

```

/*
 * kernel_prototypes.h
 *
 * This file contains prototypes for functions which are externally
 * available within the kernel, but not available to the user-interface.
 */

#ifndef _kernel_prototypes_
#define _kernel_prototypes_

#include "SnapPea.h"
#include "positioned_tet.h"

/*****
 *
 * chern_simons.c
 *
 *****/

extern void compute_CS_value_from_fudge(Triangulation *manifold);
extern void compute_CS_fudge_from_value(Triangulation *manifold);
/*
 * Compute the Chern-Simons value in terms of the fudge factor, and
 * vice-versa. Please see chern_simons.c for details.
 */

/*****
 *
 * choose_generators.c
 *
 *****/

extern void choose_generators( Triangulation *manifold,
                               Boolean compute_corners,
                               Boolean centroid_at_origin);
/*
 * Chooses a set of generators for the fundamental group of the
 * Triangulation *manifold. Various functions which use generators all
 * call choose_generators(), so they are sure to be using the same
 * generator set, and their results are directly comparable.
 * If compute_corners is TRUE, choose_generators() computes the location
 * on the sphere at infinity of each ideal vertex of each Tetrahedron,
 * using the hyperbolic structure of the Dehn filled manifold.
 * If centroid_at_origin is TRUE, the initial tetrahedron is positioned
 * with its centroid at the origin; otherwise the initial tetrahedron
 * is positioned with its vertices at {0, 1, infinity, z}.
 * If compute_corners is FALSE, centroid_at_origin is ignored.
 */

void compute_fourth_corner(
    Complex corner[4],
    VertexIndex missing_corner,
    Orientation orientation,
    ComplexWithLog cwl[3]);
/*
 * Given the location on the sphere at infinity of three of a Tetrahedron's
 * ideal vertices, compute the location of the fourth.
 */

/*****
 *
 * close_cusps.c
 *
 *****/

extern void close_cusps(Triangulation *manifold, Boolean fill_cusp[]);
/*
 * Permanently closes the cusps of *manifold for which fill_cusp[culp->index]
 * is TRUE. Assumes *manifold is triangulated as in subdivide().
 */

```

```

/*****
/*
/*                      complex.c                      */
/*
/*****

extern Complex  Zero,
                One,
                Two,
                Four,
                MinusOne,
                I,
                TwoPiI,
                Infinity;

/*****
/*
/*                      core_geodesics.c                */
/*
/*****

extern void compute_core_geodesic(  Cusp      *cusp,
                                   int       *singularity_index,
                                   Complex length[2]);

/*
 * This function is similar to the function core_geodesic() defined in
 * SnapPea.h, only it accepts a Cusp pointer as input and returns
 * the complex length relative to the ultimate and penultimate
 * hyperbolic structures (rather than reporting a precision).
 */

/*****
/*
/*                      cusps.c                        */
/*
/*****

extern void create_cusps(Triangulation *manifold);

/*
 * Creates Cusp data structures for a Triangulation with valid
 * neighbor and gluing fields (and perhaps nothing else).
 */

extern void error_check_for_create_cusps(Triangulation *manifold);

/*
 * Checks that no Cusps are present, and that all tet->cusp[]
 * fields are NULL.
 */

extern void create_one_cusp(Triangulation *manifold, Tetrahedron *tet,
                           Boolean is_finite, VertexIndex v, int cusp_index);

/*
 * Creates a single Cusp, incident to the given Tetrahedron at
 * the given ideal vertex. Assumes ideal vertices which haven't
 * been assigned to Cusps have tetrahedron->cusp[vertex] == NULL.
 */

extern void create_fake_cusps(Triangulation *manifold);

/*
 * Creates Cusp data structures for the "fake cusps" corresponding to
 * finite vertices.
 */

extern void count_cusps(Triangulation *manifold);

/*
 * counts the Cusps of each CuspTopology, and sets manifold->num_cusps,
 * manifold->num_or_cusps and manifold->num_nonor_cusps.
 */

extern Boolean mark_fake_cusps(Triangulation *manifold);
/*

