i];
        tersest[i+2] <= 5;
        tersest[i+2] |= index_by_permutation[terse->which_gluing[i]];
    }

    /*
     * Set unused bytes to 0.
     */
}

```

```

for (i = terse->num_tetrahedra + 1; i < 8; i++)
    tersest[i+2] = 0;

/*
 * Compress the Chern-Simons invariant, if present.
 */

if (terse->CS_is_present)
{
    /*
     * Set the highest-order bit in byte 1 to TRUE.
     */
    tersest[1] |= 0x80;

    /*
     * Copy the given value.
     */
    cs = terse->CS_value;

    /*
     * Make sure it's in the range [-1/4, +1/4).
     */
    while (cs < -0.25)
        cs += 0.5;
    while (cs >= 0.25)
        cs -= 0.5;

    /*
     * cs = 2*cs + 1/2 places cs in the range [0, 1)
     */
    cs = 2*cs + 0.5;

    /*
     * Peel off the significant binary digits 8 at a time
     * and stash them in tersest[10] through tersest[17].
     */
    for (i = 0; i < 8; i++)
    {
        /*
         * Do a "floating point bitshift".
         */
        cs *= (double) 0x100;

        /*
         * Extract the integer part, and replace cs with
         * the fractional part.
         */
        cs = modf(cs, &integer_part);

        /*
         * The integer part should be in the range [0, 256).
         * Store it as a 1-byte unsigned integer in tersest[10 + i].
         */
        tersest[10 + i] = (unsigned char) integer_part;
    }
}
else
{
    /*
     * Set the highest-order bit in byte 1 to FALSE.
     */
    tersest[1] &= 0x7F;

    for (i = 0; i < 8; i++)
        tersest[10 + i] = 0;
}
}

void tersest_to_terse(
    TersestTriangulation    tersest,
    TerseTriangulation      **terse)
{

```

```

int    i,
        j,
        byte,
        glue_data[16],
        num_tetrahedra,
        num_free_faces,
        bits_read;
double cs;

/*
 * Extract the bits for the glues_to_old_tet[] array. We don't yet
 * know how many are meaningful, so get all 16 bits, and then use them
 * to deduce the number of Tetrahedra. (Yes, I know the last two can't
 * possibly be meaningful, but reading them keeps the code simple.)
 */

for (i = 0; i < 2; i++)
{
    byte = tersest[i];
    for (j = 0; j < 8; j++)
    {
        glue_data[8*i + j] = byte & 0x01;
        byte >>= 1;
    }
}

/*
 * Deduce the number of Tetrahedra.
 */

num_tetrahedra = 1;
num_free_faces = 4;
bits_read      = 0;

while (num_free_faces > 0)
    if (glue_data[bits_read++] == TRUE)
    {
        num_free_faces -= 2;
    }
    else
    {
        num_tetrahedra++;
        num_free_faces += 2;
    }

if (bits_read != 2 * num_tetrahedra
    || num_tetrahedra > 7)
    uFatalError("tersest_to_terse", "tersest_triangulation");

/*
 * Allocate space for the TerseTriangulation.
 */

(*terse) = alloc_terse(num_tetrahedra);

/*
 * Set num_tetrahedra.
 */

(*terse)->num_tetrahedra = num_tetrahedra;

/*
 * Set glues_to_old_tet[].
 */

for (i = 0; i < 2 * num_tetrahedra; i++)
    (*terse)->glues_to_old_tet[i] = glue_data[i];

/*
 * Set which_old_tet[] and which_gluing[].
 */

for (i = 0; i < num_tetrahedra + 1; i++)
{

```

```
    (*terse)->which_old_tet[i] = tersest[2 + i] >> 5;
    (*terse)->which_gluing[i] = permutation_by_index[tersest[2 + i] & 0x1F];
}

/*
 * Set the Chern-Simons invariant, if present.
 *
 * These steps reverse the corresponding steps documented
 * in terse_to_tersest() above.
 */

if (tersest[1] & 0x80)
{
    (*terse)->CS_is_present = TRUE;
    cs = 0.0;
    for (i = 8; --i >= 0; )
    {
        cs += (double) tersest[10 + i];
        cs /= (double) 0x100;
    }
    cs = (cs - 0.5) / 2.0;
    (*terse)->CS_value = cs;
}
else
{
    (*terse)->CS_is_present = FALSE;
    (*terse)->CS_value = 0.0;
}
}

void tri_to_tersest(
    Triangulation      *manifold,
    TersestTriangulation tersest)
{
    TerseTriangulation *terse;

    terse = tri_to_terse(manifold);
    terse_to_tersest(terse, tersest);

    free_terse_triangulation(terse);
}

void tersest_to_tri(
    TersestTriangulation tersest,
    Triangulation      **manifold)
{
    TerseTriangulation *terse;

    tersest_to_terse(tersest, &terse);
    *manifold = terse_to_tri(terse);

    free_terse_triangulation(terse);
}
```