

```

/*
 * canonize_part_1.c
 *
 * This file provides the function
 *
 *      FuncResult proto_canonize(Triangulation *manifold);
 *
 * which replaces a Triangulation by the canonical triangulation
 * of the same manifold (if the canonical cell decomposition is a
 * triangulation) or by an arbitrary subdivision of the canonical
 * cell decomposition into Tetrahedra (if the canonical cell
 * decomposition contains cells other than tetrahedra). The
 * primary purpose of proto_canonize() is to be called by the
 * function canonize() (in canonize.c), although it can be called
 * for other reasons, too, such as checking whether a manifold
 * is a 2-bridge knot or link complement.
 *
 * proto_canonize() does not preserve the original Triangulation
 * (if you want to keep it, copy it with copy_triangulation() before
 * calling proto_canonize()). The Triangulation must admit a hyperbolic
 * structure. If one is not already present, proto_canonize() will call
 * find_complete_hyperbolic_structure() to compute it.
 *
 * proto_canonize() returns
 *
 *      func_OK      if the hyperbolic structure is of type
 *                   geometric_solution or nongeometric_solution
 *                   (in which case it will have found a subdivision
 *                   of the canonical cell decomposition)
 *
 *      func_failed  if the hyperbolic structure is of type
 *                   flat_solution, degenerate_solution, other_solution,
 *                   or no_solution (in which case it will not have
 *                   attempted the canonization)
 *
 * When proto_canonize() returns func_OK, a subdivision of the
 * canonical cell decomposition will be present, but because of the
 * possible need for random retriangulation (see below) I cannot
 * prove that proto_canonize() will terminate in all cases (but in
 * practice it always does).
 *
 * proto_canonize() uses the algorithm of
 *
 *      J. Weeks, Convex hulls and isometries of cusped hyperbolic
 *      3-manifolds, Topology Appl. 52 (1993) 127-149.
 *
 * The Tilt Theorem (contained in the above paper) is generalized
 * and given a nicer proof in
 *
 *      M. Sakuma and J. Weeks, The generalized tilt formula,
 *      Geometriae Dedicata 55 (1995) 115-123.
 *
 * Although the algorithm of "Convex hulls..." works fine in practice,
 * it has the aesthetic flaw that it does occasionally get stuck on
 * negatively oriented Tetrahedra, in which case it randomly retriangulates
 * the manifold and starts over. I've tried to find a new algorithm
 * which avoids this problem (while retaining the present algorithm's
 * virtues, namely that it's super fast, and provides an iron-clad
 * guarantee that the topology of the manifold does not change), but
 * so far without success. Maybe someday.
 *
 * Technical note: The canonization algorithm uses only the complete
 * hyperbolic structure, but the low-level retriangulation operations
 * (e.g. two_to_three()) will object if the filled structure is degenerate.
 * We take a strong precaution: we overwrite the filled structure with
 * the complete structure and let polish_hyperbolic_structures() recompute
 * the correct filled structure at the end. This really isn't as
 * inefficient as it sounds. Either the filled structure will be fairly
 * close to the complete structure, in which case it will be computed
 * quickly, or it will be far from the complete structure, in which
 * case the need to provide valid shape histories will force
 * polish_hyperbolic_structures() to recompute the filled structure
 * from scratch anyhow.
 */

