

```

/*
 * interface.c
 *
 * This file contains the following functions, which the
 * user interface uses to read the fields in the Triangulation
 * data structure.
 *
 * char          *get_triangulation_name(Triangulation *manifold);
 * char          set_triangulation_name(Triangulation *manifold, char *new_name);
 * SolutionType  get_complete_solution_type(Triangulation *manifold);
 * SolutionType  get_filled_solution_type(Triangulation *manifold);
 * int           get_num_tetrahedra(Triangulation *manifold);
 * Orientability get_orientability(Triangulation *manifold);
 * int           get_num_cusps(Triangulation *manifold);
 * int           get_num_or_cusps(Triangulation *manifold);
 * int           get_num_nonor_cusps(Triangulation *manifold);
 * int           get_max_singularity(Triangulation *manifold);
 * int           get_num_generators(Triangulation *manifold);
 * void          get_cusp_info( Triangulation *manifold,
 *                               int cusp_index,
 *                               CuspTopology *topology,
 *                               Boolean *is_complete,
 *                               double *m,
 *                               double *l,
 *                               Complex *initial_shape,
 *                               Complex *current_shape,
 *                               int *initial_shape_precision,
 *                               int *current_shape_precision,
 *                               Complex *initial_modulus,
 *                               Complex *current_modulus);
 * FuncResult    set_cusp_info( Triangulation *manifold,
 *                               int cusp_index,
 *                               Boolean *cusp_is_complete,
 *                               double m,
 *                               double l);
 * void          get_holonomy( Triangulation *manifold,
 *                              int cusp_index,
 *                              Complex *meridional_holonomy,
 *                              Complex *longitudinal_holonomy,
 *                              int *meridional_precision,
 *                              int *longitudinal_precision);
 * void          get_tet_shape( Triangulation *manifold,
 *                               int which_tet,
 *                               Boolean *fixed_alignment,
 *                               double *shape_rect_real,
 *                               double *shape_rect_imag,
 *                               double *shape_log_real,
 *                               double *shape_log_imag,
 *                               int *precision_rect_real,
 *                               int *precision_rect_imag,
 *                               int *precision_log_real,
 *                               int *precision_log_imag,
 *                               Boolean *is_geometric);
 * int           get_num_edge_classes(
 *                               Triangulation *manifold,
 *                               int edge_class_order,
 *                               Boolean *greater_than_or_equal);
 *
 * The Triangulation data structure itself, as well as its
 * component data structures, remain private to the kernel.
 *
 * These functions are documented more thoroughly in SnapPea.h.
 */

#include "kernel.h"

static int longest_side(Tetrahedron *tet);

char *get_triangulation_name(
    Triangulation *manifold)
{
    return manifold->name;
}

```

```
void set_triangulation_name(
    Triangulation *manifold,
    char *new_name)
{
    /*
     * Free the old name, if there is one.
     */

    if (manifold->name != NULL)
        my_free(manifold->name);

    /*
     * Allocate space for the new name . . .
     */

    manifold->name = NEW_ARRAY(strlen(new_name) + 1, char);

    /*
     * . . . and copy it in.
     */

    strcpy(manifold->name, new_name);
}

SolutionType get_complete_solution_type(
    Triangulation *manifold)
{
    return manifold->solution_type[complete];
}

SolutionType get_filled_solution_type(
    Triangulation *manifold)
{
    return manifold->solution_type[filled];
}

int get_num_tetrahedra(
    Triangulation *manifold)
{
    return manifold->num_tetrahedra;
}

Orientability get_orientability(
    Triangulation *manifold)
{
    return manifold->orientability;
}

int get_num_cusps(
    Triangulation *manifold)
{
    return manifold->num_cusps;
}

int get_num_or_cusps(
    Triangulation *manifold)
{
    return manifold->num_or_cusps;
}

int get_num_nonor_cusps(
    Triangulation *manifold)
{
    return manifold->num_nonor_cusps;
}
```

```

int get_max_singularity(
    Triangulation *manifold)
{
    Cusp *cusp;
    int m,
        l,
        singularity,
        max_singularity;

    max_singularity = 1;

    for ( cusp = manifold->cusplink_begin.next;
          cusp != &manifold->cusplink_end;
          cusp = cusp->next)
    {
        if (cusp->is_complete == FALSE)
        {
            m = (int) cusp->m;
            l = (int) cusp->l;

            if ( cusp->m == (double) m
                && cusp->l == (double) l)
            {
                singularity = gcd(m, l);

                if (max_singularity < singularity)
                    max_singularity = singularity;
            }
        }
    }

    return max_singularity;
}

int get_num_generators(
    Triangulation *manifold)
{
    return manifold->num_generators;
}

void get_cusp_info(
    Triangulation *manifold,
    int cusp_index,
    CuspTopology *topology,
    Boolean *is_complete,
    double *m,
    double *l,
    Complex *initial_shape,
    Complex *current_shape,
    int *initial_shape_precision,
    int *current_shape_precision,
    Complex *initial_modulus,
    Complex *current_modulus)
{
    Cusp *cusp;

    cusp = find_cusp(manifold, cusp_index);

    /*
     * Write information corresponding to nonNULL pointers.
     */

    if (topology != NULL)
        *topology = cusp->topology;

    if (is_complete != NULL)
        *is_complete = cusp->is_complete;

    if (m != NULL)
        *m = cusp->m;