```

```

* Distinguishes real cusps from fake cusps ( = finite vertices) by
* computing the Euler characteristic. Sets is_finite to TRUE for
* fake cusps, and renumbers all cusps so that real cusps have
* consecutive nonnegative indices beginning at 0 and fake cusps
* have consecutive negative indices beginning at -1.
*/

/*****
/*
/*          cusp_cross_sections.c
/*
/*
/*****

extern void allocate_cross_sections(Triangulation *manifold);
/*
* Allocates a VertexCrossSections structure for each tet->cross_section
* in manifold.
*/

extern void free_cross_sections(Triangulation *manifold);
/*
* Frees the VertexCrossSections structure and resets the
* tet->cross_section pointer to NULL for each tet in manifold.
*/

extern void compute_cross_sections(Triangulation *manifold);
/*
* Sets the (already allocated) VertexCrossSections to correspond to
* cusp cross sections bounding equal volumes. In general these cross
* sections will NOT represent maximal disjoint horoball neighborhoods
* of the cusps.
*/

extern void compute_tilts(Triangulation *manifold);
/*
* Applies the Tilt Theorem to compute the tilts from the
* VertexCrossSections. Assumes the TetShapes are correct.
*/

extern void compute_three_edge_lengths( Tetrahedron *tet,
                                       VertexIndex v,
                                       FaceIndex f,
                                       double known_length);
/*
* Sets tet->cross_section->edge_length[v][f] to known_length, computes
* the remaining two edge_lengths at vertex v in terms of it, and sets
* the has_been_set flag to TRUE.
*/

extern void compute_tilts_for_one_tet(Tetrahedron *tet);
/*
* Applies the Tilt Theorem to compute the tilts from the
* VertexCrossSections. Assumes the TetShapes are correct.
*/

/*****
/*
/*          cusp_neighborhoods.c
/*
/*
/*****

extern void cn_find_third_corner( Tetrahedron *tet,
                                Orientation h,
                                VertexIndex v,
                                FaceIndex f0,
                                FaceIndex f1,
                                FaceIndex f2);
/*
* Given the locations of corners f0 and f1 on the given triangle,
* compute and record the location of corner f2.
* This function is in spirit local to cusp_neighborhoods.c, but we
* want to make it available to the 2-3 and 3-2 simplifications

```

```

*   in simplify_triangulation.c.
*/

/*****
/*
/*                               */
/*               cusp_shapes.c   */
/*                               */
/*                               */
*****/

extern void compute_cusp_shapes(Triangulation *manifold,
                               FillingStatus  which_structure);
/*
*   Computes the shape of each unfilled cusp and stores the result in
*   cusp->cusp_shape[which_structure]. (which_structure = initial or
*   current) Stores the number of decimal places of accuracy in
*   cusp->shape_precision[which_structure].
*/

/*****
/*
/*                               */
/*               Dehn_coefficients.c   */
/*                               */
/*                               */
*****/

extern Boolean all_De hn_coefficients_are_integers(Triangulation *manifold);
/*
*   Returns FALSE if some cusp has noninteger Dehn filling coefficients.
*   Returns TRUE if each cusp is either unfilled, or has integer
*   Dehn filling coefficients.
*/

extern Boolean Dehn_coefficients_are_integers(Cusp *cusp);
/*
*   Returns FALSE if the Dehn filling coefficients of *cusp are
*   nonintegers.
*   Returns TRUE if *cusp is unfilled, or has integer Dehn filling
*   coefficients.
*/

extern Boolean all_De hn_coefficients_are_relatively_prime_integers(
                               Triangulation *manifold);
extern Boolean Dehn_coefficients_are_relatively_prime_integers(Cusp *cusp);
/*
*   Same as above, but integer coefficients must be relatively prime.
*/

extern Boolean all_cusps_are_complete(Triangulation *manifold);
extern Boolean all_cusps_are_filled(Triangulation *manifold);
/*
*   Returns TRUE if all cusps are complete (resp. filled), FALSE otherwise.
*/

/*****
/*
/*                               */
/*               direct_product.c   */
/*                               */
/*                               */
*****/

extern Boolean is_group_direct_product(SymmetryGroup *the_group);
/*
*   Checks whether the_group is a nonabelian, nontrivial direct product,
*   and sets it is_direct_product and factor[] fields accordingly.
*/

/*****
/*
/*                               */
/*               edge_classes.c   */
/*                               */
/*                               */
*****/

