

```

/*
 * symmetry_group_closed.c
 *
 * This file provides the function
 *
 * FuncResult compute_closed_symmetry_group(
 *                               Triangulation *manifold,
 *                               SymmetryGroup **symmetry_group,
 *                               Triangulation **symmetric_triangulation,
 *                               Boolean *is_full_group);
 *
 * See symmetry_group.c for an explanation of the arguments
 * and return values.
 *
 * The theory behind this algorithm is explained in the preprint
 *
 * C. Hodgson & J. Weeks, "Symmetries, isometries and length
 * spectra of closed hyperbolic 3-manifolds", to appear in
 * Experimental Mathematics.
 *
 * Please note that the current version of this preprint is
 * substantially different than the version which appeared
 * in the Geometry Center's preprint series (although I plan
 * to install the new version in the on-line preprint library).
 */

#include "kernel.h"
#include <limits.h>

#define LENGTH_EPSILON          1e-8
#define TORSION_EPSILON         1e-8
#define ZERO_TORSION_EPSILON    1e-8
#define PI_TORSION_EPSILON      1e-8
#define CRUDE_EPSILON           1e-3
#define VOLUME_ERROR_EPSILON    1e-8
#define INFINITE_ORDER          INT_MAX
#define INFINITE_MULTIPPLICITY  INT_MAX
#define MAX_DUAL_CURVE_LENGTH   8
#define MAX_RANDOMIZATIONS      16 /* for geometric complete structure */
#define MAX_RETRIANGULATIONS     8  /* for geometric filled structure */

typedef struct
{
    double length,
           torsion; /* absolute value of torsion */
    int pos_multiplicity,
        neg_multiplicity,
        zero_multiplicity, /* torsion within epsilon of zero */
        total_multiplicity; /* sum of previous three fields */
    /* torsions within epsilon of -pi should not occur */
} MergedMultiLength;

static WEPolyhedron *compute_polyhedron(Triangulation *manifold);
static FuncResult compute_symmetry_group_using_polyhedron(Triangulation *manifold,
    SymmetryGroup **symmetry_group, Triangulation **symmetric_triangulation, Boolean *
    is_full_group, WEPolyhedron *polyhedron);
static void compute_length_spectrum(WEPolyhedron *polyhedron, MultiLength **
    spectrum, int *num_lengths);
static double rigor_radius(double spine_radius, double cutoff_length);
static FuncResult merge_length_spectrum(int num_lengths, MultiLength *spectrum, int *
    num_merged_lengths, MergedMultiLength **merged_spectrum);
static void try_to_drill_curves(Triangulation *original_manifold, MergedMultiLength
    desired_curves, int *lower_bound, int *upper_bound, SymmetryGroup **symmetry_group,
    Triangulation **symmetric_triangulation);
static FuncResult drill_one_curve(Triangulation **manifold, MergedMultiLength *
    remaining_curves);
static FuncResult fill_first_cusp(Triangulation **manifold);
static FuncResult find_geometric_solution(Triangulation **manifold);
static FuncResult compute_symmetry_group_without_polyhedron(Triangulation *manifold,
    SymmetryGroup **symmetry_group, Triangulation **symmetric_triangulation, Boolean *
    is_full_group);
static void try_to_drill_unknown_curves(Triangulation **manifold, Complex
    desired_length, int *lower_bound, SymmetryGroup **symmetry_group, Triangulation **
    symmetric_triangulation);

```

```

FuncResult compute_closed_symmetry_group(
    Triangulation    *manifold,
    SymmetryGroup    **symmetry_group,
    Triangulation    **symmetric_triangulation,
    Boolean          *is_full_group)
{
    FuncResult    result;

    /*
     * Make sure the variables used to pass back our results
     * are all initially empty.
     */
    if (*symmetry_group != NULL
        || *symmetric_triangulation != NULL)
        uFatalError("compute_closed_symmetry_group", "symmetry_group");

    /*
     * compute_symmetry_group() should have passed us a 1-cusp
     * manifold with a Dehn filling on its cusp.
     */
    if (get_num_cusps(manifold) != 1
        || all_cusps_are_filled(manifold) == FALSE
        || all_Dehn_coefficients_are_relatively_prime_integers(manifold) == FALSE)
    {
        uFatalError("compute_closed_symmetry_group", "symmetry_group_closed");
    }

    /*
     * For later convenience, change the basis on the cusp
     * so that the Dehn filling curve becomes a meridian.
     */
    {
        MatrixInt22    basis_change[1];

        current_curve_basis(manifold, 0, basis_change[0]);
        change_peripheral_curves(manifold, basis_change);
    }

    /*
     * At the very least, we can try to establish a (possibly trivial)
     * lower bound on the symmetry group by computing the group
     * which preserves the given core geodesic.
     */
    {
        SymmetryGroup    *dummy = NULL;

        if (compute_cusped_symmetry_group(manifold, &dummy, symmetry_group) == func_OK)
        {
            copy_triangulation(manifold, symmetric_triangulation);
            free_symmetry_group(dummy);    /* we don't need dummy */
        }
        else
        {
            /*
             * The only way compute_cusped_symmetry_group() may fail
             * is if a canonical cell decomposition cannot be found,
             * e.g. because the manifold is not hyperbolic.
             */
            return func_failed;
        }
    }

    /*
     * For small to medium sized manifolds we should have
     * no trouble getting a Dirichlet domain. But if we can't,
     * then we want to muddle along as best we can without one.
     */
    {
        WEPolyhedron    *polyhedron;

        polyhedron = compute_polyhedron(manifold);
    }
}

