

```

/*
 * chern_simons.c
 *
 * The computation of the Chern-Simons invariant is a little
 * delicate because the formula depends on a constant which
 * must initially be supplied by the user.
 *
 * For the UI, this file provides the functions
 *
 * void      set_CS_value(   Triangulation  *manifold,
 *                          double          a_value);
 * void      get_CS_value(   Triangulation  *manifold,
 *                          Boolean         *value_is_known,
 *                          double         *the_value,
 *                          int            *the_precision,
 *                          Boolean        *requires_initialization);
 *
 * The UI calls set_CS_value() to pass to the kernel a user-supplied
 * value of the Chern-Simons invariant for the current manifold.
 *
 * The UI calls get_CS_value() to request the current value.  If the
 * current value is known (or can be computed), get_CS_value() sets
 * *value_is_known to TRUE and writes the current value and its precision
 * (the number of significant digits to the right of the decimal point)
 * to *the_value and *the_precision, respectively.  If the current value
 * is not known and cannot be computed, it sets *value_is_known to FALSE,
 * and then sets *requires_initialization to TRUE if the_value
 * is unknown because no fudge factor is available, or
 * to FALSE if the_value is unknown because the solution contains
 * negatively oriented Tetrahedra.  The UI might want to convey
 * these situations to the user in different ways.
 *
 * get_CS_value() normalizes *the_value to the range (-1/4,+1/4].
 * This is the ONLY point in code where such an adjustment is made;
 * all internal computations are done mod 0.
 *
 * The kernel manages the Chern-Simons computation by keeping track of
 * both the current value and the arbitrary constant ("fudge factor")
 * which appears in the formula.  It uses the following fields of
 * the Triangulation data structure:
 *
 * Boolean    CS_value_is_known,
 *            CS_fudge_is_known;
 * double     CS_value[2],
 *            CS_fudge[2];
 *
 * The Boolean flags indicate whether the corresponding double is
 * presently known or unknown.  To provide an estimate of precision,
 * CS_value[ultimate] and CS_value[penultimate] store the value of the
 * Chern-Simons invariant computed relative to the hyperbolic structure
 * at the ultimate and penultimate iterations of Newton's method, and
 * similarly for the fudge factor CS_fudge[].
 *
 * For the kernel, this file provides the functions
 *
 * void      compute_CS_value_from_fudge(Triangulation *manifold);
 * void      compute_CS_fudge_from_value(Triangulation *manifold);
 *
 * compute_CS_value_from_fudge() computes the CS_value in terms of
 * CS_fudge, if CS_fudge_is_known is TRUE.  (If CS_fudge_is_known is FALSE,
 * it sets CS_value_is_known to FALSE as well.)  The kernel calls this
 * function when doing Dehn fillings on a fixed Triangulation, where
 * the CS_fudge will be known (and constant) but the CS_value will be
 * changing.
 *
 * compute_CS_fudge_from_value() computes the CS_fudge in terms of
 * CS_value, if CS_value_is_known is TRUE.  (If CS_value_is_known is FALSE,
 * it sets CS_fudge_is_known to FALSE as well.)  The kernel calls this
 * function when it changes a Triangulation without changing the manifold
 * it represents.
 *
 * Bob Meyerhoff, Craig Hodgson and Walter Neumann have found at least

