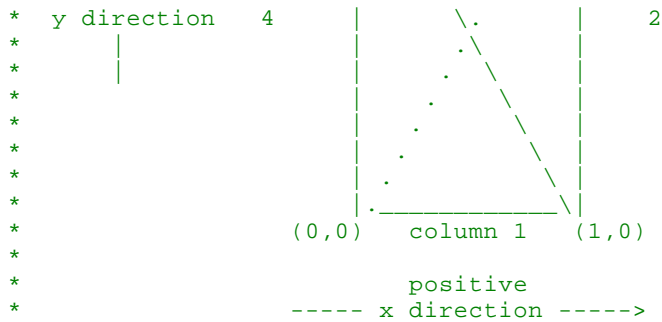


[illegible]

[illegible]

[illegible]



Relative to this coordinate system, the two strands of the link which pass the pillow will have slope either 0/1 or 1/0. In the example given above, the slope is 1/0. If, instead of double right handed twists in column 3, the bottom of the box had had double left handed twists in column 2, then the strands on the pillow would have slope 0/1. Let this slope (0/1 or 1/0) be the initial value of a fraction which we will call a/b . We begin to work our way up the rectangular box, untwisting the crossings as we encounter them. That is, we transfer the twist from the "free" part of the knot or link to the square pillow. As we do this, we keep track of the slope of the two strands of the knot or link which pass across the pillow. Each right handed twist in column 3 will introduce a twist in the pillow which takes a line of slope a/b to a line of slope $a/(a+b)$. Each left handed twist in column 2 will introduce a twist in the pillow which takes a line of slope a/b to a line of slope $(a+b)/b$. Continue in this fashion until we reach the top of the rectangular box. If the two strands which pass the pillow at the top of the box run parallel to the y axis, then (according to the Figure 1 of Hatcher-Thurston, which is reproduced near the top of this file) the value of a/b will be precisely the fraction p/q describing the 2-bridge knot or link. If the two strands which pass the pillow at the top of the box run parallel to the x axis, then we must rotate the whole figure a quarter turn; in this case fraction p/q equals $-b/a$.

It's very easy to phrase the above analysis in terms of continued fractions, but for the purposes of the program we have no need to do so. (Just read down the box picture, letting the number of consecutive crossings in a given column be a term in the continued fraction.)

```
#include "kernel.h"
```

```
/*
 * The Level data structure describes the two Tetrahedra
 * which lie at a given level in the rectangular box picture.
 */
```

```
typedef struct
```

```
{
    /*
     * tet[0] lies in the original rectangular box.
     * tet[1] lies in the complementary box.
     */
    Tetrahedron *tet[2];

    /*
     * vertex[i][j][k] is the VertexIndex of the vertex of
     * Tetrahedron i (i = 0 or 1, as in the preceding tet field)
     * which lies at (x, y) = (j, k), where x and y are defined
     * in an illustration above. The relationship between
     * the vertex numbering of the original and complementary
     * Tetrahedra is what you would expect: vertex[0][i][j]
     * of tet[0] is incident to vertex[1][i][j] of tet[1].
     */
}
```

```

    */
    VertexIndex vertex[2][2][2];

    /*
    * a/b is the current slope of the lines on the square
    * pillow, as described in the above documentation.
    * It accounts for all the crossings below this level,
    * and none of the crossings above it.
    */
    long int    a,
               b;

} Level;

static FuncResult    find_level_zero(Triangulation *manifold, Level *level_zero);
static FuncResult    position_double_bonded_tetrahedra(Tetrahedron *tet, FaceIndex i,
    FaceIndex j, Level *level_zero);
static Boolean       at_top_of_box(Level *current_level, long int *p, long int *q);
static Boolean       left_handed_double_crossing(Level *current_level);
static Boolean       right_handed_double_crossing(Level *current_level);
static FuncResult    move_to_next_level(Level *current_level);
static FuncResult    find_new_level_tetrahedra(Level *old_level, Level *new_level);
static Boolean       left_handed_crossing(Level *old_level, Level *new_level);
static Boolean       right_handed_crossing(Level *old_level, Level *new_level);
static void          interchange_x_and_y(Level *level);
static void          normal_form(long int *p, long int *q);

void two_bridge(Triangulation    *manifold,
                Boolean          *is_two_bridge,
                long int         *p,
                long int         *q)
{
    Level    current_level;

    /*
    * The overall plan is to assume the Triangulation
    * *manifold is of the form illustrated by the
    * rectangular box picture (cf. the lengthy top-of-file
    * documentation above). We'll start at the bottom
    * of the box and work our way up. If at some point
    * we find we can't fit the Triangulation into the
    * required form, we set *is_two_bridge to FALSE and
    * return. Otherwise, we build up the fraction p/q
    * as we go along, and at the end we set *is_two_bridge
    * to TRUE, set the correct values for *p and *q, and
    * return.
    */

    /*
    * To get started, find the two Tetrahedra at level 0.
    * These Tetrahedra are easily recognized by their
    * "double bond"; that is, they share two pairs of glued
    * faces. It doesn't matter which two double-bonded
    * Tetrahedra we choose: if we swap the pair at the top
    * of the rectangular box for the pair at the bottom the
    * whole rectangular box picture gets turned upside down,
    * the continued fraction expansion gets reversed, and
    * instead of the fraction p/q we get the fraction +- p'/q,
    * where p' is the inverse of p in the ring Z/q, and the
    * +- depends on whether there are an even or odd number
    * of terms in the continued fraction expansion.
    *
    * If we can't find a pair of double bonded Tetrahedra,
    * then (modulo the conjecture described above) we know
    * this isn't a two-bridge knot or link complement, and
    * we set *is_two_bridge to FALSE and return.
    */

    if (find_level_zero(manifold, &current_level) == func_failed)
    {
        *is_two_bridge = FALSE;
        return;
    }
}