```

```

    if (l != NULL)
        *l = cusp->l;

    if (initial_shape != NULL)
        *initial_shape = cusp->shape[initial];
    /* = Zero if initial hyperbolic structure is degenerate */

    if (current_shape != NULL)
        *current_shape = cusp->shape[current];
    /* = Zero if cusp is filled or hyperbolic structure is degenerate */

    if (initial_shape_precision != NULL)
        *initial_shape_precision = cusp->shape_precision[initial];
    /* = 0 if initial hyperbolic structure is degenerate */

    if (current_shape_precision != NULL)
        *current_shape_precision = cusp->shape_precision[current];
    /* = 0 if cusp is filled or hyperbolic structure is degenerate */

    if (initial_modulus != NULL)
    {
        if (cusp->shape_precision[initial] > 0)
            *initial_modulus = cusp_modulus(cusp->shape[initial]);
        else
            *initial_modulus = Zero;
    }

    if (current_modulus != NULL)
    {
        if (cusp->shape_precision[current] > 0)
            *current_modulus = cusp_modulus(cusp->shape[current]);
        else
            *current_modulus = Zero;
    }
}

FuncResult set_cusp_info(
    Triangulation *manifold,
    int cusp_index,
    Boolean cusp_is_complete,
    double m,
    double l)
{
    Cusp *cusp;

    cusp = find_cusp(manifold, cusp_index);

    /*
     * Write the given Dehn coefficients into the cusp.
     */

    if (cusp_is_complete)
    {
        cusp->is_complete = TRUE;
        cusp->m = 0.0;
        cusp->l = 0.0;
    }

    else
    {
        /*
         * Check the input.
         *
         * The comment at the top of holonomy.c explains why only
         * (m,0) Dehn fillings are possible on nonorientable cusps.
         */

        if (m == 0.0 && l == 0.0)
        {
            uAcknowledge("Can't do (0,0) Dehn filling.");
            return func_bad_input;
        }
    }
}

```

```

    if (cusp->topology == Klein_cusp && l != 0.0)
    {
        uAcknowledge("Only (p,0) Dehn fillings are possible on a nonorientable cusp.");
        return func_bad_input;
    }

    /*
     * Copy the input into the cusp.
     */

    cusp->is_complete = FALSE;
    cusp->m = m;
    cusp->l = l;

}

return func_OK;
}

void get_holonomy(
    Triangulation *manifold,
    int cusp_index,
    Complex *meridional_holonomy,
    Complex *longitudinal_holonomy,
    int *meridional_precision,
    int *longitudinal_precision)
{
    Cusp *cusp;

    cusp = find_cusp(manifold, cusp_index);

    if (meridional_holonomy != NULL)
        *meridional_holonomy = cusp->holonomy[ultimate][M];

    if (longitudinal_holonomy != NULL)
    {
        *longitudinal_holonomy = cusp->holonomy[ultimate][L];

        /*
         * Longitudes on Klein bottle cusps are stored as their
         * double covers (cf. peripheral_curves.c), so we divide
         * by two to compensate. (Recall that this isn't actually
         * the holonomy, but the log of the holonomy, i.e. the
         * complex length.)
         *
         * As explained at the top of holonomy.c, the holonomy
         * in this case must be real, so we clear any roundoff
         * error in the imaginary part.
         */
        if (cusp->topology == Klein_cusp)
        {
            longitudinal_holonomy->real /= 2.0;
            longitudinal_holonomy->imag = 0.0;
        }
    }

    if (meridional_precision != NULL)
        *meridional_precision = complex_decimal_places_of_accuracy(
            cusp->holonomy[ultimate][M],
            cusp->holonomy[penultimate][M]);

    if (longitudinal_precision != NULL)
        *longitudinal_precision = complex_decimal_places_of_accuracy(
            cusp->holonomy[ultimate][L],
            cusp->holonomy[penultimate][L]);
}

void get_tet_shape(
    Triangulation *manifold,
    int which_tet,
    Boolean fixed_alignment,
    double *shape_rect_real,