```

```

* two different algorithms for computing the Chern-Simons invariant.
* The following code allows easy substitution of algorithms, in the
* function compute_CS().
*
* 96/4/16 David Eppstein pointed out that when he does (1,0) Dehn filling
* on m074(1,0), SnapPea quits with the message "The argument in the
* dilogarithm function is too large to guarantee accuracy". I've modified
* the code so that it displays the message "The argument in the dilogarithm
* function is too large to guarantee an accurate value for the Chern-Simons
* invariant" but does not quit. Instead it sets
*
* manifold->CS_value_is_known = FALSE;
* or
* manifold->CS_fudge_is_known = FALSE;
*
* (as appropriate) and continues normally. [By the way, I rejected the
* idea of providing more coefficients for the series. The set of manifolds
* for which the existing coefficients do not suffice is very, very small:
* no problems arise for any of the manifolds in the cusped or closed censuses.
* (Eppstein's example of m074(1,0) is a 3-sphere, but other descriptions
* of the 3-sphere seem to work fine.) So I don't want to slow down the
* computation of the Chern-Simons invariant in the generic case for the
* sake of an almost vanishingly small set of exceptions.]
*/

#include "kernel.h"

#define CS_EPSILON 1e-8

#define LOG_TWO_PI 1.83787706640934548356

static FuncResult compute_CS(Triangulation *manifold, double value[2]);
static FuncResult algorithm_one(Triangulation *manifold, double value[2]);
static Complex alg1_compute_Fu(Triangulation *manifold, int which_approximation,
    Boolean *Li2_error_flag);
static Complex Li2(Complex w, ShapeInversion *z_history, Boolean *Li2_error_flag);
static Complex log_w_minus_k_with_history(Complex w, int k,
    double regular_arg, ShapeInversion *z_history);
static int get_history_length(ShapeInversion *z_history);
static int get_wide_angle(ShapeInversion *z_history, int requested_index);

void set_CS_value(
    Triangulation *manifold,
    double a_value)
{
    manifold->CS_value_is_known = TRUE;
    manifold->CS_value[ultimate] = a_value;
    manifold->CS_value[penultimate] = a_value;

    compute_CS_fudge_from_value(manifold);
}

void get_CS_value(
    Triangulation *manifold,
    Boolean *value_is_known,
    double *the_value,
    int *the_precision,
    Boolean *requires_initialization)
{
    if (manifold->CS_value_is_known)
    {
        *value_is_known = TRUE;
        *the_value = manifold->CS_value[ultimate];
        *the_precision = decimal_places_of_accuracy(
            manifold->CS_value[ultimate],
            manifold->CS_value[penultimate]);
        *requires_initialization = FALSE;

        /*
         * Normalize reported value to the range (-1/4, 1/4].
         */
        while (*the_value < -0.25 + CS_EPSILON)

```

```

        *the_value += 0.5;
    while (*the_value > 0.25 + CS_EPSILON)
        *the_value -= 0.5;
}
else
{
    *value_is_known          = FALSE;
    *the_value               = 0.0;
    *the_precision           = 0;
    *requires_initialization = (manifold->CS_fudge_is_known == FALSE);
}
}

void compute_CS_value_from_fudge(
    Triangulation *manifold)
{
    double computed_value[2];

    if (manifold->CS_fudge_is_known == TRUE
        && compute_CS(manifold, computed_value) == func_OK)
    {
        manifold->CS_value_is_known = TRUE;
        manifold->CS_value[ultimate] = computed_value[ultimate] + manifold->CS_fudge ✓
[ultimate];
        manifold->CS_value[penultimate] = computed_value[penultimate] + manifold->CS_fudge ✓
[penultimate];
    }
    else
    {
        manifold->CS_value_is_known = FALSE;
        manifold->CS_value[ultimate] = 0.0;
        manifold->CS_value[penultimate] = 0.0;
    }
}

void compute_CS_fudge_from_value(
    Triangulation *manifold)
{
    double computed_value[2];

    if (manifold->CS_value_is_known == TRUE
        && compute_CS(manifold, computed_value) == func_OK)
    {
        manifold->CS_fudge_is_known = TRUE;
        manifold->CS_fudge[ultimate] = manifold->CS_value[ultimate] - computed_value ✓
[ultimate];
        manifold->CS_fudge[penultimate] = manifold->CS_value[penultimate] - computed_value ✓
[penultimate];
    }
    else
    {
        manifold->CS_fudge_is_known = FALSE;
        manifold->CS_fudge[ultimate] = 0.0;
        manifold->CS_fudge[penultimate] = 0.0;
    }
}

static FuncResult compute_CS(
    Triangulation *manifold,
    double value[2])
{
    Cusp *cusp;

    /*
     * We can handle only orientable manifolds.
     */

    if (manifold->orientability != oriented_manifold)
        return func_failed;

    /*