```

```

    if (polyhedron != NULL)
    {
        result = compute_symmetry_group_using_polyhedron(
                                manifold,
                                symmetry_group,
                                symmetric_triangulation,
                                is_full_group,
                                polyhedron);

        free_Dirichlet_domain(polyhedron);
    }
    else
        result = compute_symmetry_group_without_polyhedron(
                                manifold,
                                symmetry_group,
                                symmetric_triangulation,
                                is_full_group);
}

return result;
}

static WEPolyhedron *compute_polyhedron(
    Triangulation *manifold)
{
    int i;
    WEPolyhedron *polyhedron;

    const static int num_precisions = 5;
    const static double precision[5] = {1e-8, 1e-6, 1e-10, 1e-4, 1e-12};

    for (i = 0; i < num_precisions; i++)
    {
        polyhedron = Dirichlet( manifold,
                                precision[i],
                                TRUE,
                                Dirichlet_stop_here,
                                TRUE);

        if (polyhedron != NULL)
            return polyhedron;
    }

    /*
     * Even after trying five precisions we still couldn't
     * get a Dirichlet domain.
     */
    return NULL;
}

static FuncResult compute_symmetry_group_using_polyhedron(
    Triangulation *manifold,
    SymmetryGroup **symmetry_group,
    Triangulation **symmetric_triangulation,
    Boolean *is_full_group,
    WEPolyhedron *polyhedron)
{
    MultiLength *spectrum;
    int num_lengths;
    MergedMultiLength *merged_spectrum;
    int i, num_merged_lengths, lower_bound, upper_bound;

    /*
     * Use the polyhedron to compute a length spectrum,
     * if a nontrivial length spectrum can be computed in
     * a reasonable amount of time.
     */
    compute_length_spectrum(polyhedron, &spectrum, &num_lengths);

    /*
     * If we couldn't get a nonempty length spectrum in

```

```

    * a reasonable amount of time, so we must fall back to
    * compute_symmetry_group_without_polyhedron().
    */
if (num_lengths == 0)
    return(compute_symmetry_group_without_polyhedron(
        manifold, symmetry_group, symmetric_triangulation, is_full_group));

/*
 * Merge complex lengths with opposite torsions, because
 * typically they will need to be drilled together.
 * Usually merge_length_spectrum() will succeed, but it will fail
 * if the numerical accuracy of the length spectrum is too poor
 * to clearly resolve equal lengths. In the latter case we must
 * fall back to compute_symmetry_group_without_polyhedron().
 */
if (merge_length_spectrum(
    num_lengths, spectrum, &num_merged_lengths, &merged_spectrum)
    == func_failed)
{
    free_length_spectrum(spectrum);
    return(compute_symmetry_group_without_polyhedron(
        manifold, symmetry_group, symmetric_triangulation, is_full_group));
}

/*
 * We no longer need the original, unmerged spectrum.
 */
free_length_spectrum(spectrum);
spectrum = NULL;
num_lengths = 0;

/*
 * We maintain lower and upper bounds on the order of the symmetry
 * group. Please see the preprint "Symmetries, isometries and length
 * spectra of closed hyperbolic 3-manifolds" for details (cf. the top
 * of this file). Most likely, compute_closed_symmetry_group() has
 * already provided us with a crude lower bound on the true group.
 */
if (*symmetry_group != NULL)
{
    lower_bound = symmetry_group_order(*symmetry_group);
    upper_bound = INFINITE_ORDER;
}
else
{
    lower_bound = 0;
    upper_bound = INFINITE_ORDER;
}

/*
 * Try to drill each MergedMultiLength in turn, to obtain various
 * Dehn filling descriptions of the manifold. A given description
 * may improve the lower bound and/or the upper bound.
 */
for (i = 0; i < num_merged_lengths; i++)
{
    try_to_drill_curves(
        manifold,
        merged_spectrum[i],
        &lower_bound,
        &upper_bound,
        symmetry_group,
        symmetric_triangulation);

    if (lower_bound == upper_bound)
        break;
}

/*
 * Free merged_spectrum.
 */
my_free(merged_spectrum);

/*
 * We know the symmetry group with complete certainty iff

