

```

/*
 * triangulation_io.h
 *
 * The question of file formats is tricky. Typically an application has a
 * different file format for each platform it runs on (Mac, Windows, etc.),
 * perhaps with some support for translating between them. Certainly
 * the easiest way to write Macintosh SnapPea would have been to use
 * the THINK Class Library's built-in file handling capabilities:
 * file i/o would have been trivial to program, and each kind of file
 * (triangulation, generators, link projection) would have its own
 * "file type" and icon. However, I chose to use straight ASCII text
 * files for the following reasons:
 *
 * (1) The files can be read and written on any platform.
 *     For example, a Mac user can send a Triangulation file
 *     to a friend using a unix version of SnapPea.
 *
 * (2) Human beings can read the files using any text editor.
 *
 * (3) Text-only files are best for people doing other programming
 *     projects which use SnapPea's manifolds as input. (Again,
 *     people do this work on a variety of platforms.)
 *
 * On the other hand, I think it would be a mistake to hard-code the
 * SnapPea kernel to read data from unix-style FILES. When the UI
 * reads a triangulation file, it passes the data to the kernel using
 * the data structures defined below. And, of course, when the UI wants
 * to write a triangulation to a file, it requests the data from the
 * kernel in this same format.
 *
 * Notes:
 *
 * (1) This format is similar to that of the Triangulation data structure.
 *     However, the present format is available to the UI, while
 *     the Triangulation data structure is private to the kernel.
 *
 * (2) The new file format is not backward compatible to the old
 *     file format, although they are similar. (I couldn't keep
 *     the old file format because it doesn't work properly with
 *     nonorientable manifolds.) SnapPea 2.0 will read (but not write)
 *     the old file format, with the exception that peripheral curves
 *     on nonorientable manifolds are recomputed from scratch.
 *
 * (3) If you (or your program) is writing TriangulationData structures
 *     from scratch, note that not all fields are required. The fields
 *     solution_type, volume, and the TetrahedronData's filled_shapes
 *     are ignored by data_to_triangulation(), because it recomputes the
 *     hyperbolic structure from scratch. The meridian and longitude
 *     are optional: if you provide them, data_to_triangulation()
 *     will use them; otherwise it provides a default basis.
 *     (In the latter case, of course, you shouldn't specify any
 *     Dehn fillings, because you aren't providing the basis relative
 *     to which they are defined.)
 *
 * 96/9/17 If you set both num_or_cusps and num_nonor_cusps to zero,
 * data_to_triangulation() will create the Cusps for you. It will
 * also create the peripheral curves.
 *
 * 96/9/17 If you specify unknown_orientability, data_to_triangulation()
 * will attempt to orient the manifold. This will typically change
 * the indexing of the Tetrahedra's vertices.
 */

/*
 * This file (triangulation_io.h) is intended solely for inclusion
 * in SnapPea.h. It depends on some of the typedefs there.
 */

typedef struct TriangulationData    TriangulationData;
typedef struct CuspData            CuspData;
typedef struct TetrahedronData     TetrahedronData;

struct TriangulationData
{

```

```
    char          *name;
    int           num_tetrahedra;
    SolutionType   solution_type;
    double        volume;
    Orientability  orientability;
    Boolean        CS_value_is_known;
    double        CS_value;
    int           num_or_cusps,
                num_nonor_cusps;
    CuspData      *cusp_data;
    TetrahedronData *tetrahedron_data;
};

struct CuspData
{
    CuspTopology   topology;
    double        m,
                l;
};

struct TetrahedronData
{
    /*
     * Note:  gluing[i][j] is the image of j under the i-th gluing permutation.
     */
    int           neighbor_index[4];
    int           gluing[4][4];
    int           cusp_index[4];
    int           curve[2][2][4][4];
    Complex        filled_shape;
};
```