```

```

extern void create_edge_classes(Triangulation *manifold);
/*
 * Adds EdgeClasses to a partially constructed manifold which does not
 * yet have them.
 */

extern void replace_edge_classes(Triangulation *manifold);
/*
 * Removes all EdgeClasses from a manifold and adds fresh ones.
 */

extern void orient_edge_classes(Triangulation *manifold);
/*
 * Orients a neighborhood of each EdgeClass, and fills in the fields
 * tet->edge_class[] to described how the EdgeClass views each
 * incident Tetrahedron.
 */

/*****
 *
 * elements_generate_group.c
 *
 *****/

extern Boolean elements_generate_group(SymmetryGroup *the_group,
    int num_possible_generators, int possible_generators[]);
/*
 * The array possible_generators[] contains num_possible_generators
 * elements of the_group. Do these elements generate the_group?
 */

/*****
 *
 * find_cusp.c
 *
 *****/

extern Cusp *find_cusp(Triangulation *manifold, int cusp_index);
/*
 * Converts a cusp_index to a Cusp pointer.
 */

/*****
 *
 * finite_vertices.c
 *
 *****/

extern void remove_finite_vertices(Triangulation *manifold);
/*
 * Removes finite vertices from the manifold.
 */

/*****
 *
 * gcd.c
 *
 *****/

extern long int gcd(long int a, long int b);
/*
 * Returns the greatest common divisor of two nonnegative long integers,
 * at least one of which is nonzero.
 */

extern long int euclidean_algorithm(long int m, long int n,
    long int *a, long int *b);
/*
 * Returns the greatest common divisor of two long integers m and n,
 * and also finds long integers a and b such that am + bn = gcd(m,n).
 */

```

```

*   The integers m and n may be negative, but may not both be zero.
*/

extern long int Zq_inverse(long int p, long int q);
/*
*   Returns the inverse of p in the ring Z/q. Assumes p and q are
*   relatively prime integers satisfying  $0 < p < q$ .
*/

/*****
/*
/*                               */
/*           gluing_equations.c   */
/*                               */
/*                               */
*****/

extern void compute_gluing_equations(Triangulation *manifold);
/*
*   compute_gluing_equations() computes the complex gluing equations
*   if Triangulation *manifold is orientable, and the real gluing
*   equations if it's nonorientable. It assumes that space for the
*   appropriate set of equations has already been assigned to the cusps
*   and edges.
*/

/*****
/*
/*                               */
/*           holonomy.c           */
/*                               */
/*                               */
*****/

extern void compute_holonomies(Triangulation *manifold);
/*
*   Computes the log of the holonomy of each cusp based on the current
*   shapes of the tetrahedra, and stores the results into the
*   holonomy[ultimate][ ] field of the Cusp data structure. The previous
*   contents of that field are transferred to holonomy[penultimate][ ].
*
*   compute_holonomies() is called whenever a hyperbolic structure is
*   computed. Therefore if you have a manifold with a hyperbolic structure
*   you may assume correct values of the holonomy are already in place.
*/

extern void compute_the_holonomies( Triangulation  *manifold,
                                   Ultimateness   which_iteration);
/*
*   Computes the holonomies for either the ultimate or penultimate
*   solution, according to the value of which_iteration.
*/

extern void compute_edge_angle_sums(Triangulation *manifold);
/*
*   For each EdgeClass, computes the sum of the logs of the complex
*   edge parameters and records the result in the edge_angle_sum field.
*   compute_edge_angle_sums() is used in finding the hyperbolic structure;
*   once the hyperbolic structure is found, each edge_angle_sum will of
*   course be  $2\pi i$ .
*/

/*****
/*
/*                               */
/*           hyperbolic_structure.c
/*                               */
/*                               */
*****/

extern void remove_hyperbolic_structures(Triangulation *manifold);
/*
*   Frees the TetShapes (if any) pointed to by each tet->shape[] and sets
*   manifold->solution_type[complete] and manifold->solution_type[filled]
*   to not_attempted.
*/