```

```

    * lower_bound == upper_bound. Otherwise we have only a
    * lower bound on the true symmetry group.
    */
    *is_full_group = (lower_bound == upper_bound);

    /*
    * If we've found any symmetry group at all, return func_OK.
    */
    if (lower_bound > 0)
        return func_OK;
    else
        return func_failed;
}

static void compute_length_spectrum(
    WEPolyhedron *polyhedron,
    MultiLength **spectrum,
    int *num_lengths)
{
    /*
    * We have two concerns:
    *
    * (1) We want to get a nontrivial length spectrum
    *     (preferably one containing several different lengths,
    *     but certainly not an empty one).
    *
    * (2) We don't want the computation to take too long.
    *
    * We address the second concern by refusing to tile
    * beyond a max_tiling_radius of 4.0 (this could be increased,
    * perhaps to 5.0, on faster machines).
    *
    * Our plan, then, is to try successively larger values
    * for cutoff_length until either we get enough_lengths, or we
    * exceed the max_tiling_radius, whichever comes first.
    * At that point we declare success if the length spectrum
    * is nonempty (even if it contains fewer than enough_lengths),
    * or failure otherwise.
    */

    double cutoff_length;

    const static double max_tiling_radius = 5.0; /* changed from 4.0 to 5.0 JRW 98/4/30 */
    /
    const static int enough_lengths = 3;

    /*
    * Initially we have no length spectrum.
    */
    *spectrum = NULL;
    *num_lengths = 0;

    /*
    * Start with cutoff_length = 1.0, and keep incrementing it
    * until either (1) we get enough_lengths, or (2) the tiling_radius
    * becomes unacceptably large.
    */
    for
    (
        cutoff_length = 1.0;
        *num_lengths < enough_lengths
        && rigor_radius(polyhedron->spine_radius, cutoff_length) < max_tiling_radius;
        cutoff_length += 1.0
    )
    {
        /*
        * If a spectrum is left over from the previous iteration
        * of the for(;;) loop, free it.
        */
        if (*spectrum != NULL)
        {
            free_length_spectrum(*spectrum);
            *spectrum = NULL;
        }
    }
}

```

```

        *num_lengths      = 0;
    }

    length_spectrum(polyhedron, cutoff_length, TRUE, TRUE, 0.0, spectrum, num_lengths);
}

/*
 * We've done our best, so return whether or not *num_lengths > 0.
 */
}

static double rigor_radius(
    double spine_radius,
    double cutoff_length)
{
    return 2*arccosh(cosh(spine_radius)*cosh(cutoff_length/2));
}

static FuncResult merge_length_spectrum(
    int num_lengths,
    MultiLength *spectrum,
    int *num_merged_lengths,
    MergedMultiLength **merged_spectrum)
{
    int i,
        j;
    Boolean already_on_list;

    /*
     * compute_symmetry_group_using_polyhedron() has already checked
     * that num_lengths is nonzero.
     */
    if (num_lengths <= 0)
        uFatalError("merge_length_spectrum", "symmetry_group_closed");

    /*
     * The merged_spectrum will require at most as many entries
     * as the original spectrum, so we allocate an array of that
     * length. This could waste space by up to a factor of two,
     * but this is no big deal.
     */
    *merged_spectrum = NEW_ARRAY(num_lengths, MergedMultiLength);

    *num_merged_lengths = 0;

    /*
     * Look at each MultiLength in turn, and decide how it should
     * be incorporated into the merged_spectrum.
     */
    for (i = 0; i < num_lengths; i++)
    {
        /*
         * Handle torsion zero as a special case.
         */
        if (fabs(spectrum[i].length.imag) < ZERO_TORSION_EPSILON)
        {
            (*merged_spectrum)[*num_merged_lengths].length = spectrum[i].length.real;
            (*merged_spectrum)[*num_merged_lengths].torsion = 0.0;
            (*merged_spectrum)[*num_merged_lengths].pos_multiplicity = 0;
            (*merged_spectrum)[*num_merged_lengths].neg_multiplicity = 0;
            (*merged_spectrum)[*num_merged_lengths].zero_multiplicity = spectrum[i].multiplicity;
            (*merged_spectrum)[*num_merged_lengths].total_multiplicity = spectrum[i].multiplicity;
            (*num_merged_lengths)++;
            continue;
        }

        /*
         * The complex length program should never report a torsion
         * of -pi. (It always converts to +pi.)
         */
    }
}