```

```

* Programming note: The old version of canonize.c shuffled the
* tetrahedra about on various queues. The present code does not
* do this. Instead, every time it makes some progress (cancellation
* or a 2-3 or 3-2 move) it starts the loop over. This causes some
* unnecessary scanning of the EdgeClass list, but I feel the wasted
* time is small, and is more than compensated for by the reduced
* complexity of the code.
*/

#include "kernel.h"
#include "canonize.h"

#define MAX_ATTEMPTS          64
#define MAX_RETRIANGULATIONS  64
#define ANGLE_EPSILON        1e-6

static FuncResult validate_hyperbolic_structure(Triangulation *manifold);
static Boolean attempt_cancellation(Triangulation *manifold);
static Boolean attempt_three_to_two(Triangulation *manifold);
static Boolean concave_edge(EdgeClass *edge);
static Boolean attempt_two_to_three(Triangulation *manifold);
static Boolean concave_face(Tetrahedron *tet, FaceIndex f);
static double sum_of_tilts(Tetrahedron *tet0, FaceIndex f0);
static Boolean would_create_negatively_oriented_tetrahedra(Tetrahedron *tet0,
    FaceIndex f0);
static Boolean validate_canonical_triangulation(Triangulation *manifold);

FuncResult proto_canonize(
    Triangulation *manifold)
{
    /*
    * 95/10/12 JRW
    * I added the needs_polishing flag so that the solution will be
    * polished iff it needs to be. In the past this was no big deal,
    * but now we want the cusp neighborhoods module to be able to
    * maintain a canonical triangulation in real time. If no changes
    * need to be made and all cusps are complete (so we don't have to
    * worry about restoring the filled solution), we want to get out
    * of here as quickly as possible.
    */

    Boolean all_done,
           needs_polishing;
    int num_attempts;

    num_attempts = 0;
    needs_polishing = FALSE;

    do
    {
        /*
        * First make sure that a hyperbolic structure is present, and
        * all Tetrahedra are positively oriented. Overwrite the filled
        * structure with the complete one.
        */

        if (manifold->solution_type[complete] == geometric_solution
            && all_cusps_are_complete(manifold) == TRUE)
        {
            /*
            * This is the express route.
            *
            * No polishing will be required if the triangulation is
            * already canonical, because the hyperbolic structure won't
            * change and there is no need to restore the filled solution.
            */
        }
        else
        {
            /*
            * This is the generic algorithm.
            *
            * (validate_hyperbolic_structure() overwrites the filled

```

```

        * solution with the complete solution.)
    */
    if (validate_hyperbolic_structure(manifold) == func_failed)
    {
        compute_CS_fudge_from_value(manifold);
        return func_failed;
    }
    needs_polishing = TRUE;
}

/*
 * Choose cusp cross sections bounding equal volumes,
 * and use the Tilt Theorem to compute the tilts.
 * (See "Convex hulls..." for details.)
 */

allocate_cross_sections(manifold);
compute_cross_sections(manifold);
compute_tilts(manifold);

/*
 * Keep going through the following loop as long as we
 * continue to keep improving the triangulation.
 * Do not perform any operation (i.e. any two_to_three()
 * move) that would create negatively oriented Tetrahedra.
 */

while (TRUE)
{
    /*
     * Cancel pairs of Tetrahedra sharing an EdgeClass
     * of order two. (Since the Triangulation contains
     * no negatively oriented Tetrahedra, Tetrahedra sharing
     * an EdgeClass of order two will be within epsilon of
     * being flat.)
     */
    if (attempt_cancellation(manifold) == TRUE)
    {
        needs_polishing = TRUE;
        continue;
    }

    /*
     * Perform 3-2 moves wherever necessary.
     */
    if (attempt_three_to_two(manifold) == TRUE)
    {
        needs_polishing = TRUE;
        continue;
    }

    /*
     * Perform 2-3 moves wherever necessary.
     */
    if (attempt_two_to_three(manifold) == TRUE)
    {
        needs_polishing = TRUE;
        continue;
    }

    /*
     * We can't make any more progress.
     * Break out of the loop, and then check whether we've
     * really found a subdivision of the canonical cell
     * decomposition, or whether we've had the misfortune to
     * get stuck on (potential) negatively oriented Tetrahedra.
     */
    break;
}

/*
 * We don't need the VertexCrossSections any more, so
 * we might as well get rid of them before (possibly)
 * randomizing the manifold.