```

```

    * Cusps must either be complete, or have Dehn filling
    * coefficients which are relatively prime integers.
    */

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)

        if (Dehn_coefficients_are_relatively_prime_integers(cusp) == FALSE)

            return func_failed;

    /*
    * Here we plug in the algorithm of our choice.
    */

    return algorithm_one(manifold, value);
}

static FuncResult algorithm_one(
    Triangulation *manifold,
    double value[2])
{
    Boolean Li2_error_flag;
    int i;
    Complex Fu[2],
        core_length_sum[2],
        complex_volume[2],
        length[2];
    int singularity_index;
    Cusp *cusp;

    /*
    * This algorithm is taken directly from Craig Hodgson's
    * preprint "Computation of the Chern-Simons invariants".
    * It extends previous implementations in that it uses
    * the shape_histories of the Tetrahedra to compute
    * the dilogarithms, which allows solutions with negatively
    * oriented Tetrahedra.
    */

    /*
    * To use the Chern-Simons formula, both the complete and filled
    * solutions must be geometric, nongeometric or flat.
    */

    for (i = 0; i < 2; i++) /* i = complete, filled */

        if (manifold->solution_type[i] != geometric_solution
            && manifold->solution_type[i] != nongeometric_solution
            && manifold->solution_type[i] != flat_solution)

            return func_failed;

    /*
    * Initialize the Li2_error_flag to FALSE.
    * If the coefficients in Li2() don't suffice to compute the dilogarithm
    * to full precision, Li2() will set Li2_error_flag to TRUE.
    */

    Li2_error_flag = FALSE;

    /*
    * Compute F(u) relative to the ultimate and penultimate
    * hyperbolic structures, to allow an estimation of precision.
    */

    for (i = 0; i < 2; i++) /* i = ultimate, penultimate */

        Fu[i] = alg1_compute_Fu(manifold, i, &Li2_error_flag);

    /*
    * If Li2() failed, return func_failed;
    */

```

```

    */

    if (Li2_error_flag == TRUE)
    {
        uAcknowledge("An argument in the dilogarithm function is too large to guarantee an
accurate value for the Chern-Simons invariant.");
        return func_failed;
    }

    /*
    * F(u) is
    *
    *      (complex volume) + pi/2 (sum of complex core lengths)
    *
    * So we subtract off the complex lengths of the core geodesics
    * to be obtain the complex volume.
    */

    for (i = 0; i < 2; i++) /* i = ultimate, penultimate */
        core_length_sum[i] = Zero;

    for (cusp = manifold->cusp_list_begin.next;
        cusp != &manifold->cusp_list_end;
        cusp = cusp->next)
    {
        compute_core_geodesic(cusp, &singularity_index, length);

        switch (singularity_index)
        {
            case 0:
                /*
                * The cusp is complete. Do nothing.
                */
                break;

            case 1:
                /*
                * Add this core length to the sum.
                */
                for (i = 0; i < 2; i++) /* i = ultimate, penultimate */
                    core_length_sum[i] = complex_plus(
                        core_length_sum[i],
                        length[i]);
                break;

            default:
                /*
                * We should never arrive here.
                */
                uFatalError("algorithm_one", "chern_simons");
        }
    }

    /*
    * (complex volume) = F(u) - (pi/2)(sum of core lengths)
    */

    for (i = 0; i < 2; i++) /* i = ultimate, penultimate */
    {
        complex_volume[i] = complex_minus(
            Fu[i],
            complex_real_mult(
                PI_OVER_2,
                core_length_sum[i]
            )
        );

        value[i] = complex_volume[i].imag / (2.0 * PI * PI);
    }

    return func_OK;
}