```

```

    */
    if (spectrum[i].length.imag < -PI + PI_TORSION_EPSILON)
        uFatalError("merge_length_spectrum", "symmetry_group_closed");

    /*
     * Treat torsion pi as a special case.
     */
    if (spectrum[i].length.imag > PI - PI_TORSION_EPSILON)
    {
        (*merged_spectrum)[*num_merged_lengths].length      = spectrum[i].length.real;
        (*merged_spectrum)[*num_merged_lengths].torsion      = PI;
        (*merged_spectrum)[*num_merged_lengths].pos_multiplicity = spectrum[i].multiplicity;
        (*merged_spectrum)[*num_merged_lengths].neg_multiplicity = 0;
        (*merged_spectrum)[*num_merged_lengths].zero_multiplicity = 0;
        (*merged_spectrum)[*num_merged_lengths].total_multiplicity = spectrum[i].multiplicity;
        (*num_merged_lengths)++;
        continue;
    }

    /*
     * Is the given (length, abs(torsion)) already on the merged list?
     * If so, fold in the new values.
     */
    already_on_list = FALSE;
    for (j = 0; j < *num_merged_lengths; j++)
    {
        if (fabs(spectrum[i].length.real - (*merged_spectrum)[j].length) <
            LENGTH_EPSILON
            && fabs(spectrum[i].length.imag - (*merged_spectrum)[j].torsion) <
            TORSION_EPSILON)
        {
            if (spectrum[i].length.imag > 0.0)
                (*merged_spectrum)[j].pos_multiplicity += spectrum[i].multiplicity;
            else
                (*merged_spectrum)[j].neg_multiplicity += spectrum[i].multiplicity;
            (*merged_spectrum)[j].total_multiplicity += spectrum[i].multiplicity;
            already_on_list = TRUE;
            break;
        }
    }
    if (already_on_list == TRUE)
        continue;

    /*
     * As a guard against errors, check that the current complex length
     * is not within some very crude epsilon of an existing length.
     */
    for (j = 0; j < *num_merged_lengths; j++)
    {
        if (fabs(spectrum[i].length.real - (*merged_spectrum)[j].length) <
            CRUDE_EPSILON
            && fabs(spectrum[i].length.imag - (*merged_spectrum)[j].torsion) <
            CRUDE_EPSILON)
        {
            my_free(*merged_spectrum);
            *merged_spectrum = NULL;
            *num_merged_lengths = 0;
            return func_failed;
        }
    }

    /*
     * Create a new entry on the merged list.
     */
    (*merged_spectrum)[*num_merged_lengths].length = spectrum[i].length.real;
    (*merged_spectrum)[*num_merged_lengths].torsion = fabs(spectrum[i].length.imag);
    if (spectrum[i].length.imag > 0.0)
    {
        (*merged_spectrum)[*num_merged_lengths].pos_multiplicity = spectrum[i].multiplicity;
        (*merged_spectrum)[*num_merged_lengths].neg_multiplicity = 0;
    }

```

```

    }
    else
    {
        (*merged_spectrum)[*num_merged_lengths].pos_multiplicity = 0;
        (*merged_spectrum)[*num_merged_lengths].neg_multiplicity = spectrum[i].
multiplicity;
    }
    (*merged_spectrum)[*num_merged_lengths].zero_multiplicity = 0;
    (*merged_spectrum)[*num_merged_lengths].total_multiplicity = spectrum[i].
multiplicity;
    (*num_merged_lengths)++;
}

if (*num_merged_lengths > num_lengths)
    uFatalError("merge_length_spectrum", "symmetry_group_closed");

return func_OK;
}

static void try_to_drill_curves(
    Triangulation      *original_manifold,
    MergedMultiLength  desired_curves,
    int                *lower_bound,
    int                *upper_bound,
    SymmetryGroup      **symmetry_group,
    Triangulation      **symmetric_triangulation)
{
    Triangulation      *manifold;
    double             old_volume,
                      new_volume;
    int                singularity_index;
    Complex             core_length;
    SymmetryGroup      *manifold_sym_grp = NULL,
                      *link_sym_grp = NULL;
    int                new_upper_bound;
    MergedMultiLength  remaining_curves;
    int                num_possible_images;

    /*
     * Let's work on a copy, and leave the original_manifold untouched.
     */
    copy_triangulation(original_manifold, &manifold);

    /*
     * As a guard against creating weird triangulations,
     * do an "unnecessary" error check.
     */
    old_volume = volume(manifold, NULL);

    /*
     * To assist in the bookkeeping, we make a copy of the MergedMultiLength
     * and use it to keep track of how many curves remain.
     */
    remaining_curves = desired_curves;

    /*
     * In cases where the number of geodesics with positive torsion
     * does not equal the number with negative torsion (and both numbers
     * are nonzero), the manifold is clearly chiral, and we want to
     * drill only the curves with the lesser multiplicity.
     */
    if (remaining_curves.pos_multiplicity > 0
        && remaining_curves.neg_multiplicity > 0
        && remaining_curves.pos_multiplicity != remaining_curves.neg_multiplicity)
    {
        /*
         * Suppress the curves with the greater multiplicity.
         */
        if (remaining_curves.pos_multiplicity > remaining_curves.neg_multiplicity)
        {
            remaining_curves.total_multiplicity -= remaining_curves.pos_multiplicity;
            remaining_curves.pos_multiplicity = 0;
        }
    }
}