```



```

}

/*
 * Now we work our way up the rectangular box until
 * we reach the top. When we reach the top, we
 * account for the final double crossing, and compute
 * the fraction p/q.
 */

while (at_top_of_box(&current_level, p, q) == FALSE)

    if (move_to_next_level(&current_level) == func_failed)
    {
        *is_two_bridge = FALSE;
        return;
    }

/*
 * For a given 2-bridge knot or link, the denominator q
 * in the fraction p/q is well-defined, but there are typically
 * four possibilities for p in the range  $-q < p < q$ . We report
 * the value of p whose absolute value is smallest
 * (if  $(p)(-p) \equiv 1 \pmod{q}$  we report the positive value).
 * This convention makes it obvious when two knots or links
 * are equivalent, and also makes it obvious when they are
 * mirror-images of each other.
 */

normal_form(p, q);

/*
 * Set *is_two_bridge to TRUE and return.
 * The function at_top_of_box() has already set *p and *q.
 */

*is_two_bridge = TRUE;
}

static FuncResult find_level_zero(
    Triangulation *manifold,
    Level *level_zero)
{
    Tetrahedron *tet;
    FaceIndex i,
              j;

    /*
     * Look for a pair of double-bonded Tetrahedra.
     * If found, try to position them as described in the rectangular
     * box picture at the top of this file. If successful,
     * fill in the fields of *level_zero. If anything goes
     * wrong, return func_failed.
     */
    /*
     * In almost all cases, there will be only one candidate (i, j)
     * for the double bond. The exception is the figure eight knot
     * complement, where the bonds at the bottom of the box might
     * be confused with the bonds at the top (that is, in the following
     * loop, the index i might refer to a face at the top of the box,
     * while the index j refers to a face at the bottom). With this
     * case in mind, we are careful to return from within the loop
     * only when position_double_bonded_tetrahedra() is successful.
     * When it's unsuccessful we keep on going.
     */

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

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

            for (j = i + 1; j < 4; j++)

                if (tet->neighbor[i] == tet->neighbor[j])

