

```

/*
 * orient.c
 *
 * This file provides the functions
 *
 *     void orient(Triangulation *manifold);
 *     void reorient(Triangulation *manifold);
 *     void fix_peripheral_orientations(Triangulation *manifold);
 *
 * orient() attempts to consistently orient the Tetrahedra of the
 * Triangulation *manifold. It begins with manifold->tet_list_begin.next
 * and recursively reorients its neighbors as necessary until either
 * all the tetrahedra are consistently oriented, or it discovers that
 * the manifold is nonorientable. (In particular, the Orientation of
 * an orientable Triangulation will match the initial Orientation of
 * manifold->tet_list_begin.next, which is important in subdivide.c and
 * terse_triangulation.c.) orient() sets the manifold->orientability
 * field to oriented_manifold or nonorientable_manifold, as appropriate.
 * Its method for reorienting a tetrahedron is to swap the indices
 * of vertices 2 and 3, and adjust the relevant fields accordingly
 * (including the gluing fields of neighboring tetrahedra).
 * It is aware of the fact that some fields may not yet be set,
 * and doesn't try to modify data structures if they aren't present
 * (all it requires are the neighbor and gluing fields).
 *
 * If the manifold is orientable, then
 *
 * (1) All peripheral curves (i.e. the meridians and longitudes)
 *     are transferred to the right_handed sheets of the orientation
 *     double covers of the cusps. This makes their holonomies complex
 *     analytic functions of the tetrahedron shapes, rather than
 *     complex conjugates of complex analytic functions. (However,
 *     the {meridian, longitude} pair may no longer be right-handed.
 *     To fix that, call fix_peripheral_orientations().)
 *
 * (2) All edge_orientations are set to right_handed.
 *
 * orient() and orient_edge_classes() may be called in either order.
 *
 * If orient() is called first, then
 *     If the manifold is orientable,
 *         orient() will set all edge_orientations to right_handed, and then
 *         orient_edge_classes() will (redundantly) do the same thing.
 *     If the manifold is nonorientable,
 *         orient() won't set the edge_orientations, but
 *         orient_edge_classes() will.
 *
 * If orient_edge_classes() is called first, then
 *     If the manifold is orientable,
 *         orient_edge_classes() will assign an arbitrary Orientation
 *         to each EdgeClass, and then
 *         orient() will overwrite it with the right_handed Orientation.
 *     If the manifold is nonorientable,
 *         orient_edge_classes() will assign an arbitrary Orientation
 *         to each EdgeClass, and
 *         orient() will preserve it.
 *
 * orient() could be made available to the UI with no modifications
 * beyond moving the prototype.
 *
 * reorient() reverses the orientation of all the Tetrahedra
 * in the Triangulation, by swapping VertexIndices 2 and 3 as described
 * above. If the manifold is orientable, reorient() also
 * transfers the peripheral curves to the right_handed sheets of the
 * double covers and sets all edge_orientations to right_handed, as
 * described above. It reverses the directions of all meridians, so the
 * peripheral curves will continue to adhere to the standard orientation
 * convention. reorient() is intended for oriented manifolds, but
 * nothing terrible will happen if you pass it a nonorientable manifold.
 *
 * fix_peripheral_orientations() makes sure each {meridian, longitude}

```

```

* pairs obeys the right-hand rule. It should be called only for
* orientable manifolds, typically following a call to orient().
*
* Note to myself: Eventually I may want to make transfer_peripheral_curves()
* responsible for making the peripheral curves adhere to the standard
* orientation convention. If this is done, reorient() won't have to
* explicitly change the directions of meridians. (However, this
* change would require that orient() check whether Cusps are present.)
*/

#include "kernel.h"

static void reverse_orientation(Tetrahedron *tet);
static void renumber_neighbors_and_gluings(Tetrahedron *tet);
static void renumber_cusps(Tetrahedron *tet);
static void renumber_peripheral_curves(Tetrahedron *tet);
static void renumber_edge_classes(Tetrahedron *tet);
static void renumber_shapes(Tetrahedron *tet);
static void renumber_shape_histories(Tetrahedron *tet);
static void renumber_one_part(ComplexWithLog edge_parameters[3]);
static void swap_rows(int m[4][4], int a, int b);
static void swap_columns(int m[4][4], int a, int b);
static void swap_sheets(int m[2][4][4]);
static void transfer_peripheral_curves(Triangulation *manifold);
static void make_all_edge_orientations_right_handed(Triangulation *manifold);
static void reverse_all_meridians(Triangulation *manifold);

void orient(
    Triangulation *manifold)
{
    /*
     * Pick an arbitrary initial Tetrahedron,
     * and try to extend its orientation to the whole manifold.
     */
    extend_orientation(manifold, manifold->tet_list_begin.next);
}

void extend_orientation(
    Triangulation *manifold,
    Tetrahedron *initial_tet)
{
    Tetrahedron **queue,
                *tet;
    int queue_first,
        queue_last;
    FaceIndex f;

    /*
     * Set all the tet->flag fields to FALSE,
     * to show that no Tetrahedra have been visited.
     */

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

        tet->flag = FALSE;

    /*
     * Tentatively assume the manifold is orientable.
     */
    manifold->orientability = oriented_manifold;

    /*
     * Allocate space for a queue of pointers to the Tetrahedra.
     * Each Tetrahedron will appear on the queue exactly once,
     * so an array of length manifold->num_tetrahedra will be just right.
     */
    queue = NEW_ARRAY(manifold->num_tetrahedra, Tetrahedron *);

    /*
     * Put the initial Tetrahedron on the queue,

