

```

/*
 * punctured_torus_bundles.c
 *
 * Please see function descriptions in SnapPea.h.
 */

#include "kernel.h"

#define MAX_NAME_LENGTH      31
#define BIG_BUNDLE_NAME     "untitled bundle"

void bundle_LR_to_monodromy(
    LRFactorization *anLRFactorization,
    MatrixInt22      aMonodromy)
{
    int i,
        temp;

    /*
     * The factorization should be available.
     */
    if (anLRFactorization->is_available == FALSE)
        uFatalError("bundle_LR_to_monodromy", "punctured_torus_bundles");

    /*
     * Initialize aMonodromy to the identity.
     */
    aMonodromy[0][0] = 1;
    aMonodromy[0][1] = 0;
    aMonodromy[1][0] = 0;
    aMonodromy[1][1] = 1;

    /*
     * Right multiply by the LR factors.
     */

    for (i = 0; i < anLRFactorization->num_LR_factors; i++)
        switch (anLRFactorization->LR_factors[i])
        {
            case 'L':
            case 'l':
                /*
                 * ( a  b ) ( 1  0 ) = ( a+b  b )
                 * ( c  d ) ( 1  1 )   ( c+d  d )
                 */
                aMonodromy[0][0] += aMonodromy[0][1];
                aMonodromy[1][0] += aMonodromy[1][1];
                break;

            case 'R':
            case 'r':
                /*
                 * ( a  b ) ( 1  1 ) = ( a  a+b )
                 * ( c  d ) ( 0  1 )   ( c  c+d )
                 */
                aMonodromy[0][1] += aMonodromy[0][0];
                aMonodromy[1][1] += aMonodromy[1][0];
                break;

            default:
                uFatalError("bundle_LR_to_monodromy", "punctured_torus_bundles");
        }

    /*
     * If the determinant should be negative, then
     *
     * ( 0  1 ) ( a  b ) = ( c  d )
     * ( 1  0 ) ( c  d )   ( a  b )
     */
    if (anLRFactorization->negative_determinant == TRUE)
    {
        temp
            = aMonodromy[0][0];

```

```

        aMonodromy[0][0]    = aMonodromy[1][0];
        aMonodromy[1][0]    = temp;

        temp                = aMonodromy[0][1];
        aMonodromy[0][1]    = aMonodromy[1][1];
        aMonodromy[1][1]    = temp;
    }

    /*
     * If the trace should be negative, then
     *
     *       $\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} -a & -b \\ -c & -d \end{pmatrix}$ 
     */
    if (anLRFactorization->negative_trace == TRUE)
    {
        aMonodromy[0][0] = -aMonodromy[0][0];
        aMonodromy[0][1] = -aMonodromy[0][1];
        aMonodromy[1][0] = -aMonodromy[1][0];
        aMonodromy[1][1] = -aMonodromy[1][1];
    }
}

void bundle_monodromy_to_LR(
    MatrixInt22    aMonodromy,
    LRFactorization **anLRFactorization)
{
    int            a,
                  b,
                  c,
                  d,
                  aa,
                  bb,
                  cc,
                  dd,
                  t,
                  theNumFactors;
    Boolean        theTraceWasNegative,
                  theDeterminantWasNegative;

    /*
     * Copy the entries of aMonodromy into the variables a, b, c and d.
     * This makes the notation more concise, and also means we don't
     * have to worry about overwriting aMonodromy.
     */
    a = aMonodromy[0][0];
    b = aMonodromy[0][1];
    c = aMonodromy[1][0];
    d = aMonodromy[1][1];

    /*
     * Is the matrix OK?
     *
     * The factorization is available iff
     *
     *       $\det = +1$  and  $|\text{trace}| \geq 2$ 
     * or
     *       $\det = -1$  and  $|\text{trace}| > 0$ .
     *
     * I'm pretty sure the manifold cannot be hyperbolic when
     *  $(\det = 1 \text{ and } |\text{trace}| \leq 2)$  or  $(\det = -1 \text{ and } \text{trace} = 0)$ , but
     * I don't know the proof. Note that we factor the  $\det = 1$ 
     * and  $|\text{trace}| = 2$  case even though it's not hyperbolic.
     */
    switch (a*d - b*c)
    {
        case +1:
            if (a + d < 2 && a + d > -2)
            {
                (*anLRFactorization) = alloc_LR_factorization(0);
                (*anLRFactorization)->is_available = FALSE;
                (*anLRFactorization)->negative_determinant = FALSE;
                (*anLRFactorization)->negative_trace = (a + d < 0);
            }
        }
    }
}