```



```

    * We hope to have a right handed double crossing in column 3.
    */
    twist = right_handed;

else

    return func_failed;

/*
 * Assign the vertices of tet[1].
 */

level_zero->vertex[1][0][0] = EVALUATE(
    tet->gluing[twist == left_handed ? j : i],
    level_zero->vertex[0][0][0]);
level_zero->vertex[1][1][0] = EVALUATE(
    tet->gluing[j],
    level_zero->vertex[0][1][0]);
level_zero->vertex[1][0][1] = EVALUATE(
    tet->gluing[i],
    level_zero->vertex[0][0][1]);
level_zero->vertex[1][1][1] = EVALUATE(
    tet->gluing[twist == left_handed ? i : j],
    level_zero->vertex[0][1][1]);

/*
 * Now check to make sure the configuration is really
 * what we are hoping for.
 */

/*
 * Are the values of level_zero->vertex[1][][] distinct?
 *
 * If so, then we've defined a valid position for tet[1].
 * (It's the only possible candidate making this Triangulation
 * look like the rectangular box picture from the top-of-file
 * documentation. In a moment we'll check whether it really works.)
 *
 * If not, then this Triangulation can't possibly be of the
 * desired form, and we return func_failed.
 */

for (i0 = 0; i0 < 4; i0++)
    for (i1 = i0 + 1; i1 < 4; i1++)
        if (level_zero->vertex[1][i0/2][i0%2]
            == level_zero->vertex[1][i1/2][i1%2])
            return func_failed;

/*
 * We know that tet->gluing[i] must map two of the three relevant
 * vertex[0][][]'s to the correct vertex[1][][]'s, and similarly
 * for tet->gluing[j], since that's how the vertex[1][][]'s were
 * defined. It remains to check the third vertex on each face.
 */

if (
    twist == left_handed
?
    (
        (EVALUATE(tet->gluing[i], level_zero->vertex[0][0][0])
         != level_zero->vertex[1][1][0])
        ||
        (EVALUATE(tet->gluing[j], level_zero->vertex[0][1][1])
         != level_zero->vertex[1][0][1])
    )
:
    /* twist == right_handed */
    (
        (EVALUATE(tet->gluing[i], level_zero->vertex[0][1][1])
         != level_zero->vertex[1][1][0])
        ||
        (EVALUATE(tet->gluing[j], level_zero->vertex[0][0][0])
         != level_zero->vertex[1][0][1])
    )
)

```

```

    )
    return func_failed;

/*
 * We now know that tet[0] and tet[1] have been positioned
 * as in the rectangular box picture. All that remains is
 * to set the fraction a/b.
 */

if (twist == left_handed)
{
    /*
     * The initial slope of the segments is 0/1, but after accounting
     * for the left handed double crossing it's 2/1.
     */

    level_zero->a = 2;
    level_zero->b = 1;
}
else /* twist == right_handed */
{
    /*
     * The initial slope of the segments is 1/0, but after accounting
     * for the right handed double crossing it's 1/2.
     */

    level_zero->a = 1;
    level_zero->b = 2;
}

return func_OK;
}

static Boolean at_top_of_box(
    Level      *current_level,
    long int    *p,
    long int    *q)
{
    /*
     * Check whether the top of the current_level glues
     * to itself in the manner corresponding to the
     * top of the rectangular box picture, as described
     * at the top of this file. If it does, set *p and *q,
     * and return TRUE. If it doesn't, return FALSE.
     *
     * In setting *p and *q, we must account for two things:
     *
     * (1) The double crossing at the top of the box,
     *     which either adds twice the denominator to
     *     the numerator, or vice versa.
     *
     * (2) The way the two arcs are attached to the
     *     remainder of the knot or link. The usual
     *     convention (see Figure 1 of Hatcher-Thurston,
     *     reproduced at the top of this file) is that
     *     the attached arcs have slope 1/0 relative to
     *     the x-y coordinate system we're using.
     *     This will be the case when we have a right
     *     handed double crossing in column 3. When
     *     we have a left handed double crossing in
     *     column 2, we must rotate the whole box a
     *     quarter turn about its vertical axis; this
     *     replaces the slope a/b with -b/a.
     *
     * Note that at_top_of_box() needn't worry about any
     * error conditions -- that's the job of move_to_next_level().
     */

    if (left_handed_double_crossing(current_level))
    {
        *p = - current_level->b;
        *q = current_level->a + 2 * current_level->b;
        return TRUE;
    }