```

```

extern void initialize_tet_shapes(Triangulation *manifold);
/*
 * Sets all Tetrahedra to be regular ideal tetrahedra. Allocates the
 * TetShapes if necessary. Clears the shape_histories if necessary.
 */

extern void polish_hyperbolic_structures(Triangulation *manifold);
/*
 * Attempts to increase the accuracy of both the complete and the Dehn
 * filled hyperbolic structures already present in *manifold. It's
 * designed to be called following retriangulation operations which
 * diminish the accuracy of the TetShapes.
 */

extern void copy_solution(Triangulation *manifold, FillingStatus source, FillingStatus dest);
extern void complete_all_cusps(Triangulation *manifold);
/*
 * These are low-level routines used mainly within hyperbolic_structure.c,
 * but also to transfer the Chern-Simons invariant in drilling.c.
 */

/*****
/*
 *                               identify_solution_type.c
/*
 *                               *****/

extern void identify_solution_type(Triangulation *manifold);
/*
 * 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.
 */

extern Boolean solution_is_degenerate(Triangulation *manifold);
/*
 * Returns TRUE if any TetShape is close to {0, 1, infinity}.
 * Otherwise returns FALSE.
 */

extern Boolean tetrahedron_is_geometric(Tetrahedron *tet);
/*
 * Returns TRUE if all tetrahedra are geometric; returns FALSE otherwise.
 * A tetrahedron is geometric iff all dihedral angles lie in the
 * range [-FLAT_EPSILON, pi + FLAT_EPSILON].
 */

/*****
/*
 *                               intersection_numbers.c
/*
 *                               *****/

extern void compute_intersection_numbers(Triangulation *manifold);
/*
 * Computes the intersection numbers of the curves stored
 * in the scratch_curve[][][][] fields of the Tetrahedra, and
 * writes the results to the intersection_number[][] fields of
 * the Cusps. Please see intersection_numbers.c for details.
 */

extern void copy_curves_to_scratch( Triangulation *manifold,
                                   int which_set,
                                   Boolean double_copy_on_tori);
/*
 * Copies the current peripheral curves to the scratch_curves[which_set]
 * fields of the manifold's Tetrahedra. If double_copy_on_tori is TRUE,
 * it copies peripheral curves on orientable cusps to both sheets of
 * the Cusps' orientation double covers.
 */

```

```

/*****
/*
/*          isometry_closed.c
/*
/*
/*****

extern FuncResult compute_closed_isometry(  Triangulation  *manifold0,
                                           Triangulation  *manifold1,
                                           Boolean        *are_isometric);

/*
*   If it determines with absolute rigor that manifold0 and manifold1 are
*   isometric, sets *are_isometric to TRUE and returns func_OK.
*   If it determines with absolute rigor that manifold0 and manifold1 are
*   nonhomeomorphic, sets *are_isometric to FALSE and returns func_OK.
*   If it fails to decide, returns func_failed.
*/

/*****
/*
/*          isometry_cusped.c
/*
/*
/*****

extern FuncResult compute_cusped_isometries(
                                           Triangulation  *manifold0,
                                           Triangulation  *manifold1,
                                           IsometryList   **isometry_list,
                                           IsometryList   **isometry_list_of_links);

/*
*   Finds all isometries from manifold0 to manifold1 (ignoring Dehn
*   fillings), stores them in an IsometryList data structure, and sets
*   isometry_list to point to it.  If isometry_list_of_links is not NULL,
*   it copies those Isometries which extend to Isometries of the associated
*   links (i.e. those which take meridians to plus-or-minus meridians)
*   onto a separate list, and stores its address in *isometry_list_of_links.
*   If manifold0 == manifold1, this function is finding all symmetries.
*/

/*****
/*
/*          Moebius_transformations.c
/*
/*
/*****

extern CONST MoebiusTransformation  Moebius_identity;

extern void      Moebius_copy(      MoebiusTransformation  *dest,
                                   MoebiusTransformation  *source);
extern void      Moebius_invert(    MoebiusTransformation  *mt,
                                   MoebiusTransformation  *mt_inverse);
extern void      Moebius_product(   MoebiusTransformation  *a,
                                   MoebiusTransformation  *b,
                                   MoebiusTransformation  *product);

/*
*   These functions do what you would expect.
*/

/*****
/*
/*          my_malloc.c
/*
/*
/*****

extern void *my_malloc(size_t bytes);

/*
*   Calls malloc() to request a block of memory of size "bytes".
*   If successful, returns a pointer to the memory.
*   If unsuccessful, calls uAcknowledge() to notify the user, then exits.
*/