```

```

        return;
    }
    break;

case -1:
    if (a + d == 0)
    {
        (*anLRFactorization) = alloc_LR_factorization(0);
        (*anLRFactorization)->is_available      = FALSE;
        (*anLRFactorization)->negative_determinant = TRUE;
        (*anLRFactorization)->negative_trace      = FALSE;
        return;
    }
    break;

default:
    (*anLRFactorization) = alloc_LR_factorization(0);
    (*anLRFactorization)->is_available      = FALSE;
    (*anLRFactorization)->negative_determinant = (a*d - b*c < 0);
    (*anLRFactorization)->negative_trace      = (a + d < 0);
    return;
}

/*
 * Step 1. Make the trace positive.
 *
 * If the trace is negative, factor out -I.
 *
 *      ( a  b )   ( -1  0 ) ( -a -b )
 *      ( c  d )   (  0 -1 ) ( -c -d )
 *
 * Note that -I lies in the center of GL(2,Z); in particular,
 * it commutes with any matrices we may later conjugate by.
 */
if (a + d < 0)
{
    a = -a;
    b = -b;
    c = -c;
    d = -d;
    theTraceWasNegative = TRUE;
}
else
    theTraceWasNegative = FALSE;

/*
 * Step 2. Make a >= d.
 *
 * If a < d, conjugate to swap a and d.
 *
 *      ( 0  1 ) ( a  b ) ( 0 -1 ) = ( d -c )
 *      (-1  0 ) ( c  d ) ( 1  0 )   (-b a )
 */
if (a < d)
{
    t = a;
    a = d;
    d = t;

    t = b;
    b = -c;
    c = -t;
}

/*
 * Step 3. Make d nonnegative as well.
 *
 * At this point we know
 *
 *      trace = a + d > 0
 *
 * and
 *
 *      a >= d,
 *
 * which together imply that a > 0.

```

```

* We'd like to conjugate so that d >= 0 as well.
*
* Lemma. If d < 0, then either 0 < |b| < a or 0 < |c| < a.
*
* Proof. If d < 0, then a + d > 0 implies a > -d > 0, hence a >= 2.
* Neither b nor c can be zero, since then we'd have |det| = |ad - bc|
* = |ad| = |a|*|d| >= 2. On the other hand, if both |b| >= a
* and |c| >= a, then |det| = |ad - bc| >= |bc| - |ad| >= |aa| - |ad|
* = a*(a + d) >= 2*1 = 2. QED
*
* The lemma implies that we can do one of the following conjugations
* to increase the value of d without making a negative. Repeat
* until both a and d are nonnegative.
*
*      ( 1 0 ) ( a b ) ( 1 0 ) = ( a-b    b )
*      ( 1 1 ) ( c d ) (-1 1 )   (c+a-b-d d+b )
*
*      ( 1 0 ) ( a b ) ( 1 0 ) = ( a+b    b )
*      (-1 1 ) ( c d ) ( 1 1 )   (c-a-b+d d-b )
*
*      ( 1 -1 ) ( a b ) ( 1 1 ) = ( a-c    b+a-c-d )
*      ( 0 1 ) ( c d ) ( 0 1 )   ( c      d+c )
*
*      ( 1 1 ) ( a b ) ( 1 -1 ) = ( a+c    b-a-c+d )
*      ( 0 1 ) ( c d ) ( 0 1 )   ( c      d-c )
*
* Note: It may no longer be true that a >= d, but that's OK.
*/
while (d < 0)
{
    if (b > 0 && b < a)          /* use +b */
    {
        c += a - b - d;
        a -= b;
        d += b;
    }
    else if (b < 0 && b > -a)     /* use -b */
    {
        c += d - a - b;
        a += b;
        d -= b;
    }
    else if (c > 0 && c < a)      /* use +c */
    {
        b += a - c - d;
        a -= c;
        d += c;
    }
    else if (c < 0 && c > -a)     /* use -c */
    {
        b += d - a - c;
        a += c;
        d -= c;
    }
    else
        uFatalError("bundle_monodromy_to_LR", "punctured_torus_bundles");
}

/*
* Step 4. Make b and c nonnegative as well.
*/
if (b >= 0 && c >= 0)
{
    /* nothing to do here! */
}
else if (b <= 0 && c <= 0)
{
    /*
    * Conjugate using
    *
    *      ( 0 1 ) ( a b ) ( 0 -1 ) = ( d -c )
    *      (-1 0 ) ( c d ) ( 1 0 )   (-b a )
    */
    t = a;