```

```

}
if (right_handed_double_crossing(current_level))
{
    *p = current_level->a;
    *q = current_level->b + 2 * current_level->a;
    return TRUE;
}

return FALSE;
}

static Boolean left_handed_double_crossing(
Level      *current_level)
{
Tetrahedron *tet0,
             *tet1;
Permutation gluingA,
            gluingB;

/*
 * If we have a left handed double crossing in column 2, the
 * top faces of current_level->tet[0] and current_level->tet[1]
 * will be glued as shown:
 */
*
*   (0,0) |-----1-|---->(1,0)
*         / \       .
*        / 3     B' .
*       /---\   .
*      / 3     tet[1]
*     v A' .
*     ^
*     |
*     |
*     |
*   (0,1) -----2-|---->(1,1)
*                   .
*                  A .
*                 / \
*                / 3
*               / \ tet[0]
*              / 3
*             / \
*           (0,0) |-----1-|---->(1,0)
*
* The pattern of face identifications must be exactly this,
* so it is very easy to check.
*/

tet0 = current_level->tet[0];
tet1 = current_level->tet[1];

if (tet0->neighbor[current_level->vertex[0][0][0]] != tet1 ||
    tet0->neighbor[current_level->vertex[0][1][1]] != tet1)
    return FALSE;

gluingA = tet0->gluing[current_level->vertex[0][0][0]];
gluingB = tet0->gluing[current_level->vertex[0][1][1]];

if (gluingA != CREATE_PERMUTATION(
    current_level->vertex[0][0][0], current_level->vertex[1][1][0],
    current_level->vertex[0][0][1], current_level->vertex[1][0][1],
    current_level->vertex[0][1][0], current_level->vertex[1][0][0])
```

[illegible]

```

    * computed level, and new_level be a pointer to
    * the new level we are hoping to find. At the end
    * of the function we will, if successful, copy
    * the contents of *new_level into *current_level,
    * and return func_ok. Otherwise we'll leave *current_level
    * unmodified, and return func_failed.
    */

old_level = current_level;
new_level = &the_new_level;

/*
 * The rectangular box picture described in the
 * documentation at the top of this file shows
 * how each level glues to the next level.
 * Keep a copy of that picture handy as you read
 * this documentation.
 */

/*
 * First find the two tetrahedra at the new_level,
 * and set the fields new_level->tet[0] and new_level->tet[1].
 */

if (find_new_level_tetrahedra(old_level, new_level) == func_failed)
    return func_failed;

/*
 * Now see whether we can label the vertices of
 * new_level->tet[0] and new_level->tet[1] in such a way
 * as to realize either a left handed crossing in column 2 or
 * a right handed crossing in column 3.
 */

if (left_handed_crossing(old_level, new_level) == TRUE)
{
    /*
     * The left handed crossing twists the pillow in such
     * a way that the segments of slope a/b at the old level
     * get taken to segments of slope (a+b)/b at the new level.
     */

    new_level->a = old_level->a + old_level->b;
    new_level->b = old_level->b;

    /*
     * We're done!
     */

    *current_level = *new_level;
    return func_OK;
}

if (right_handed_crossing(old_level, new_level) == TRUE)
{
    /*
     * The right handed crossing twists the pillow in such
     * a way that the segments of slope a/b at the old level
     * get taken to segments of slope a/(a+b) at the new level.
     */

    new_level->a = old_level->a;
    new_level->b = old_level->a + old_level->b;

    /*
     * We're done!
     */

    *current_level = *new_level;
    return func_OK;
}

/*
 * Oh, well. This isn't a 2-bridge knot or link complement.

```



```

    */

    return func_failed;
}

static FuncResult find_new_level_tetrahedra(
    Level *old_level,
    Level *new_level)
{
    int i,
        j;

    /*
     * Regardless of whether we have a left handed crossing in
     * column 2 or a right handed crossing in column 3, the face of
     * old_level->tet[0] opposite vertex[0][1][1] will glue to
     * new_level->tet[0], and the face of old_level->tet[0] opposite
     * vertex[0][0][0] will glue to new_level->tet[1].
     */

    new_level->tet[0] = old_level->tet[0]->neighbor[old_level->vertex[0][1][1]];
    new_level->tet[1] = old_level->tet[0]->neighbor[old_level->vertex[0][0][0]];

    /*
     * Do we have two distinct new Tetrahedra?
     * (The new Tetrahedra couldn't possibly equal any that occur at
     * lower levels (because the latter have all their faces accounted
     * for), but, if this isn't a 2-bridge knot or link complement,
     * then the two new Tetrahedra might coincide with each other,
     * or with one of the Tetrahedra at the old_level. If they do,
     * return func_failed.
     */

    if (new_level->tet[0] == new_level->tet[1])
        return func_failed;

    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            if (new_level->tet[i] == old_level->tet[j])
                return func_failed;

    return func_OK;
}

static Boolean left_handed_crossing(
    Level *old_level,
    Level *new_level)
{
    int i,
        j,
        k;
    Permutation gluing;

    /*
     * If we do in fact have a left handed crossing in column 2,
     * the top surface of the old level will glue to the bottom
     * surface of the new level as illustrated below.
     *
     * For greater clarity, the illustration supresses the diagonals
     * on the top surface of the new level and and bottom surface of the
     * old level. They are irrelevant to the present gluing.
     *
     * Each double square in the illustration is really a tetrahedron
     * (thought of as a 2-complex, not a 3-complex). We are describing
     * a map from one tetrahedron to another. You may also think of
     * it as a map from one square pillowcase to another, if you prefer.
     * In any case, you don't have to study the illustration too
     * carefully, because it's clear that the net effect of the
     * map is to leave vertices (0, 0) and (0, 1) alone, and interchange
     * (1, 0) and (1, 1).
     *
     *          (0,1)          (1,1)          (0,1)
    */