```



```

extern void my_free(void *ptr);
/*
 * Calls free() to deallocate the block of memory pointed to by ptr.
 */

extern int malloc_calls(void);
/*
 * Returns the number of calls to my_malloc() minus the number of
 * calls to my_free(), for use in debugging.
 */

/*****
 *
 * normal_surface_construction.c
 *
 *****/

extern void recognize_embedded_surface( Triangulation *manifold,
                                       Boolean *connected,
                                       Boolean *orientable,
                                       Boolean *two_sided,
                                       int *Euler_characteristic);
/*
 * Reports the connectedness, orientability, two-sidedness and Euler
 * characteristic of the normal surface described in the parallel_edge,
 * num_squares and num_triangles fields of the manifold's Tetrahedra.
 * The present implementation assumes the manifold has no filled cusps.
 */

/*****
 *
 * o3l_matrices.c
 *
 *****/

extern O3lMatrix O3l_identity;

extern void o3l_copy(O3lMatrix dest, O3lMatrix source);
extern void o3l_invert(O3lMatrix m, O3lMatrix m_inverse);
extern FuncResult gl4R_invert(GL4RMatrix m, GL4RMatrix m_inverse);
extern void o3l_product(O3lMatrix a, O3lMatrix b, O3lMatrix product);
extern Boolean o3l_equal(O3lMatrix a, O3lMatrix b, double epsilon);
extern double o3l_deviation(O3lMatrix m);
extern void o3l_GramSchmidt(O3lMatrix m);
extern void o3l_conjugate(O3lMatrix m, O3lMatrix t, O3lMatrix Tmt);
extern double o3l_inner_product(O3lVector u, O3lVector v);
extern void o3l_matrix_times_vector(O3lMatrix m, O3lVector v, O3lVector product);
extern void o3l_constant_times_vector(double r, O3lVector v, O3lVector product);
extern void o3l_copy_vector(O3lVector dest, O3lVector source);
extern void o3l_vector_sum(O3lVector a, O3lVector b, O3lVector sum);
extern void o3l_vector_diff(O3lVector a, O3lVector b, O3lVector diff);
/*
 * These functions all do what you would expect.
 * o3l_conjugate() replaces m with (t^-1) m t.
 */

/*****
 *
 * orient.c
 *
 *****/

extern void orient(Triangulation *manifold);
/*
 * Attempts to consistently orient the Tetrahedra of the
 * Triangulation *manifold. Sets manifold->orientability to
 * oriented_manifold or nonorientable_manifold, as appropriate.
 */

extern void extend_orientation( Triangulation *manifold,

```

```

                                Tetrahedron    *initial_tet);
/*
 * Extends the orientation of the given initial_tet to the entire manifold.
 */

extern void fix_peripheral_orientations(Triangulation *manifold);
/*
 * Makes sure each {meridian, longitude} pairs obeys the right-hand rule.
 * Should be called only for orientable manifolds, typically following
 * a call to orient().
 */

/*****
 *
 *                                peripheral_curves.c
 *
 *****/

extern void peripheral_curves(Triangulation *manifold);
/*
 * Puts a meridian and longitude on each cusp, and records each cusp's
 * CuspTopology in the field cusp->topology. If the manifold is
 * oriented, the meridian and longitude adhere to the usual
 * orientation convention (see peripheral_curves.c for details).
 */

extern void peripheral_curves_as_needed(Triangulation *manifold);
/*
 * Like peripheral_curves(), but puts a meridian and longitude
 * only onto cusps which don't already have them. Pre-existing
 * meridians and longitudes are left untouched.
 */

/*****
 *
 *                                polyhedral_group.c
 *
 *****/

extern Boolean is_group_polyhedral(SymmetryGroup *the_group);
/*
 * Checks whether the_group is a polyhedral group (i.e. (binary) dihedral,
 * tetrahedral, octahedral or icosahedral), and fills in the_group's
 * is_polyhedral, is_full_group, p, q and r fields accordingly. Assumes
 * the_group's is_dihedral field has already been set (but this
 * restriction could be eliminated -- see the documentation at the top
 * of triangle_group.c).
 */

/*****
 *
 *                                positioned_tet.c
 *
 *****/

extern void veer_left(PositionedTet *ptet);
extern void veer_right(PositionedTet *ptet);
/*
 * Accepts a PositionedTet and replaces it with the neighboring
 * tetrahedron incident to the left (resp. right) face. The positioning
 * is as if you rotated the old tetrahedron onto the new one about the
 * axis defined by the edge of the old tetrahedron lying between
 * near_face and left_face (resp. right_face).
 */

extern void veer_backwards(PositionedTet *ptet);
/*
 * Accepts a PositionedTet and replaces it with the neighboring
 * tetrahedron incident to the near_face. The positioning
 * is as if you rotated the old tetrahedron onto the new one
 * about an axis running up the center of the near_face from
 * the bottom_face to the ideal vertex opposite the bottom_face.
 */