```

```

static Complex alg1_compute_Fu(
    Triangulation *manifold,
    int which_approximation, /* ultimate or penultimate */
    Boolean *Li2_error_flag)
{
    Complex Fu;
    Tetrahedron *tet;
    static const Complex minus_i = {0.0, -1.0};

    /*
     * We compute the function F(u), which Yoshida has proved holomorphic.
     * (See Craig's preprint mentioned above.)
     */

    /*
     * Initialize F(u) to Zero.
     */

    Fu = Zero;

    /*
     * Add up the log terms.
     */

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        Fu = complex_minus(
            Fu,
            complex_mult(
                tet->shape[ filled ]->cwl[which_approximation][0].log,
                tet->shape[ filled ]->cwl[which_approximation][1].log
            )
        );

        Fu = complex_plus(
            Fu,
            complex_mult(
                tet->shape[ filled ]->cwl[which_approximation][0].log,
                complex_conjugate(
                    tet->shape[complete]->cwl[which_approximation][1].log
                )
            )
        );

        Fu = complex_minus(
            Fu,
            complex_mult(
                tet->shape[ filled ]->cwl[which_approximation][1].log,
                complex_conjugate(
                    tet->shape[complete]->cwl[which_approximation][0].log
                )
            )
        );

        Fu = complex_minus(
            Fu,
            complex_mult(
                tet->shape[complete]->cwl[which_approximation][0].log,
                complex_conjugate(
                    tet->shape[complete]->cwl[which_approximation][1].log
                )
            )
        );
    }

    /*
     * Multiply through by one half.
     */

    Fu = complex_real_mult(0.5, Fu);

    /*
     * Add in the dilogarithms.
     */

```

```

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * To compute the dilogarithm of z, Li2() wants to be
         * passed w = log(z) / 2 pi i and the shape_history of z.
         */

        Fu = complex_plus
        (
            Fu,
            Li2
            (
                complex_div
                (
                    tet->shape[filled]->cwl[which_approximation][0].log,
                    TwoPiI
                ),
                tet->shape_history[filled],
                Li2_error_flag
            )
        );
    }

    /*
     * Multiply by -i.
     */

    Fu = complex_mult(minus_i, Fu);

    return Fu;
}

static Complex Li2(
    Complex      w,
    ShapeInversion *z_history,
    Boolean      *Li2_error_flag)
{
    /*
     * Compute the dilogarithm of z = exp(2 pi i w) as explained
     * in Craig's preprint mentioned above. Note that we use
     * the variable w instead of the z which appears in Craig's
     * preprint, to avoid confusion with the z which appears
     * in the formula for F(u).
     *
     * The term Craig calls "S" we compute in two parts
     *
     *      s0 = sum from i = 1 to infinity . . .
     *      s1 = sum from k = 1 to N . . . + Nw
     *
     * The remaining part of the formula we call
     *
     *      t = pi^2/6 + 2 pi i w - 2 pi i w log(-2 pi i w) + (pi w)^2
     */

    Complex s0,
            s1,
            s,
            t,
            w_squared,
            two_pi_i_w,
            kk,
            k_plus_w,
            k_minus_w,
            result;

    int i,
        k;

    static const Complex pi_squared_over_6 = {PI*PI/6.0, 0.0},
                        four_pi_i         = {0.0, 4.0*PI},
                        minus_pi_i         = {0.0, -PI},
                        log_minus_two_pi_i = {LOG_TWO_PI, -PI_OVER_2};

```

```

/*
 * The array a[] contains the coefficients for the infinite series
 * in s0. The constant num_terms tells how many we need to use to
 * insure accuracy (see Analysis of Convergence below).
 *
 * The following Mathematica code computed these coefficients
 * for N = 2. (I use "n" where Craig used "N" to conform both
 * to programming conventions regarding capital letters, and also
 * to Mathematica's conventions.)
 *
 *      a[i_, n_] :=
 *          N[(Zeta[2i] - Sum[k^(-2i), {k,1,n}]) / (2i(2i + 1)), 60]
 *      a2[i_] := a[i, 2]
 *      Array[a2, 30]
 *
 * Note that to get 20 significant digits in a2[30] = 6.4e-33, we
 * must request at least 53 decimal places of accuracy from
 * Mathematica, and probably a little more since the accuracy we
 * request is the accuracy to which the intermediate calculations
 * are truncated -- the final accuracy could be a little worse.
 * By the way, we really do need a lot of that accuracy even in
 * the tiny coefficients, because they will be multiplied by high
 * powers of w, and |w| may be greater than one.
 */
static const int    num_terms = 30;
static const int    n = 2;
static const double a[] = {
    0.0,
    6.58223444747044060787e-2,
    9.91161685556909575800e-4,
    4.09062377249795170123e-5,
    2.37647497144915803729e-6,
    1.63751161982593974054e-7,
    1.24738994105660169102e-8,
    1.01418480335632980259e-9,
    8.62880373230578403363e-11,
    7.59064144690016509252e-12,
    6.85041587014555123901e-13,
    6.30901702974110744035e-14,
    5.90712644809102073367e-15,
    5.60732930747841393884e-16,
    5.38501558411235458177e-17,
    5.22344536523359867175e-18,
    5.11092595568460128406e-19,
    5.03912265560217431595e-20,
    5.00200835767964640183e-21,
    4.99518851712940000071e-22,
    5.01545492014257760830e-23,
    5.06048349504093155712e-24,
    5.12862546072263579933e-25,
    5.21876054821516289501e-26,
    5.33019249317297967524e-27,
    5.46257395282628942810e-28,
    5.61585233364316675625e-29,
    5.79023077981676178469e-30,
    5.98614037451033538648e-31,
    6.20422080422041301360e-32,
    6.44530754870034591211e-33};

/*
 * Analysis of convergence.
 *
 * The i-th coefficient in the (partial) zeta function is
 *
 *      (N+1)^(-2i) + (N+2)^(-2i) + (N+3)^(-2i) + ...
 *
 * Lemma. For large i, this series may be approximated by its first
 * term (N+1)^(-2i).
 *
 * Proof. [Probably not worth reading, but I figured I ought to
 * include it.] Get an upper bound on the sum of the neglected
 * terms by comparing them to an integral:
 */