```



```

    * We want to leave (0, 0) and (0, 1) alone,
    * and swap (1, 0) and (1,1).
    */

    for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
            new_level->vertex[i][j][k]
                = EVALUATE(gluing, old_level->vertex[0][j][j^k]);
}

/*
 * Given the above definitions of the new_level->vertex_index[][][],
 * check whether gluings C and D are what they should be.
 */

for (i = 0; i < 2; i++)
{
    /*
     * "gluing" will be gluing C when i = 0 and gluing D when i = 1.
     */

    gluing = old_level->tet[1]->gluing[old_level->vertex[1][i][!i]];

    for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
            if (new_level->vertex[i][j][k]
                != EVALUATE(gluing, old_level->vertex[1][j][j^k]))
                return FALSE;
}

/*
 * We've checked that the tet->neighbor and tet->gluing fields
 * correspond to a left handed crossing in column 2, and we've
 * set the new_level->vertex[][][] fields, so return TRUE.
 */

return TRUE;
}

static Boolean right_handed_crossing(
    Level *old_level,
    Level *new_level)
{
    Boolean result;

    /*
     * right_handed_crossing() should be identical to
     * left_handed_crossing(), except that the roles of the
     * x and y coordinates are reversed. So in the interest
     * of concise, easily modifiable code, it makes sense to
     * write the former as a function call to the latter.
     */

    interchange_x_and_y(old_level);

    result = left_handed_crossing(old_level, new_level);

    interchange_x_and_y(old_level);
    interchange_x_and_y(new_level);

    return result;
}

static void interchange_x_and_y(
    Level *level)
{
    int i;
    VertexIndex temp;

    for (i = 0; i < 2; i++)
    {
        temp
            = level->vertex[i][0][1];
    }

```

```

        level->vertex[i][0][1] = level->vertex[i][1][0];
        level->vertex[i][1][0] = temp;
    }
}

static void normal_form(
    long int    *p,
    long int    *q)
{
    long int    pp[4];
    int         i;

    /*
     * normal_form() accepts a fraction p/q in lowest terms
     * satisfying  $-q < p < q$ . Typically the range  $(-q, q)$ 
     * contains four values of p such that the corresponding
     * fractions p/q represent the same knot or link.
     * normal_form() replaces the original p with an equivalent
     * one of minimal absolute value. In case of ties, it
     * chooses the positive value. This convention makes it
     * obvious when two knots are equivalent, and when they are
     * mirror-images.
     */

    /*
     * Let pp[0] be a candidate for p in the range  $(0, q)$ .
     */

    pp[0] = (*p > 0) ? *p : *p + *q;

    /*
     * Let pp[1] be the inverse of pp[0] in ring  $\mathbb{Z}/q$ .
     * pp[1] will lie in the range  $(0, q)$ .
     */
    pp[1] = Zq_inverse(pp[0], *q);

    /*
     * Let pp[2] and pp[3] be candidates for p in the range  $(-q, 0)$ .
     */
    pp[2] = pp[0] - *q;
    pp[3] = pp[1] - *q;

    /*
     * Let *p be the value of pp[0-3] with the smallest absolute value.
     * In case of a tie, choose the positive value.
     */
    for (i = 0; i < 4; i++)
        if (ABS(pp[i]) < ABS(*p)
            || (ABS(pp[i]) == ABS(*p) && pp[i] > 0))
            *p = pp[i];
}

```