```

```

*/

extern Boolean same_positioned_tet(PositionedTet *ptet0, PositionedTet *ptet1);
/*
 * Returns TRUE if the two PositionedTets are equal, FALSE otherwise.
 */

extern void set_left_edge(EdgeClass *edge, PositionedTet *ptet);
/*
 * Fills in the fields of *ptet so as to position the EdgeClass *edge
 * between the near_face and the left_face of *ptet.
 */

/*****
/*
 *                               */
/*                               */
/*                               */
/*                               */
*****/

extern int decimal_places_of_accuracy(double x, double y);
extern int complex_decimal_places_of_accuracy(Complex x, Complex y);
/*
 * Returns the number of decimal places which x and y have in
 * common. Typically x and y will be two estimates of the same
 * computed quantity (e.g. the volume of a manifold), and
 * decimal_places_of_accuracy() will be used to tell the user
 * interface how many decimal places should be printed.
 */

/*****
/*
 *                               */
/*                               */
/*                               */
/*                               */
*****/

/*
 * The high-level functions basic_simplification() and
 * randomize_triangulation() are declared in SnapPea.h.
 */

extern FuncResult cancel_tetrahedra(EdgeClass *edge, EdgeClass **where_to_resume, int *
    num_tetrahedra_ptr);
extern FuncResult three_to_two(EdgeClass *edge, EdgeClass **where_to_resume, int *
    num_tetrahedra_ptr);
extern FuncResult two_to_three(Tetrahedron *tet0, FaceIndex f, int *num_tetrahedra_ptr);
extern void one_to_four(Tetrahedron *tet, int *num_tetrahedra_ptr, int
    new_cusp_index);
/*
 * Low-level functions for
 *
 *   cancelling two Tetrahedra which share a common edge of order 2,
 *
 *   replacing three Tetrahedra surrounding a common edge with
 *       two Tetrahedra sharing a common face
 *
 *   replacing two Tetrahedra sharing a common face with
 *       three Tetrahedra surrounding a common edge
 *
 *   replacing a Tetrahedron with four Tetrahedra meeting a point.
 *
 * If an operation cannot be performed because of a topological or geometric
 * obstruction, the function does nothing and returns func_failed.
 * Otherwise, it performs the operation and returns func_OK.
 *
 * The function one_to_four() will always succeed, and therefore returns void.
 * It introduces a finite vertex at the center of the Tetrahedron, and therefore
 * cannot be used when a hyperbolic structure is present.
 *
 * The three_to_two(), two_to_three() and one_to_four() operations each correspond
 * to a projection of a 4-simplex.
 *
 * For further details, see the documentation in simplify_triangulation.c,