```

```

double      *shape_rect_imag,
double      *shape_log_real,
double      *shape_log_imag,
int         *precision_rect_real,
int         *precision_rect_imag,
int         *precision_log_real,
int         *precision_log_imag,
Boolean     *is_geometric)
{
    int      count,
            the_coordinate_system;
    Tetrahedron *tet;
    ComplexWithLog *ultimate_shape,
                *penultimate_shape;

    /*
     * If no solution is present, return all zeros.
     */

    if (manifold->solution_type[filled] == not_attempted)
    {
        *shape_rect_real      = 0.0;
        *shape_rect_imag     = 0.0;
        *shape_log_real      = 0.0;
        *shape_log_imag     = 0.0;

        *precision_rect_real  = 0;
        *precision_rect_imag  = 0;
        *precision_log_real   = 0;
        *precision_log_imag   = 0;

        *is_geometric        = FALSE;

        return;
    }

    /*
     * Check that which_tet is within bounds.
     */

    if (which_tet < 0 || which_tet >= manifold->num_tetrahedra)
        uFatalError("get_tet_shape", "interface");

    /*
     * Find the Tetrahedron in position which_tet.
     */

    for (tet = manifold->tet_list_begin.next, count = 0;
         tet != &manifold->tet_list_end && count != which_tet;
         tet = tet->next, count++)
        ;

    /*
     * If we went all the way through the list of Tetrahedra
     * without finding position which_tet, then something
     * is very wrong.
     */

    if (tet == &manifold->tet_list_end)
        uFatalError("get_tet_shape", "interface");

    /*
     * If fixed_alignment is TRUE, use a fixed coordinate system.
     * Otherwise choose the_coordinate_system so that the longest side
     * of the triangle is the initial side of the angle.
     */

    if (fixed_alignment == TRUE)
        the_coordinate_system = 0;
    else
        the_coordinate_system = (longest_side(tet) + 1) % 3;

    /*
     * Note the addresses of the ultimate and penultimate shapes.

```

```

    */

    ultimate_shape      = &tet->shape[filled]->cwl[ ultimate ][the_coordinate_system];
    penultimate_shape = &tet->shape[filled]->cwl[penultimate][the_coordinate_system];

    /*
    *   Report the ultimate shapes.
    */

    *shape_rect_real = ultimate_shape->rect.real;
    *shape_rect_imag = ultimate_shape->rect.imag;
    *shape_log_real  = ultimate_shape->log.real;
    *shape_log_imag  = ultimate_shape->log.imag;

    /*
    *   Estimate the precision.
    */

    *precision_rect_real = decimal_places_of_accuracy(ultimate_shape->rect.real,
    penultimate_shape->rect.real);
    *precision_rect_imag = decimal_places_of_accuracy(ultimate_shape->rect.imag,
    penultimate_shape->rect.imag);
    *precision_log_real  = decimal_places_of_accuracy(ultimate_shape->log.real,
    penultimate_shape->log.real);
    *precision_log_imag  = decimal_places_of_accuracy(ultimate_shape->log.imag,
    penultimate_shape->log.imag);

    /*
    *   Check whether the tetrahedron is geometric.
    */

    *is_geometric = tetrahedron_is_geometric(tet);
}

static int longest_side(
    Tetrahedron *tet)
{
    int      i,
            desired_index;
    double   sine[3],
            max_sine;

    /*
    *   longest_side() returns the index (0, 1 or 2) of the edge opposite
    *   the longest side of tet's triangular vertex cross section.
    *
    *   We'll use the Law of Sines, which says that the lengths of a triangle's
    *   sides are proportional to the sines of the opposite angles.
    *
    *   We take the absolute value of each sine, just in case the
    *   Tetrahedron is negatively oriented.
    */

    for (i = 0; i < 3; i++)
        sine[i] = fabs(tet->shape[filled]->cwl[ultimate][i].rect.imag) /
            complex_modulus(tet->shape[filled]->cwl[ultimate][i].rect);

    max_sine = -1.0;
    for (i = 0; i < 3; i++)
        if (sine[i] > max_sine)
        {
            max_sine      = sine[i];
            desired_index  = i;
        }

    return desired_index;
}

int get_num_edge_classes(
    Triangulation *manifold,
    int            edge_class_order,
    Boolean        greater_than_or_equal)

```

```
{
    int      count;
    EdgeClass *edge;

    count = 0;

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)

        if (    greater_than_or_equal ?
              edge->order >= edge_class_order :
              edge->order == edge_class_order)

            count++;

    return count;
}
```