

```

/*
 * update_shapes.c
 *
 * This file provides the function
 *
 * void update_shapes(Triangulation *manifold, Complex *delta);
 *
 * which is called by do_Deihn_filling() in hyperbolic_structure.c.
 * update_shapes() updates the shapes of the tetrahedra in *manifold
 * by the amounts specified in the array delta. If necessary, delta
 * is first scaled so that no delta[i].real or delta[i].imag exceeds
 * the limit specified by the constant allowable_change (see below).
 *
 * The entries in delta are interpreted relative to the coordinate system
 * given by the coordinate_system field of each Tetrahedron, and the
 * indexing of delta is assumed to correspond to the index field of each
 * tetrahedron.
 */

/*
 * The allowable_change constant specifies the maximum amount
 * the log of the complex edge parameter may change.
 *
 * allowable_change.real is the maximum allowable change in
 * the log of its modulus, and
 *
 * allowable_change.imag is the maximum allowable change in
 * its argument.
 *
 * If necessary, all the delta[i] are scaled by a constant (between
 * zero and one) so that no delta[i] exceeds the allowable change.
 *
 * Setting allowable_change.
 *
 * A small value for allowable_change makes Newton's method slow,
 * but reliable. A larger value speeds it up, but increases the risk
 * of winding up on some funny branch of the solution space.
 * The values of allowable_change.real and allowable_change.imag
 * must not exceed 0.5, for the following reasons.
 *
 * (1) Because choose_coordinate_system() is called at the start of
 * each iteration of Newton's method, we know that the current
 * value of the edge parameter (relative to the chosen coordinate
 * system) satisfies  $|z-1| \geq 1$  and  $\text{Re}(z) \leq 0.5$  (see the comment
 * preceding choose_coordinate_system() in hyperbolic_structure.c).
 * Therefore if allowable_change.imag is less than  $\pi/6 = 0.52\dots$ ,
 * the parameter  $z$  cannot go more than half way to the singularity
 * at 1. If allowable_change.real is less than  $\log(2) = 0.69\dots$ ,
 * then  $z$  cannot go more than half way to the singularity at 0, nor
 * can it go "more than half way to infinity", in the sense that
 * its modulus cannot increase by more than a factor of two.
 *
 * (2) The code which maintains the shape_history assumes that when a
 * Tetrahedron's shape changes, the edge parameter given by
 * coordinate_system is the one passing through  $\pi \pmod{2\pi}$ , and
 * the other two edge parameters are passing through  $0 \pmod{2\pi}$ .
 * This assumption relies on the fact that allowable_change.imag
 * is less than  $\pi/6 = 0.52\dots$  .
 */

#include "kernel.h"

/*
 * The entries in allowable_change must not exceed 0.5.
 * See explanation above.
 */
static const Complex allowable_change = {0.5, 0.5};

static void scale_delta(Triangulation *manifold, Complex *delta);
static void recompute_shapes(Triangulation *manifold, Complex *delta);

void update_shapes(

```

```

    Triangulation  *manifold,
    Complex        *delta)
{
    scale_delta(manifold, delta);
    recompute_shapes(manifold, delta);
}

static void scale_delta(
    Triangulation  *manifold,
    Complex        *delta)
{
    int            i;
    Complex max;
    double         scaled_max,
                  factor;

    /*
     * Find the maximum values of delta[i].real and delta[i].imag.
     */

    max = Zero;

    for (i = 0; i < manifold->num_tetrahedra; i++)
    {
        if ( fabs(delta[i].real) > max.real )
            max.real = fabs(delta[i].real);

        if ( fabs(delta[i].imag) > max.imag )
            max.imag = fabs(delta[i].imag);
    }

    /*
     * Scale the solution if necessary.
     */

    scaled_max = MAX(
        max.real/allowable_change.real,
        max.imag/allowable_change.imag
    );

    if (scaled_max > 1.0)
    {
        factor = 1.0 / scaled_max;

        for (i = 0; i < manifold->num_tetrahedra; i++)
            delta[i] = complex_real_mult(factor, delta[i]);
    }
}

static void recompute_shapes(
    Triangulation  *manifold,
    Complex        *delta)
{
    Tetrahedron    *tet;
    int            i,
                  c[3];
    Complex         log_z,
                  z[3],
                  old_z,
                  new_z;
    ShapeInversion  *dead_shape_inversion,
                  *new_shape_inversion;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * The array c[] is used to index the coordinate systems.