```



```

*      (N+2)^(-2i) + (N+3)^(-2i) + ...
*      < (N+2)^(-2i) + integral from x = N+2 to infinity of x^(-2i) dx
*      = (N+2)^(-2i) + (N+2)^(-2i+1)/(2i-1)
*      < (N+2)^(-2i) + (N+2)^(-2i)
*      = 2 (N+2)^(-2i)
*
* Therefore the ratio (error)/(first term) is less than
*
*      [2 (N+2)^(-2i)] / [(N+1)^(-2i)]
*      = 2 ((N+1)/(N+2))^(2i)
*
* Thus, for example, if N = 2 and i >= 10, the ratio
* (error)/(first term) will be less than 2 (3/4)^10 = 1%.
* When N = 4 we need i >= 15 to obtain 1% accuracy.
* Q.E.D.
*
*
* The preceding lemma implies that the infinite series
* for S has the same convergence behavior as the series
*
*      S' = (w/(N+1))^2i / i^2
*
* so we analyze S' instead of S. The error introduced
* by truncating the series after some i = i0 is bounded
* by the corresponding error in the geometric series
*
*      S'' = |w/(N+1)|^2i / i0^2
*
* The latter error is
*
*      (first neglected term) / (1 - ratio)
*
*      = (|w/(N+1)|^2i0 / i0^2) / (1 - |w/(N+1)|^2)
*
*      = |w/(N+1)|^2i0 / (i0^2 (1 - |w/(N+1)|^2))
*
* A quick calculation in Mathematica shows that if
* we are willing to calculate 30 terms in the series,
* then |w/(N+1)| < 0.5 implies the error will be
* less than 1e-20. In other words, the series can
* be used successfully for |w| < (N+1)/2. What values
* of z (i.e. what actual simplex shapes) does this
* correspond to?
*
* Letting w = x + iy, we get
*
*      z = exp(2 pi i (x + i y))
*      = exp(-2 pi y + 2 pi i x)
*      = exp(-2 pi y) * (cos(2 pi x) + i sin(2 pi x))
*
* In other words, at an argument of 2 pi x, the acceptable
* parameters z are those with moduli between
* exp(-2 pi sqrt(((N+1)/2)^2 - x^2)) and
* exp(+2 pi sqrt(((N+1)/2)^2 - x^2)).
*
* For N = 2:
*   When x = 0 we get values of z along the positive real axis
*   from 0.00008 to 12000.
*   When x = 1/2 we get values of z along the negative real axis
*   from -0.0001 to -7000.
*   When x = 1 we get values of z along the positive real axis
*   from 0.0008 to 1000.
*
* This is good news: it means that the 30-term series for S will
* be accurate to 1e-20 for all (reasonable) nondegenerate values
* of z. I don't foresee the need for a greater radius of
* convergence, but if one is ever needed, just switch to N = 4.
*/
/*
* According to the preceding Analysis of Convergence, our
* computations will be accurate to 1e-20 whenever
* |w| < (N+1)/2 = 3/2.
*/