```



```

    else
    {
        remaining_curves.total_multiplicity -= remaining_curves.neg_multiplicity;
        remaining_curves.neg_multiplicity = 0;
    }
}

/*
 * Note the original number of curves in the set we'll be drilling.
 * It's the number of possible images of a given curve under the
 * action of the symmetry group. (In the case of differing positive
 * and negative multiplicity, the curves of lesser multiplicity
 * must be taken to themselves. Otherwise curves of positive
 * torsion could be taken to curves of negative torsion.)
 */
num_possible_images = remaining_curves.total_multiplicity;

/*
 * The current core geodesic may or may not have the desired length.
 * If it doesn't, replace it with a new one which does.
 * Note that the absolute value of the torsion might be correct,
 * but the sign of the torsion might have been "suppressed" above.
 */

core_geodesic(manifold, 0, &singularity_index, &core_length, NULL);

if
(
    /*
     * length is wrong
     */
    fabs(desired_curves.length - core_length.real) > LENGTH_EPSILON
||
    /*
     * torsion is wrong
     */
    (
        /*
         * Doesn't match desired positive torsion.
         */
        (
            /*
             * We're not looking for positive torsion.
             */
            remaining_curves.pos_multiplicity == 0
        ||
            /*
             * It doesn't have the correct positive torsion.
             */
            fabs(desired_curves.torsion - core_length.imag) > TORSION_EPSILON
        )
    &&
        /*
         * Doesn't match desired negative torsion.
         */
        (
            /*
             * We're not looking for negative torsion.
             */
            remaining_curves.neg_multiplicity == 0
        ||
            /*
             * It doesn't have the correct negative torsion.
             */
            fabs((-desired_curves.torsion) - core_length.imag) > TORSION_EPSILON
        )
    &&
        /*
         * Doesn't match desired zero torsion.
         */
        (
            /*
             * We're not looking for zero torsion.
             */

```

```

        remaining_curves.zero_multiplicity == 0
    ||
    /*
     * It doesn't have zero torsion.
     */
    fabs(core_length.imag) > ZERO_TORSION_EPSILON
)
)
{
    if (drill_one_curve(&manifold, &remaining_curves) == func_failed
    || fill_first_cusp(&manifold) == func_failed)
    {
        free_triangulation(manifold);
        return;
    }
}
else /* core geodesic is already a desired curve */
{
    /*
     * The complex length program should never report a torsion
     * of -pi. (It always converts to +pi.)
     */
    if (core_length.imag < -PI + PI_TORSION_EPSILON)
        uFatalError("try_to_drill_curves", "symmetry_group_closed");

    if (core_length.imag > ZERO_TORSION_EPSILON)
        remaining_curves.pos_multiplicity--;
    else if (core_length.imag < -ZERO_TORSION_EPSILON)
        remaining_curves.neg_multiplicity--;
    else
        remaining_curves.zero_multiplicity--;

    remaining_curves.total_multiplicity--;
}

/*
 * At this point we have drilled out one curve of the correct length.
 * If the description is positively oriented, we may obtain an upper bound
 * on the order of the symmetry group.
 */
if (find_geometric_solution(&manifold) == func_OK)
{
    if (compute_cusped_symmetry_group(manifold, &manifold_sym_grp, &link_sym_grp) !=
func_OK)
    {
        free_triangulation(manifold);
        return;
    }

    new_upper_bound = num_possible_images * symmetry_group_order(link_sym_grp);
    if (new_upper_bound < *upper_bound)
        *upper_bound = new_upper_bound;

    free_symmetry_group(manifold_sym_grp);
    free_symmetry_group(link_sym_grp);
    manifold_sym_grp = NULL;
    link_sym_grp = NULL;
}

/*
 * We have drilled out one curve of the correct length.
 * Keep drilling curves until we've got them all.
 */
while (remaining_curves.total_multiplicity > 0)
{
    if (drill_one_curve(&manifold, &remaining_curves) == func_failed)
    {
        free_triangulation(manifold);
        return;
    }
}

/*
 * We could have a degenerate solution if we drilled out