```

```

    */

    free_cross_sections(manifold);

    /*
     * Did we really find a subdivision of the canonical
     * cell decomposition?
     * Or did we just get stuck on (potential) negatively
     * oriented Tetrahedra?
     */

    all_done = validate_canonical_triangulation(manifold);

    /*
     * If we got stuck on (potential) negatively oriented
     * Tetrahedra, randomize the Triangulation and try
     * again.
     */

    if (all_done == FALSE)
        randomize_triangulation(manifold);

    /*
     * The documentation says that if a hyperbolic structure
     * with all positively oriented tetrahedra is present, then
     * proto_canonize() will never fail. And indeed with enough
     * random retriangulations it should always be able to find
     * a subdivision of the canonical cell decomposition. But
     * if a bug shows up somewhere we don't want this loop to run
     * forever, so if num_attempts exceeds MAX_ATTEMPTS we should
     * declare a fatal error and quit.
     */

    if (++num_attempts > MAX_ATTEMPTS)
        uFatalError("proto_canonize", "canonize_part_1");

} while (all_done == FALSE);

/*
 * Clean up.
 */

if (needs_polishing == TRUE)
{
    /*
     * polish_hyperbolic_structures() takes responsibility for
     * the shape_histories.
     */
    tidy_peripheral_curves(manifold);
    polish_hyperbolic_structures(manifold);

    /*
     * If the Chern-Simons invariant is present, update the fudge factor.
     */
    compute_CS_fudge_from_value(manifold);
}

return func_OK;
}

static FuncResult validate_hyperbolic_structure(
    Triangulation *manifold)
{
    int i;

    /*
     * First make sure some sort of solution is in place.
     */
    if (manifold->solution_type[complete] == not_attempted)
        find_complete_hyperbolic_structure(manifold);

    /*
     * If the solution is something other than geometric_solution

```

```

    * or nongeometric_solution, we're out of luck.
    */
    if (manifold->solution_type[complete] != geometric_solution
        && manifold->solution_type[complete] != nongeometric_solution)
        return func_failed;

    /*
     * Overwrite the filled structure with the complete structure
     * to keep the low-level retriangulation routines happy.
     * (See the technical note at the top of this file for a
     * more complete explanation.)
     */
    copy_solution(manifold, complete, filled);

    /*
     * If all Tetrahedra are positively oriented, we're golden.
     */
    if (manifold->solution_type[complete] == geometric_solution)
        return func_OK;

    /*
     * Try to find a geometric_solution by randomizing the Triangulation.
     * If we can't find one within MAX_RETRIANGULATIONS randomizations,
     * give up and return func_failed.
     */

    for (i = 0; i < MAX_RETRIANGULATIONS; i++)
    {
        randomize_triangulation(manifold);
        if (manifold->solution_type[complete] == geometric_solution)
            return func_OK;
    }

    /*
     * Before we go, we'd better restore the filled solution.
     */
    polish_hyperbolic_structures(manifold);

    return func_failed;
}

static Boolean attempt_cancellation(
    Triangulation *manifold)
{
    EdgeClass *edge,
              *where_to_resume;

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

        if (edge->order == 2)

            if (cancel_tetrahedra(edge, &where_to_resume, &manifold->num_tetrahedra) ==
                func_OK)

                return TRUE;

    return FALSE;
}

static Boolean attempt_three_to_two(
    Triangulation *manifold)
{
    EdgeClass *edge,
              *where_to_resume;

    /*
     * Note: It's easy to prove that if the three original Tetrahedra
     * are positively oriented, then the two new Tetrahedra must be
     * positively oriented as well. So we needn't worry about negatively
     * oriented Tetrahedra here.
     */