```

```

    * and mark it as visited.
    */
    queue_first = 0;
    queue_last = 0;
    queue[0] = initial_tet;
    queue[0]->flag = TRUE;

    /*
    * Start processing the queue.
    */
do
{
    /*
    * Pull a Tetrahedron off the front of the queue.
    */
    tet = queue[queue_first++];

    /*
    * Look at the four neighboring Tetrahedra.
    */
    for (f = 0; f < 4; f++)
    {
        /*
        * If the neighbor hasn't been visited...
        */
        if (tet->neighbor[f]->flag == FALSE)
        {
            /*
            * ...reverse its orientation if necessary...
            */
            if (parity[tet->gluing[f]] == orientation_reversing)
                reverse_orientation(tet->neighbor[f]);

            /*
            * ...mark it as visited...
            */
            tet->neighbor[f]->flag = TRUE;

            /*
            * ...and put it on the back of the queue.
            */
            queue[++queue_last] = tet->neighbor[f];
        }
        /*
        * If the neighbor has been visited . . .
        */
        else
        {
            /*
            * ...check whether its orientation is consistent.
            */
            if (parity[tet->gluing[f]] == orientation_reversing)
                manifold->orientability = nonorientable_manifold;
        }
    }
}

/*
* Keep going until either we discover the manifold is nonorientable,
* or the queue is exhausted.
*/
while (manifold->orientability == oriented_manifold
    && queue_first <= queue_last);

/*
* Free the memory used for the queue.
*/
my_free(queue);

/*
* An "unnecessary" (but quick) error check.
*/
if (manifold->orientability == oriented_manifold
    && ( queue_first != manifold->num_tetrahedra
        || queue_last != manifold->num_tetrahedra - 1))

```

```

    uFatalError("orient", "orient");

/*
 * Another error check.
 * We should have oriented a manifold before attempting to
 * compute the Chern-Simons invariant.
 */
if (manifold->CS_value_is_known || manifold->CS_fudge_is_known)
    uFatalError("orient", "orient");

/*
 * Respect the conventions for peripheral curves and
 * edge orientations in oriented manifolds.
 */
if (manifold->orientability == oriented_manifold)
{
    transfer_peripheral_curves(manifold);
    make_all_edge_orientations_right_handed(manifold);
}

}

/*
 * reverse_orientation() reverses the orientation of a Tetrahedron
 * by swapping the indices of vertices 2 and 3. It adjusts all
 * relevant fields, including the gluing fields of neighboring
 * tetrahedra.
 */

static void reverse_orientation(
    Tetrahedron *tet)
{
    renumber_neighbors_and_gluings(tet);
    renumber_cusps(tet);
    renumber_peripheral_curves(tet);
    renumber_edge_classes(tet);
    renumber_shapes(tet);
    renumber_shape_histories(tet);
}

static void renumber_neighbors_and_gluings(
    Tetrahedron *tet)
{
    Tetrahedron *temp_neighbor;
    Permutation temp_gluing;
    int i,
        j,
        d[4],
        temp_digit;
    Tetrahedron *nbr_tet;

/*
 * Renumbering the neighbors is easy: we simply swap
 * neighbor[2] and neighbor[3].
 */

    temp_neighbor = tet->neighbor[2];
    tet->neighbor[2] = tet->neighbor[3];
    tet->neighbor[3] = temp_neighbor;

/*
 * Renumbering the gluings is trickier, because three
 * changes are required:
 *
 * Change A: Swap gluing[2] and gluing[3].
 *
 * Change B: Within each gluing of tet, swap the image of
 * vertex 2 and the image of vertex 3, e.g. 0312 -> 3012.
 *
 * Change C: For each gluing of a face (typically of a Tetrahedron
 * other than tet) that glues to tet, interchange
 * 2 and 3, e.g. 0312 -> 0213.
 */