```

```

    * curves in the wrong isotopy classes.
    */
    if (get_filled_solution_type(manifold) == degenerate_solution)
    {
        free_triangulation(manifold);
        return;
    }

    /*
    * Try to get a geometric_solution.
    */
    (void) find_geometric_solution(&manifold);

    /*
    * Finish our "unnecessary" error check.
    */
    new_volume = volume(manifold, NULL);
    if (fabs(new_volume - old_volume) > VOLUME_ERROR_EPSILON)
    {
        /*
        * If ever we're looking for pathological triangulations,
        * this would be a good place to set a breakpoint.
        */
        free_triangulation(manifold);
        return;
    }

    /*
    * Check the symmetry group of what we have, and see whether it provides
    * a lower bound. If a geometric_solution was found, then we'll have
    * an upper bound as well, i.e. we'll know the symmetry group exactly.
    */
    if (compute_cusped_symmetry_group(manifold, &manifold_sym_grp, &link_sym_grp) == func_failed)
    {
        free_triangulation(manifold);
        return;
    }

    if (symmetry_group_order(link_sym_grp) > *lower_bound)
    {
        *lower_bound = symmetry_group_order(link_sym_grp);

        free_symmetry_group(*symmetry_group);          /* NULL is OK */
        *symmetry_group = link_sym_grp;

        free_triangulation(*symmetric_triangulation);  /* NULL is OK */
        copy_triangulation(manifold, symmetric_triangulation);
    }

    if (get_filled_solution_type(manifold) == geometric_solution)
    {
        /*
        * Include an error check.
        */
        new_upper_bound = symmetry_group_order(link_sym_grp);
        if (new_upper_bound < *upper_bound)
            *upper_bound = new_upper_bound;
        if (*upper_bound != *lower_bound)
            uFatalError("try_to_drill_curves", "symmetry_group_closed");
    }

    free_symmetry_group(manifold_sym_grp);
    if (link_sym_grp != *symmetry_group)
        free_symmetry_group(link_sym_grp);

    free_triangulation(manifold);
}

static FuncResult drill_one_curve(
    Triangulation      **manifold,
    MergedMultiLength  *remaining_curves)
{

```

```

int i;
int num_curves;
DualOneSkeletonCurve **the_curves;
int desired_index;
Complex filled_length;
Triangulation *new_manifold;
int count;

/*
 * See what curves are drillable.
 */
dual_curves(*manifold, MAX_DUAL_CURVE_LENGTH, &num_curves, &the_curves);
if (num_curves == 0)
    return func_failed;

desired_index = -1;
for (i = 0; i < num_curves; i++)
{
    get_dual_curve_info(the_curves[i], NULL, &filled_length, NULL);

    if (
        fabs(remaining_curves->length - filled_length.real) < LENGTH_EPSILON
        && fabs(remaining_curves->torsion - fabs(filled_length.imag)) < TORSION_EPSILON
        && (
            (
                remaining_curves->pos_multiplicity > 0
                && filled_length.imag > ZERO_TORSION_EPSILON
            )
            ||
            (
                remaining_curves->neg_multiplicity > 0
                && filled_length.imag < -ZERO_TORSION_EPSILON
            )
            ||
            (
                remaining_curves->zero_multiplicity > 0
                && fabs(filled_length.imag) < ZERO_TORSION_EPSILON
            )
        )
    )
    {
        desired_index = i;
        break;
    }
}
if (desired_index == -1)
{
    free_dual_curves(num_curves, the_curves);
    return func_failed;
}

new_manifold = drill_cusp(*manifold, the_curves[desired_index], get_triangulation_name
(*manifold));

if (new_manifold == NULL)
{
    free_dual_curves(num_curves, the_curves);
    return func_failed;
}

/*
 * Usually the complete solution will be geometric, even if
 * the filled solution is not. But occasionally we'll get
 * a new_manifold which didn't simplify sufficiently, and
 * we'll need to rattle it around to get a decent triangulation.
 */
count = MAX_RANDOMIZATIONS;
while (--count >= 0
    && get_complete_solution_type(new_manifold) != geometric_solution)
    randomize_triangulation(new_manifold);

/*
 * Set the new Dehn filling coefficient to (1, 0)
 * to recover the closed manifold.