```

```

if (complex_modulus(w) > 1.5)
{
    *Li2_error_flag = TRUE;
    return Zero;
}

/*
 * Note the values of w^2 and 2 pi i w.
 */

w_squared = complex_mult(w, w);
two_pi_i_w = complex_mult(TwoPiI, w);

/*
 * Compute t.
 *
 * In the third term, - 2 pi i w will lie in the strip
 * 0 < Im(- 2 pi i w) < - pi i, so we choose the argument
 * in its log to be in the range (0, - pi).
 */

t = pi_squared_over_6;

t = complex_plus(
    t,
    two_pi_i_w
);

t = complex_minus(
    t,
    complex_mult(
        two_pi_i_w,
        complex_plus(
            log_minus_two_pi_i,
            log_w_minus_k_with_history(w, 0, 0.0, z_history)
        )
    )
);

t = complex_plus(
    t,
    complex_real_mult(PI * PI, w_squared)
);

/*
 * Compute s0.
 *
 * Start with the high order terms and work backwards.
 * It's a little faster, because fewer multiplications
 * are required, and might also be a little more accurate.
 */

s0 = Zero;
for (i = num_terms; i > 0; --i)
{
    s0.real += a[i];
    s0 = complex_mult(s0, w_squared);
}
s0 = complex_mult(s0, w);

/*
 * Compute s1.
 */

s1 = Zero;
for (k = 1; k <= n; k++)
{
    kk = complex_real_mult(k, One);
    k_plus_w = complex_plus(kk, w);
    k_minus_w = complex_minus(kk, w);

    s1 = complex_plus(
        s1,

```

```

        complex_real_mult(log(k), w)
    );

    s1 = complex_minus(
        s1,
        complex_real_mult(
            0.5,
            complex_mult(
                k_plus_w,
                log_w_minus_k_with_history(w, -k, 0.0, z_history)
            )
        )
    );

    s1 = complex_plus(
        s1,
        complex_real_mult(
            0.5,
            complex_mult(
                k_minus_w,
                /*
                 * We write Craig's log(k - w), which had an
                 * argument of 0 for the regular case, as
                 * log(-1) + log(w - k), and choose arg(log(-1)) = -pi
                 * and arg(log(w - k)) = +pi for the regular case.
                 */
                complex_plus(
                    minus_pi_i,
                    log_w_minus_k_with_history(w, k, PI, z_history)
                )
            )
        )
    );

}

s1 = complex_plus(
    s1,
    complex_real_mult(n, w)
);

/*
 * Add t + (4 pi i)(s0 + s1) to get the final answer.
 */

s = complex_plus(s0, s1);
result = complex_plus(
    t,
    complex_mult(four_pi_i, s)
);

return result;
}

static Complex log_w_minus_k_with_history(
    Complex w,
    int k,
    double regular_arg,
    ShapeInversion *z_history)
{
    int which_strip;
    double estimated_argument;
    int i;

    /*
     * This function computes log(w - k), taking into account the "history"
     * of the shape z from which w is derived (z = exp(2 pi i w), as
     * explained above). That is, it takes into account z's precise
     * path through the parameter space, up to isotopy.
     *
     * regular_arg supplies the correct argument for the case of a
     * regular ideal tetrahedron, with z = (1/2) + (sqrt(3)/2)i,
     * w = 1/6, and a trivial "history". Typically regular_arg
     * will be 0 for k <= 0, and +pi for k > 0.
     */