```

```

    */

/*
 * Change A: Swap gluing[2] and gluing[3].
 */

temp_gluing = tet->gluing[2];
tet->gluing[2] = tet->gluing[3];
tet->gluing[3] = temp_gluing;

/*
 * Changes B and C are carried out for each of the four gluings of tet.
 */

for (i = 0; i < 4; i++)
{
    /*
     * Change B: Swap the image of vertex 2 and the image of vertex 3.
     */

    /*
     * Unpack the digits of the gluing.
     */
    for (j = 0; j < 4; j++)
    {
        d[j] = tet->gluing[i] & 0x3;
        tet->gluing[i] >>= 2;
    }

    /*
     * Swap the digits in positions 2 and 3.
     */
    temp_digit = d[3];
    d[3] = d[2];
    d[2] = temp_digit;

    /*
     * Repack the digits.
     */
    for (j = 4; --j >= 0; )
    {
        tet->gluing[i] <<= 2;
        tet->gluing[i] += d[j];
    }

    /*
     * Change C: Fix up the inverse of tet->gluing[i].
     *
     * If tet->neighbor[i] != tet, we simply write the inverse of
     * tet->gluing[i] into the appropriate gluing field of the neighbor.
     *
     * If tet->neighbor[i] == tet, the simple approach doesn't work
     * because of the messy interaction between Changes B and C.
     * Instead, we exploit the fact that tet->gluing[i] is the inverse
     * of some other gluing of tet, and apply Change C directly to
     * tet->gluing[i] rather than its inverse.
     */

    nbr_tet = tet->neighbor[i];

    if (nbr_tet != tet)
    {
        /*
         * Write the inverse directly.
         */

        nbr_tet->gluing[EVALUATE(tet->gluing[i],i)] = inverse_permutation[tet->gluing
[i]];
    }
    else
    {
        /*
         * Perform Change C on tet->gluing[i].

```

```

        */

        /*
        *  Unpack the digits.
        */
        for (j = 0; j < 4; j++)
        {
            d[j] = tet->gluing[i] & 0x3;
            tet->gluing[i] >>= 2;
        }

        /*
        *  Swap 2 and 3 in the images.
        */
        for (j = 0; j < 4; j++)
            switch (d[j])
            {
                case 0:
                case 1:
                    /* leave d[j] alone */
                    break;
                case 2:
                    d[j] = 3;
                    break;
                case 3:
                    d[j] = 2;
                    break;
            }

        /*
        *  Repack the digits.
        */
        for (j = 4; --j >= 0; )
        {
            tet->gluing[i] <=& 2;
            tet->gluing[i] += d[j];
        }
    }
}

static void renumber_cusps(
    Tetrahedron *tet)
{
    Cusp      *temp_cusp;

    temp_cusp      = tet->cusp[2];
    tet->cusp[2]    = tet->cusp[3];
    tet->cusp[3]    = temp_cusp;
}

static void renumber_peripheral_curves(
    Tetrahedron *tet)
{
    int i,
        j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            swap_rows    (tet->curve[i][j], 2, 3);
            swap_columns(tet->curve[i][j], 2, 3);
        }

        swap_sheets(tet->curve[i]);
    }
}

static void renumber_edge_classes(