```

```

    * For example, if tet->coordinate_system is 1, then
    * c[0] = 1 (the current coordinate system), c[1] = 2 (the next
    * one), and c[2] = 0 (the one after that, cyclically speaking).
    */

for (i = 0; i < 3; i++)
    c[i] = (tet->coordinate_system + i) % 3;

/*
 * Find the new value of log(z) in the primary coordinate system.
 */

log_z = complex_plus(
    tet->shape[filled]->cwl[ultimate][c[0]].log,    /* old log_z */
    delta[tet->index]                               /* change in log_z */
);

/*
 * Compute the new edge parameters in rectangular form.
 * Use z1 = 1/(1 - z0), etc.
 */

z[c[0]] = complex_exp(log_z);
z[c[1]] = complex_div( One, complex_minus(One, z[c[0]]) );
z[c[2]] = complex_div( One, complex_minus(One, z[c[1]]) );

/*
 * Note the old z[0] and the new z[0].
 *
 * If the Tetrahedron has experienced a ShapeInversion, update
 * its shape_history.
 *
 * Note that this approach is completely robust with respect
 * to roundoff errors. A Tetrahedron is considered positively
 * oriented if its shape has z.imag >= 0.0, and negatively
 * oriented if its shape has z.imag < 0.0. (It doesn't matter
 * whether z.imag == 0.0 is considered positively or negatively
 * oriented, just so we make a convention and use it
 * consistently.) Also, whether a Tetrahedron is perceived
 * as positively or negatively oriented is independent of its
 * coordinate_system: the above computation of z[c[0]], z[c[1]],
 * and z[c[2]] insures that the imaginary parts of all three will
 * have the same sign (-, 0, +), regardless of roundoff errors.
 */

old_z = tet->shape[filled]->cwl[ultimate][0].rect;
new_z = z[0];

if ((old_z.imag >= 0.0) != (new_z.imag >= 0.0))
{
    /*
     * The Tetrahedron has undergone a ShapeInversion.
     * Because old_z is in the region |z-1| >= 1 and Re(z) <= 0.5
     * (see the comment preceding choose_coordinate_system() in
     * hyperbolic_structure.c) and allowable_change.imag <= 0.5 < pi/6,
     * it follows that the edge parameter coordinate_system
     * passed through pi (mod 2 pi), and the other two edge
     * parameters passed through 0 (mod 2 pi). That is, the
     * ShapeInversion we are adding to the stack will have
     * wide_angle = coordinate_system.
     *
     * If the last item on the shape_history stack also has its
     * wide_angle field equal to the present coordinate_system,
     * then we remove it, because it cancels with the present
     * ShapeInversion. Otherwise we add the new ShapeInversion
     * to the stack.
     */

    /*
     * If there's a nonempty shape_history stack and the last
     * ShapeInversion has wide_angle == coordinate_system, then
     * remove it. It cancels with the ShapeInversion we were
     * about to put on the stack.
     */
}

```

```

    if (tet->shape_history[filled] != NULL
        && tet->shape_history[filled]->wide_angle == tet->coordinate_system)
    {
        dead_shape_inversion      = tet->shape_history[filled];
        tet->shape_history[filled] = tet->shape_history[filled]->next;
        my_free(dead_shape_inversion);
    }
    /*
     * Otherwise add the new ShapeInversion to the stack.
     */
    else
    {
        new_shape_inversion      = NEW_STRUCT(ShapeInversion);
        new_shape_inversion->wide_angle = tet->coordinate_system;
        new_shape_inversion->next      = tet->shape_history[filled];
        tet->shape_history[filled]      = new_shape_inversion;
    }
}

/*
 * For each of the three complex edge parameters . . .
 */
for (i = 0; i < 3; i++)
{
    /*
     * Copy the ultimate shape to the penultimate.
     */

    tet->shape[filled]->cwl[penultimate][i] = tet->shape[filled]->cwl[ultimate][i];

    /*
     * Copy in the new ultimate shape in rectangular form.
     */

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

    /*
     * Compute the log, using the argument of the previous log
     * to choose the branch (for analytic continuation).
     */

    tet->shape[filled]->cwl[ultimate][i].log = complex_log(
        tet->shape[filled]->cwl[ultimate][i].rect,
        tet->shape[filled]->cwl[penultimate][i].log.imag
    );
}
}
}

```