```

```

*
* To understand what's going on here, it will be helpful to make
* yourself pictures of the z- and w-planes, as follows:
*
* z-plane.    Draw axes for the complex plane representing z.
*              Draw small circles at 0 and 1 to show where
*              z is singular.
*              Color the real axis blue from -infinity to 0, and label
*              it '0' to indicate that z crosses this segment when
*              there is a ShapeInversion with wide_angle == 0.
*              Color the real axis red from 1 to +infinity, and label
*              it '1' to indicate that z crosses this segment when
*              there is a ShapeInversion with wide_angle == 1.
*              Color the real axis green from 0 to 1, and label
*              it '2' to indicate that z crosses this segment when
*              there is a ShapeInversion with wide_angle == 2.
*
* w-plane.    Draw the preimage of the z-plane picture under the
*              map  $z = \exp(2 \pi i w)$ .
*              The singularities occur at the integer points on
*              the real axis.
*              Red half-lines labeled 1 extend from each singularity
*              downward to infinity.
*              Green half-lines labeled 2 extend from each singularity
*              upward to infinity.
*              Blue lines labelled 0 pass vertically through each
*              half-integer point on the real axis.
*
* We will use the z_history to trace the path of w through the
* w-plane picture, keeping track of the argument of  $\log(w - k)$ 
* as we go. We begin with the shape of a regular ideal tetrahedron,
* namely  $z = (1/2) + (\sqrt{3}/2) i$ ,  $w = 1/6 + 0 i$ .
*
* It suffices to keep track of the approximate argument to the
* nearest multiple of  $\pi$ , since the true argument will be within
*  $\pi/2$  of that estimate.
*
* The vertical strips in the w-plane (which are preimages of
* the halfplane  $z.\text{imag} > 0$  and  $z.\text{imag} < 0$  in the z-plane)
* are indexed by integers. Strip n is the strip extending
* from  $w.\text{real} = n/2$  to  $w.\text{real} = (n+1)/2$ .
*/

/*
* We begin at  $w = 1/6$ , and set the estimated_argument to
* regular_arg (this will typically be 0 if  $k \leq 0$ , or  $\pi$  if  $k > 0$ ,
* corresponding to Walter and Craig's choices for the case of
* positively oriented Tetrahedra).
*/

which_strip      = 0;
estimated_argument = regular_arg;

/*
* Now we read off the z_history, adjusting which_strip
* and estimated_argument accordingly.
*
* Typically the z_history will be NULL, so nothing happens here.
*
* Technical note: this isn't the most efficient way to read
* a linked list backwards, but clarity is more important than
* efficiency here, but the z_histories are likely to be so short.
*/

for (i = 0; i < get_history_length(z_history); i++)
    switch (get_wide_angle(z_history, i))
    {
        case 0:
            /*
             * If we're in an even numbered strip, move to the right.
             * If we're in an odd numbered strip, move to the left.
             * The estimated_argument does not change.
             */

```

```

        if (which_stripe % 2 == 0)
            which_stripe++;
        else
            which_stripe--;
        break;

    case 1:
        /*
         * If we're in an even numbered stripe, move to the left,
         * and if we pass under the singularity at k,
         * subtract pi from the estimated_argument.
         * If we're in an odd numbered stripe, move to the right,
         * and if we pass under the singularity at k,
         * add pi to the estimated_argument.
         */
        if (which_stripe % 2 == 0)
        {
            which_stripe--;
            if (which_stripe == 2*k - 1)
                estimated_argument -= PI;
        }
        else
        {
            which_stripe++;
            if (which_stripe == 2*k)
                estimated_argument += PI;
        }
        break;

    case 2:
        /*
         * If we're in an even numbered stripe, move to the left,
         * and if we pass over the singularity at k,
         * add pi to the estimated_argument.
         * If we're in an odd numbered stripe, move to the right,
         * and if we pass over the singularity at k,
         * subtract pi from the estimated_argument.
         */
        if (which_stripe % 2 == 0)
        {
            which_stripe--;
            if (which_stripe == 2*k - 1)
                estimated_argument += PI;
        }
        else
        {
            which_stripe++;
            if (which_stripe == 2*k)
                estimated_argument -= PI;
        }
        break;

    default:
        uFatalError("log_w_minus_k_with_history", "chern_simons");
}

/*
 * Compute log(w - k) using the estimated_argument.
 */

return (
    complex_log(
        complex_minus(
            w,
            complex_real_mult((double)k, One)
        ),
        estimated_argument
    )
);
}

static int get_history_length(
    ShapeInversion *z_history)

```

```
{
    int length;

    length = 0;

    while (z_history != NULL)
    {
        length++;
        z_history = z_history->next;
    }

    return length;
}

static int get_wide_angle(
    ShapeInversion *z_history,
    int requested_index)
{
    while (--requested_index >= 0)
        z_history = z_history->next;

    return z_history->wide_angle;
}
```