```

```

* both at the top of file and immediately preceding each function.
*/

/*****
/*
/*                               sl2c_matrices.c                               */
/*
/*                               */
*****/

extern void    sl2c_copy(SL2CMatrix dest, CONST SL2CMatrix source);
extern void    sl2c_invert(CONST SL2CMatrix a, SL2CMatrix inverse);
extern void    sl2c_complex_conjugate(CONST SL2CMatrix a, SL2CMatrix conjugate);
extern void    sl2c_product(CONST SL2CMatrix a, CONST SL2CMatrix b, SL2CMatrix product);
extern void    sl2c_adjoint(CONST SL2CMatrix a, SL2CMatrix adjoint);
extern void    sl2c_normalize(SL2CMatrix a);
extern Boolean sl2c_matrix_is_real(CONST SL2CMatrix a);

/*****
/*
/*                               solve_equations.c                               */
/*
/*                               */
*****/

extern FuncResult solve_complex_equations(Complex **complex_equations,
                                           int num_rows, int num_columns, Complex *solution);
extern FuncResult solve_real_equations(double **real_equations,
                                         int num_rows, int num_columns, double *solution);
/*
* These functions solve num_rows linear equations in num_columns
* variables. For more information, see solve_equations.c.
*/

/*****
/*
/*                               subdivide.c                               */
/*
/*                               */
*****/

extern Triangulation *subdivide(Triangulation *manifold, char *new_name);
/*
* Returns a pointer to a subdivision of *manifold. See subdivide.c for
* more details. subdivide() does not change *manifold in any way.
*/

/*****
/*
/*                               symmetric_group.c                               */
/*
/*                               */
*****/

extern Boolean is_group_S5(SymmetryGroup *the_group);
/*
* Is the_group S5?
*/

/*****
/*
/*                               symmetry_group_cusped.c                               */
/*
/*                               */
*****/

extern FuncResult compute_cusped_symmetry_group(
    Triangulation *manifold,
    SymmetryGroup **symmetry_group_of_manifold,
    SymmetryGroup **symmetry_group_of_link);
/*
* Computes the SymmetryGroup of a cusped manifold, and also the
* SymmetryGroup of the corresponding link (defined at the top of
* symmetry_group_cusped.c).

```

```

*/

extern void compute_orders_of_elements(SymmetryGroup *the_group);
/*
 * Assumes the_group->order and the_group->product[][] are already
 * in place, and computes the_group->order_of_element[].
 */

extern void compute_inverses(SymmetryGroup *the_group);
/*
 * Assumes the_group->order and the_group->product[][] are already
 * in place, and computes the_group->inverse[].
 */

extern void recognize_group(SymmetryGroup *the_group);
/*
 * Attempts to recognize the_group as abelian, cyclic, dihedral,
 * polyhedral or a direct product.
 */

/*****
/*
/*          symmetry_group_closed.c          */
/*
/*          */
*****/

extern FuncResult compute_closed_symmetry_group(
    Triangulation    *manifold,
    SymmetryGroup    **symmetry_group,
    Triangulation    **symmetric_triangulation,
    Boolean          *is_full_group);
/*
 * Attempts to compute the symmetry group of a closed manifold.
 * Also provides a symmetry Dehn filling description.
 */

/*****
/*
/*          terse_triangulation.c          */
/*
/*          */
*****/

extern TerseTriangulation *alloc_terse(int num_tetrahedra);
/*
 * Allocates a TerseTriangulation.
 */

extern TerseTriangulation *tri_to_terse_with_base(
    Triangulation    *manifold,
    Tetrahedron      *base_tetrahedron,
    Permutation       base_permutation);
/*
 * Similar to tri_to_terse(), but allows an explicit choice
 * of base tetrahedron.
 */

/*****
/*
/*          tet_shapes.c          */
/*
/*          */
*****/

extern void add_edge_angles( Tetrahedron *tet0, EdgeIndex e0,
    Tetrahedron *tet1, EdgeIndex e1,
    Tetrahedron *tet2, EdgeIndex e2);
/*
 * Add the angles of edge e0 of tet0 and edge e1 of tet1 and writes the
 * results to edge e2 of tet2. Accounts for edge_orientations. The
 * EdgeIndices should be in the range 0-5, not 0-2. Chooses arguments
 * in the range  $[(-1/2)\pi, (3/2)\pi]$ , regardless of the angles of the
 * summands.