```

```

    */

    set_cusp_info(new_manifold, get_num_cusps(new_manifold) - 1, FALSE, 1.0, 0.0);
    do_Dehn_filling(new_manifold);

    free_dual_curves(num_curves, the_curves);

    free_triangulation(*manifold);
    *manifold = new_manifold;
    new_manifold = NULL;

    if (filled_length.imag > ZERO_TORSION_EPSILON)
        remaining_curves->pos_multiplicity--;
    else if (filled_length.imag < -ZERO_TORSION_EPSILON)
        remaining_curves->neg_multiplicity--;
    else
        remaining_curves->zero_multiplicity--;

    remaining_curves->total_multiplicity--;

    return func_OK;
}

static FuncResult fill_first_cusp(
    Triangulation    **manifold)
{
    Triangulation    *new_manifold;
    int               count;
    Boolean           fill_cusp[2] = {TRUE, FALSE};

    if (get_num_cusps(*manifold) != 2)
        uFatalError("fill_first_cusp", "symmetry_group_closed");

    new_manifold = fill_cusps(*manifold, fill_cusp, get_triangulation_name(*manifold),
        FALSE);
    if (new_manifold == NULL)
        return func_failed; /* this seems unlikely */

    /*
     * Usually the complete solution will be geometric, even if
     * the filled solution is not. But occasionally we'll get
     * a new_manifold which didn't simplify sufficiently, and
     * we'll need to rattle it around to get a decent triangulation.
     */
    count = MAX_RANDOMIZATIONS;
    while (--count >= 0
        && get_complete_solution_type(new_manifold) != geometric_solution)
        randomize_triangulation(new_manifold);

    free_triangulation(*manifold);
    *manifold = new_manifold;
    new_manifold = NULL;

    return func_OK;
}

static FuncResult find_geometric_solution(
    Triangulation    **manifold)
{
    int               i;
    Triangulation    *copy;

    /*
     * If it ain't broke, don't fix it.
     */
    if (get_filled_solution_type(*manifold) == geometric_solution)
        return func_OK;

    /*
     * Save a copy in case we make the triangulation even worse,
     * e.g. by converting a nongeometric_solution to a degenerate_solution.
     */

```

```

    copy_triangulation(*manifold, &copy);

    /*
     * Attempt to find a geometric_solution.
     */
    for (i = 0; i < MAX_RETRIANGULATIONS; i++)
    {
        randomize_triangulation(*manifold);
        if (get_filled_solution_type(*manifold) == geometric_solution)
        {
            free_triangulation(copy);
            return func_OK;
        }

        /*
         * Every so often try canonizing as a further
         * stimulus to randomization.
         */
        if ((i%4) == 3)
        {
            proto_canonize(*manifold);
            if (get_filled_solution_type(*manifold) == geometric_solution)
            {
                free_triangulation(copy);
                return func_OK;
            }
        }
    }

    /*
     * What have we got left?
     */
    switch (get_filled_solution_type(*manifold))
    {
        case geometric_solution:
            free_triangulation(copy);
            return func_OK;

        case nongeometric_solution:
            free_triangulation(copy);
            return func_failed;

        default:
            /*
             * If we don't have at least a nongeometric_solution,
             * restore the original triangulation.
             */
            free_triangulation(*manifold);
            *manifold = copy;
            return func_failed;
    }
}

static FuncResult compute_symmetry_group_without_polyhedron(
    Triangulation    *manifold,
    SymmetryGroup    **symmetry_group,
    Triangulation    **symmetric_triangulation,
    Boolean          *is_full_group)
{
    int                lower_bound,
                      num_curves,
                      i;

    DualOneSkeletonCurve **the_curves;
    Complex              prev_length,
                      filled_length;
    Triangulation        *copy;

    /*
     * Without a polyhedron we can't get a length spectrum,
     * and without a length spectrum we can't know for sure
     * when we've found the whole symmetry group.
     */
    *is_full_group = FALSE;

```

```

/*
 * compute_closed_symmetry_group() will have already computed
 * the symmetry subgroup preserving the current core geodesic.
 * (compute_closed_symmetry_group() also checks that we have
 * precisely one cusp.)
 */

if (*symmetry_group == NULL)
    uFatalError("compute_symmetry_group_without_polyhedron", "symmetry_group_closed");

/*
 * Initialize a (possible trivial) lower_bound on the order
 * of the full symmetry group.
 */

lower_bound = symmetry_group_order(*symmetry_group);

/*
 * What curves are available in the 1-skeleton?
 */

dual_curves(manifold, MAX_DUAL_CURVE_LENGTH, &num_curves, &the_curves);

prev_length = Zero;

for (i = 0; i < num_curves; i++)
{
    /*
     * Get the filled_length of the_curves[i].
     */
    get_dual_curve_info(the_curves[i], NULL, &filled_length, NULL);

    /*
     * Work with the absolute value of the torsion.
     */
    filled_length.imag = fabs(filled_length.imag);

    /*
     * Skip lengths which appear equal a previous length.
     * (Note: dual_curves() first sorts by filled_length,
     * then complete_length.)
     */
    if (fabs(filled_length.real - prev_length.real) < LENGTH_EPSILON
        && fabs(filled_length.imag - prev_length.imag) < TORSION_EPSILON)
        continue;
    else
        prev_length = filled_length;

    /*
     * Let's work on a copy, and leave the original_manifold untouched.
     */
    copy_triangulation(manifold, &copy);

    /*
     * Proceed as in try_to_drill_curves(), but with the realization
     * that we don't know how many curves we have of each given length.
     */
    try_to_drill_unknown_curves(
        &copy,
        filled_length,
        &lower_bound,
        symmetry_group,
        symmetric_triangulation);

    /*
     * Free the copy.
     */
    free_triangulation(copy);
}

free_dual_curves(num_curves, the_curves);

return func_OK;
}