```

```

    */

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

        if (edge->order == 3)

            if (concave_edge(edge) == TRUE)
            {
                if (three_to_two(edge, &where_to_resume, &manifold->num_tetrahedra) ==
func_OK)

                    return TRUE;

                else
                /*
                 * The only reason three_to_two() can fail is that the
                 * three Tetrahedra surrounding the EdgeClass are not
                 * distinct. But by Corollary 3 of "Convex hulls..."
                 * this cannot happen where the hull is concave.
                 */
                uFatalError("attempt_three_to_two", "canonize_part_1");

            }

    return FALSE;
}

static Boolean concave_edge(
    EdgeClass *edge)
{
    Tetrahedron *tet;
    FaceIndex f;

    /*
     * The hull in Minkowski space will be concave at an EdgeClass
     * of order 3 iff it is concave at each of the ideal 2-simplices
     * incident to the EdgeClass.
     */

    tet = edge->incident_tet;
    f = one_face_at_edge[edge->incident_edge_index];

    /*
     * According to Section 5 of "Convex hulls..." or Proposition 1.2
     * of "Canonical cell decompositions...", a dihedral angle on the
     * hull will be concave iff the sum of the tilts is positive.
     *
     * We want to create a triangulation with as few Tetrahedra as possible,
     * so when the sum of the tilts is zero, we should return TRUE so the
     * three_to_two() move will be performed. Thus we return TRUE iff
     * the sum of the tilts is greater than some small negative epsilon.
     */

    return ( sum_of_tilts(tet, f) > - CONCAVITY_EPSILON );
}

static Boolean attempt_two_to_three(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    FaceIndex f;

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

        for (f = 0; f < 4; f++)

            if (concave_face(tet, f) == TRUE
                && would_create_negatively_oriented_tetrahedra(tet, f) == FALSE)
            {

```

```

        if (two_to_three(tet, f, &manifold->num_tetrahedra) == func_OK)

            return TRUE;

        else

            /*
             * We should never get to this point.
             */
            uFatalError("attempt_two_to_three", "canonize_part_1.c");

    }

    return FALSE;
}

static Boolean concave_face(
    Tetrahedron *tet,
    FaceIndex    f)
{
    /*
     * According to Section 5 of "Convex hulls..." or Proposition 1.2
     * of "Canonical cell decompositions...", a dihedral angle on the
     * hull will be concave iff the sum of the tilts is positive.
     *
     * If the sum of the tilts is zero we want to return FALSE,
     * to avoid an unnecessary two_to_three() move.
     * So we check whether the sum is greater than some small
     * positive epsilon.
     */

    return ( sum_of_tilts(tet, f) > CONCAVITY_EPSILON );
}

static double sum_of_tilts(
    Tetrahedron *tet0,
    FaceIndex    f0)
{
    Tetrahedron *tet1;
    FaceIndex    f1;

    tet1 = tet0->neighbor[f0];
    f1 = EVALUATE(tet0->gluing[f0], f0);

    return ( tet0->tilt[f0] + tet1->tilt[f1] );
}

static Boolean would_create_negatively_oriented_tetrahedra(
    Tetrahedron *tet0,
    FaceIndex    f0)
{
    Permutation gluing;
    Tetrahedron *tet1;
    FaceIndex    f1,
                side0,
                side1;

    gluing = tet0->gluing[f0];
    tet1 = tet0->neighbor[f0];
    f1 = EVALUATE(gluing, f0);

    /*
     * tet0 and tet1 meet at a common 2-simplex. For each edge
     * of that 2-simplex, add the incident dihedral angles of
     * tet0 and tet1. If any such sum is greater than pi, then
     * the two_to_three() move would create a negatively oriented
     * Tetrahedron on that side, and we return TRUE. Otherwise
     * no negatively oriented Tetrahedra will be created, and we
     * return FALSE.
     *
     * Choose ANGLE_EPSILON to allow the creation of Tetrahedra which
     * are just barely negatively oriented, but essentially flat.
    */

```

```
    */

    for (side0 = 0; side0 < 4; side0++)
    {
        if (side0 == f0)
            continue;

        side1 = EVALUATE(gluing, side0);

        if (tet0->shape[complete]->cwl[ultimate][edge3_between_faces[f0][side0]].log.imag
            + tet1->shape[complete]->cwl[ultimate][edge3_between_faces[f1][side1]].log.imag
            > PI + ANGLE_EPSILON)

            return TRUE;
    }

    return FALSE;
}

static Boolean validate_canonical_triangulation(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    FaceIndex f;

    /*
     * Check whether the sum of the tilts is nonnegative at each 2-simplex.
     */

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

        for (f = 0; f < 4; f++)

            if (concave_face(tet, f) == TRUE)

                return FALSE;

    return TRUE;
}
```