```

```

*/

extern Boolean angles_sum_to_zero(Tetrahedron *tet0, EdgeIndex e0,
                                Tetrahedron *tet1, EdgeIndex e1);
/*
 * angles_sum_to_zero() returns TRUE iff one of the angles
 * (shape[complete]->cwl[ultimate] or shape[filled]->cwl[ultimate])
 * at edge e0 of tet0 cancels the corresponding angle at edge e1
 * of tet1 (mod 2 pi).  Accounts for edge_orientations.
 */

extern void compute_remaining_angles(Tetrahedron *tet, EdgeIndex e);
/*
 * Assumes the angle at edge e is correct, and computes the remaining
 * angles in terms of it.  Chooses arguments in the range  $[(-1/2)\pi, (3/2)\pi]$ .
 */

/*****
/*
/*          tidy_peripheral_curves.c          */
/*
/*          */
/*****/

extern void tidy_peripheral_curves(Triangulation *manifold);
/*
 * Replaces the existing peripheral curves on *manifold with an
 * equivalent set which is efficient and contains no trivial loops.
 */

/*****
/*
/*          transcendentals.c          */
/*
/*          */
/*****/

extern double safe_acos(double x);
extern double safe_asin(double x);
extern double safe_sqrt(double x);
/*
 * These are like the usual acos(), asin() and sqrt(),
 * except that they round almost-legal values to legal ones.
 * E.g. safe_acos(1.00000001) = acos(1.0) = 0.0, not NaN.
 */

extern double arcsinh(double x);
extern double arccosh(double x);
/*
 * The inverse hyperbolic sine and cosine, which the standard ANSI
 * libraries lack.  [Some but not all platforms now include asinh()
 * and acosh(), so I've renamed my own implementations arcsinh()
 * and arccosh() to avoid conflicts.  JRW 2000/02/20]
 */

/*****
/*
/*          triangulations.c          */
/*
/*          */
/*****/

extern void initialize_triangulation(Triangulation *manifold);
/*
 * Initializes the fields of *manifold to correspond to the
 * empty Triangulation.
 */

extern void initialize_tetrahedron(Tetrahedron *tet);
extern void initialize_cusp(Cusp *cusp);
extern void initialize_edge_class(EdgeClass *edge_class);
/*

```

```

* Initialize the fields of Tetrahedra, Cusps and EdgeClasses to the
* most benign values possible.
*/

extern void free_tetrahedron(Tetrahedron *tet);
/*
* Frees a Tetrahedron and all attached data structures, but does NOT
* remove the Tetrahedron from any doubly linked list it may be on.
*/

extern void clear_shape_history(Tetrahedron *tet);
extern void copy_shape_history(ShapeInversion *source, ShapeInversion **dest);
extern void clear_one_shape_history(Tetrahedron *tet, FillingStatus which_history);
/*
* What you'd expect. See triangulation.c for details.
*/

extern FuncResult check_Euler_characteristic_of_boundary(Triangulation *manifold);
/*
* Returns func_OK if the Euler characteristic of the total boundary of
* the manifold is zero. Otherwise returns func_failed.
*/

extern void number_the_tetrahedra(Triangulation *manifold);
/*
* Sets each Tetrahedron's index field equal to its position in
* the linked list. Indices range from 0 to (num_tetrahedra - 1).
*/

extern void number_the_edge_classes(Triangulation *manifold);
/*
* Sets each EdgeClass's index field equal to its position in the
* linked list. Indices range from 0 to ((number of EdgeClasses) - 1).
*/

extern Permutation compose_permutations(Permutation p1, Permutation p0);
/*
* Returns the composition of two permutations. Permutations are
* composed right-to-left: the composition p1 o p0 is what you get
* by first doing p0, then p1.
*/

/*****
/*
/* update_shapes.c
/*
*****/

extern void update_shapes(Triangulation *manifold, Complex *delta);
/*
* 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 update_shapes.c
* for more details). 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.
*/

/*****
/*
/* volume.c
/*
*****/

extern double birectangular_tetrahedron_volume(
    O31Vector a,
    O31Vector b,
    O31Vector c,
    O31Vector d);
/*

```

```
* Computes the volume of a birectangular tetrahedron using Vinberg's
* article. Please see volume.c for a citation to Vinberg's article.
*/
```

```
#endif
```