```

```

    Tetrahedron *tet)
{
    EdgeClass    *temp_edge_class;
    Orientation temp_edge_orientation;
    EdgeIndex    i;

    temp_edge_class    = tet->edge_class[1];
    tet->edge_class[1]  = tet->edge_class[2];
    tet->edge_class[2]  = temp_edge_class;

    temp_edge_class    = tet->edge_class[3];
    tet->edge_class[3]  = tet->edge_class[4];
    tet->edge_class[4]  = temp_edge_class;

    for (i = 1; i < 5; i++)
        if (tet->edge_class[i] != NULL)
        {
            tet->edge_class[i]->incident_tet = tet;
            tet->edge_class[i]->incident_edge_index = i;
        }

    temp_edge_orientation    = tet->edge_orientation[1];
    tet->edge_orientation[1]  = tet->edge_orientation[2];
    tet->edge_orientation[2]  = temp_edge_orientation;

    temp_edge_orientation    = tet->edge_orientation[3];
    tet->edge_orientation[3]  = tet->edge_orientation[4];
    tet->edge_orientation[4]  = temp_edge_orientation;

    for (i = 0; i < 6; i++)
        tet->edge_orientation[i] = ! tet->edge_orientation[i];
}

static void renumber_shapes(
    Tetrahedron *tet)
{
    /*
     * Renumber the TetShapes iff they are actually present.
     */

    int i,
        j;

    if (tet->shape[complete] != NULL)

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

                renumber_one_part(tet->shape[i]->cwl[j]);
}

static void renumber_one_part(
    ComplexWithLog edge_parameters[3])
{
    ComplexWithLog temp;
    int i;

    /*
     * Swap the indices on edges 1 and 2.
     */

    temp          = edge_parameters[1];
    edge_parameters[1] = edge_parameters[2];
    edge_parameters[2] = temp;

    /*
     * Invert the modulus of each edge parameter, but leave
     * the angles the same.
     */

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

```

```

    {
        edge_parameters[i].log.real = - edge_parameters[i].log.real;
        edge_parameters[i].rect = complex_exp(edge_parameters[i].log);
    }
}

```

```

static void renumber_shape_histories(
    Tetrahedron *tet)
{
    int i;
    ShapeInversion *shape_inversion;

    for (i = 0; i < 2; i++)

        for ( shape_inversion = tet->shape_history[i];
              shape_inversion != NULL;
              shape_inversion = shape_inversion->next)

            switch (shape_inversion->wide_angle)
            {
                case 0:
                    shape_inversion->wide_angle = 0;
                    break;

                case 1:
                    shape_inversion->wide_angle = 2;
                    break;

                case 2:
                    shape_inversion->wide_angle = 1;
                    break;
            }
}

```

```

static void swap_rows(
    int m[4][4],
    int a,
    int b)
{
    int j,
        temp;

    for (j = 0; j < 4; j++)
    {
        temp = m[a][j];
        m[a][j] = m[b][j];
        m[b][j] = temp;
    }
}

```

```

static void swap_columns(
    int m[4][4],
    int a,
    int b)
{
    int i,
        temp;

    for (i = 0; i < 4; i++)
    {
        temp = m[i][a];
        m[i][a] = m[i][b];
        m[i][b] = temp;
    }
}

```

```

static void swap_sheets(
    int m[2][4][4])
{
    int i,
        j,
        temp;

```



```

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
        {
            temp                = m[right_handed][i][j];
            m[right_handed][i][j] = m[left_handed ][i][j];
            m[left_handed ][i][j] = temp;
        }
}

/*
 * We want the peripheral curves of an oriented manifold to lie
 * on the right_handed sheets of the orientation double covers
 * of the cusps. transfer_peripheral_curves() adds the curves
 * from the left_handed sheet to the right_handed sheet, and
 * sets the contents of the left_handed sheet to zero.
 */

static void transfer_peripheral_curves(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    PeripheralCurve c;
    VertexIndex v;
    FaceIndex f;

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

        for (c = 0; c < 2; c++)

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

                for (f = 0; f < 4; f++)
                {
                    tet->curve[c][right_handed][v][f] += tet->curve[c][left_handed][v][f];
                    tet->curve[c][left_handed][v][f] = 0;
                }
}

static void make_all_edge_orientations_right_handed(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    EdgeIndex e;

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

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

            tet->edge_orientation[e] = right_handed;
}

void reorient(
    Triangulation *manifold)
{
    Tetrahedron *tet;

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

        reverse_orientation(tet);

    if (manifold->orientability == oriented_manifold)
    {

