

```
/*
 * identify_solution_type.c
 *
 * This file provides the function
 *
 * void identify_solution_type(Triangulation *manifold);
 *
 * which identifies the type of solution contained in the
 * tet->shape[filled] structures of the Tetrahedra of Triangulation *manifold,
 * and writes the result to manifold->solution_type[filled]. Possible
 * values are given by the SolutionType enum (see SnapPea.h).
 *
 * Its subroutine
 *
 * Boolean solution_is_degenerate(Triangulation *manifold);
 *
 * Is also available within the kernel, so do_Dehn_filling() can tell
 * whether it is converging towards a degenerate structure.
 */

#include "kernel.h"

/*
 * A solution must have volume at least VOLUME_EPSILON to count
 * as a positive volume solution. Otherwise the volume will be
 * considered zero or negative.
 */

#define VOLUME_EPSILON 1e-2

/*
 * DEGENERACY_EPSILON defines how close a tetrahedron shape must
 * be to zero to count as zero. It is given in logarithmic form.
 * E.g., if DEGENERACY_EPSILON is -6, then the tetrahedron shape
 * (in rectangular form) must lie within a distance  $\exp(-6) = 0.0024\dots$ 
 * of the origin.
 */

#define DEGENERACY_EPSILON -6

/*
 * A solution is considered flat iff it's not degenerate and the
 * argument of each edge parameter is within FLAT_EPSILON of 0.0 or PI.
 */

#define FLAT_EPSILON 1e-2

static Boolean solution_is_flat(Triangulation *manifold);
static Boolean solution_is_geometric(Triangulation *manifold);

void identify_solution_type(
    Triangulation *manifold)
{
    if (solution_is_degenerate(manifold))
    {
        manifold->solution_type[filled] = degenerate_solution;
        return;
    }

    if (solution_is_flat(manifold))
    {
        manifold->solution_type[filled] = flat_solution;
        return;
    }

    if (solution_is_geometric(manifold))
    {
        manifold->solution_type[filled] = geometric_solution;
        return;
    }
}
```

```

    if (volume(manifold, NULL) > VOLUME_EPSILON)
    {
        manifold->solution_type[filled] = nongeometric_solution;
        return;
    }

    manifold->solution_type[filled] = other_solution;
}

Boolean solution_is_degenerate(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i;

    /*
     * If any complex edge parameter of any Tetrahedron is
     * close to zero, return TRUE. Otherwise return FALSE.
     *
     * Note that it's enough to check for shapes close to
     * zero: if an edge parameter is close to one or infinity,
     * then some other edge parameter of the same Tetrahedron
     * will be close to zero.
     */

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

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

            if (tet->shape[filled]->cwl[ultimate][i].log.real < DEGENERACY_EPSILON)

                return TRUE;

    return FALSE;
}

static Boolean solution_is_flat(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    int i;
    double the_angle;

    /*
     * If any edge parameter has angle more than FLAT_EPSILON away
     * from 0.0 or PI, return FALSE. Otherwise, return TRUE.
     */

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

        for (i = 0; i < 3; i++)
        {
            the_angle = tet->shape[filled]->cwl[ultimate][i].log.imag;

            if (fabs(the_angle) > FLAT_EPSILON
                && fabs(the_angle - PI) > FLAT_EPSILON)

                return FALSE;
        }

    return TRUE;
}

static Boolean solution_is_geometric(
    Triangulation *manifold)
{

```

```
Tetrahedron *tet;

/*
 * If any edge parameter has argument less than minus FLAT_EPSILON
 * or greater than PI + FLAT_EPSILON, return FALSE.
 * Otherwise, return TRUE.
 *
 * This allows a solution with some flat tetrahedra to count as geometric.
 * However, if all the tetrahedra were flat, the SolutionType would have
 * been previously identified as flat_solution, and we wouldn't have
 * gotten to this function.
 */

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

    if (tetrahedron_is_geometric(tet) == FALSE)

        return FALSE;

return TRUE;
}

Boolean tetrahedron_is_geometric(
    Tetrahedron *tet)
{
    int i;
    double the_angle;

    /*
     * See comments in solution_is_geometric() above.
     */

    for (i = 0; i < 3; i++)
    {
        the_angle = tet->shape[filled]->cw1[ultimate][i].log.imag;

        if (the_angle < - FLAT_EPSILON
            || the_angle > PI + FLAT_EPSILON)

            return FALSE;
    }

    return TRUE;
}
```