```

```

static void try_to_drill_unknown_curves(
    Triangulation      **manifold,
    Complex             desired_length,
    int                 *lower_bound,
    SymmetryGroup       **symmetry_group,
    Triangulation       **symmetric_triangulation)
{
    double              old_volume;
    MergedMultiLength   the_desired_curves;
    int                 singularity_index;
    Complex              core_length;
    SymmetryGroup        *manifold_sym_grp = NULL,
    *link_sym_grp = NULL;

    /*
     * We want to drill as many curves as possible whose length
     * is desired_length.real and whose torsion has absolute value
     * desired_length.imag. We don't know how many curves of this
     * complex length to expect, so we won't know when to stop.
     * (We stop when we can't find another such curve to drill, or
     * a drilling fails.) We compute the symmetry group at each
     * step, since drilling additional curves may sometimes decrease
     * the size of the symmetry subgroup.
     */

    /*
     * As a guard against creating weird triangulations,
     * do an "unnecessary" error check.
     */
    old_volume = volume(*manifold, NULL);

    /*
     * Mock up a MergedMultiLength to represent the desired_length
     * we want to drill. (desired_length.imag will always be nonnegative.)
     */

    the_desired_curves.length           = desired_length.real;
    the_desired_curves.torsion           = desired_length.imag;
    the_desired_curves.pos_multiplicity = INFINITE_MULTIPPLICITY;
    the_desired_curves.neg_multiplicity  = INFINITE_MULTIPPLICITY;
    the_desired_curves.zero_multiplicity = INFINITE_MULTIPPLICITY;
    the_desired_curves.total_multiplicity = INFINITE_MULTIPPLICITY;

    /*
     * The current core geodesic may or may not have the desired length.
     * If it doesn't, replace it with a new one which does.
     * Note that the absolute value of the torsion might be correct,
     * but the sign of the torsion might have been "suppressed" above.
     */

    core_geodesic(*manifold, 0, &singularity_index, &core_length, NULL);

    if (fabs(desired_length.real - core_length.real) > LENGTH_EPSILON
        || fabs(desired_length.imag - fabs(core_length.imag)) > TORSION_EPSILON)
    {
        /*
         * Try to drill a curve of the desired_length.
         */
        if (drill_one_curve(manifold, &the_desired_curves) == func_failed
            || fill_first_cusp(manifold) == func_failed)
            return;
    }

    /*
     * At this point we've got one curve of the desired_length drilled.
     * Compute its symmetry subgroup, try to drill another curve, . . .
     * until we can no longer drill.
     */
    do
    {
        /*

```



```

    * Don't mess with anything worse than a nongeometric_solution.
    */
    if (get_filled_solution_type(*manifold) != geometric_solution
        && get_filled_solution_type(*manifold) != nongeometric_solution)
        break;

    /*
    * Make sure the volume is plausible, as a further guard against
    * weird solutions. (The symmetry subgroup would still be valid,
    * but we certainly don't want to proceed further.)
    */
    if (fabs(volume(*manifold, NULL) - old_volume) > VOLUME_ERROR_EPSILON)
        break;

    /*
    * Compute the symmetry subgroup.
    */
    if (compute_cusped_symmetry_group(*manifold, &manifold_sym_grp, &link_sym_grp) ==
func_failed)
        break;

    /*
    * Have we improved the lower bound?
    */
    if (symmetry_group_order(link_sym_grp) > *lower_bound)
    {
        *lower_bound = symmetry_group_order(link_sym_grp);

        free_symmetry_group(*symmetry_group);          /* NULL is OK */
        *symmetry_group = link_sym_grp;

        free_triangulation(*symmetric_triangulation);  /* NULL is OK */
        copy_triangulation(*manifold, symmetric_triangulation);
    }

    free_symmetry_group(manifold_sym_grp);
    if (link_sym_grp != *symmetry_group)
        free_symmetry_group(link_sym_grp);

    manifold_sym_grp    = NULL;
    link_sym_grp        = NULL;

} while (drill_one_curve(manifold, &the_desired_curves) == func_OK);
}

```