```

```

    /*
     * The peripheral curves haven't gone anywhere,
     * but the sheets they are on are now considered
     * the left_handed sheets rather than the right_handed
     * sheets. So transfer them to what used to be the
     * left_handed sheets but are now the right_handed sheets.
     */
    transfer_peripheral_curves(manifold);

    /*
     * To adhere to the orientation conventions for peripheral curves
     * (see the documentation at the top of peripheral_curves.c)
     * we must reverse the directions of all meridians.
     *
     * Note that it was the act of transferring the peripheral curves
     * from the left_handed to right_handed sheets -- note the reversal
     * of the Tetrahedra -- that caused the violation of the orientaiton
     * convention. In particular, curves in nonorientable manifold, even
     * on (double covers of) Klein bottle cusps, still respect the convention.
     */
    reverse_all_meridians(manifold);

    /*
     * Adjust the edge orientations, too.
     */
    make_all_edge_orientations_right_handed(manifold);
}

/*
 * The Chern-Simons invariant of the manifold will be negated,
 * and the fudge factor will be different.
 */
if (manifold->CS_value_is_known)
{
    manifold->CS_value[ultimate] = - manifold->CS_value[ultimate];
    manifold->CS_value[penultimate] = - manifold->CS_value[penultimate];
}
compute_CS_fudge_from_value(manifold);
}

static void reverse_all_meridians(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    Cusp *cusp;
    int i,
        j,
        k;

    /*
     * Change the directions of all meridians.
     */

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

        for (i = 0; i < 2; i++)

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

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

                    tet->curve[M][i][j][k] = - tet->curve[M][i][j][k];

    /*
     * Negating the m coefficient of all Dehn fillings compensates for
     * the fact that we reversed the meridian, and gives us the same
     * (oriented) Dehn filling curve as before. However, this curve
     * will now wind clockwise around the core geodesics, relative to
     * the new orientation on the manifold. This causes a whole new
     * solution to be found to the gluing equations. To avoid this,
     * we reverse the direction of the Dehn filling curve (i.e. we

```

```

    * negate both the m and l coefficients). The net effect is that
    * we negate the l coefficient.
    *
    * This reversal of the Dehn filling curve is not really
    * necessary, and could be eliminated if it's ever causes problems.
    */

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

    cusp->l = - cusp->l;

/*
 * Adjust all cusp_shapes.
 * (The current cusp_shape of a filled Cusp will be Zero, but that's OK.)
 */

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

    for (i = 0; i < 2; i++) /* i = initial, current */

        cusp->cusp_shape[i].real = - cusp->cusp_shape[i].real;

/*
 * Adjust the holonomies.
 *
 * Changing the orientation of the manifold negates the imaginary
 * parts of log(H(m)) and log(H(l)).
 *
 * But we also reversed the direction of the meridian, which
 * negates both the real and imaginary parts of log(H(m)), so
 * the net effect on log(H(m)) is that its real part is negated.
 */

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

    for (i = 0; i < 2; i++) /* i = ultimate, penultimate */
    {
        cusp->holonomy[i][M].real = - cusp->holonomy[i][M].real;
        cusp->holonomy[i][L].imag = - cusp->holonomy[i][L].imag;
    }
}

void fix_peripheral_orientations(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    VertexIndex v;
    FaceIndex f;
    Cusp *cusp;

    /*
     * This function should get called only for orientable manifolds.
     */
    if (manifold->orientability != oriented_manifold)
        uFatalError("fix_peripheral_orientations", "orient");

    /*
     * Compute the intersection number of the meridian and longitude.
     */
    copy_curves_to_scratch(manifold, 0, FALSE);
    copy_curves_to_scratch(manifold, 1, FALSE);
    compute_intersection_numbers(manifold);

    /*
     * Reverse the meridian on cusps with intersection_number[L][M] == -1.
     */

```

```

/* which Tetrahedron */
for (tet = manifold->tet_list_begin.next;
     tet != &manifold->tet_list_end;
     tet = tet->next)

    /* which ideal vertex */
    for (v = 0; v < 4; v++)

        if (tet->cuspid[v]->intersection_number[L][M] == -1)

            /* which side of the vertex */
            for (f = 0; f < 4; f++)

                if (v != f)
                {
                    tet->curve[M][right_handed][v][f] = - tet->curve[M][right_handed]
[v][f];

                    if (tet->curve[M][left_handed][v][f] != 0.0
                        || tet->curve[L][left_handed][v][f] != 0.0)
                        uFatalError("fix_peripheral_orientations", "orient");
                }

/*
 * When we reverse the meridian we must also negate the meridional
 * Dehn filling coefficient in order to maintain the same (oriented)
 * Dehn filling curve as before. However, this Dehn filling curve
 * will wind clockwise around the core geodesics, relative to
 * the global orientation on the manifold (because the global
 * orientation disagrees with the local orientation we had been using
 * on the nonorientable manifold's torus cusp). This forces a whole
 * new solution to be found to the gluing equations. To avoid this,
 * we reverse the direction of the Dehn filling curve (i.e. we
 * negate both the m and l coefficients). The net effect is that
 * we negate the l coefficient.
 *
 * This reversal of the Dehn filling curve is not really
 * necessary, and could be eliminated if it's ever causes problems.
 */

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

    if (cusp->intersection_number[L][M] == -1)

        cusp->l = - cusp->l;
}

```