```

```

    a = d;
    d = t;

    t = b;
    b = -c;
    c = -t;
}
else
{
    /*
     * b and c have opposite signs. This implies det > 0.
     * Hence det = +1, {b,c} = {-1,+1}, and {a,d} = {n,0}.
     * Furthermore, when det = +1 we handle only trace >= 2,
     * so n >= 2. Use one of the conjugations from Step 3
     * to make b and c nonnegative while maintaining the
     * nonnegativity of a and d.
     */
    if (b == +1) /* && c == -1 */
    {
        if (a >= 2) /* && d == 0 */
        {
            c += a - b - d;
            a -= b;
            d += b;
        }
        else /* a == 0 && d >= 2 */
        {
            c += d - a - b;
            a += b;
            d -= b;
        }
    }
    else /* b == -1 && c == +1 */
    {
        if (a >= 2) /* && d == 0 */
        {
            b += a - c - d;
            a -= c;
            d += c;
        }
        else /* a == 0 && d >= 2 */
        {
            b += d - a - c;
            a += c;
            d -= c;
        }
    }
}

/*
 * Step 5. Make the determinant positive by factoring if necessary
 *
 *      ( a  b ) = ( 0  1 ) ( c  d )
 *      ( c  d ) = ( 1  0 ) ( a  b )
 *
 * Note that (0, 1; 1, 0) does not commute with most matrices
 * in GL(2,Z), so it's important that we factored it out *after*
 * doing all necessary conjugations.
 */
if (a*d - b*c < 0)
{
    t = a;
    a = c;
    c = t;

    t = b;
    b = d;
    d = t;

    theDeterminantWasNegative = TRUE;
}
else
    theDeterminantWasNegative = FALSE;

```

```

/*
 * Step 6. Now that the matrix has no negative entries, we may factor
 * it as a product of L's and R's
 *
 *          L = ( 1  0 )          R = ( 1  1 )
 *              ( 1  1 )          ( 0  1 )
 *
 * Note that factoring out an L (resp. R) corresponds to subtracting
 * the first row from the second (resp. the second from the first).
 *
 *          ( a  b ) = ( 1  0 ) ( a  b )
 *          ( c  d )   ( 1  1 ) (c-a d-b)
 *
 *          ( a  b ) = ( 1  1 ) (a-c b-d)
 *          ( c  d )   ( 0  1 ) ( c  d )
 *
 * Lemma. If a, b, c and d are all nonnegative and det = +1,
 * then either
 *
 * (1) a <= c and b <= d,
 * (2) a >= c and b >= d, or
 * (3) a = d = 1 and b = c = 0.
 *
 * Comment. In case (1) we will factor out an L, in case (2) we
 * will factor out an R, and in case (3) we've reached the identity
 * and we're done. The algorithm has the flavor of the Euclidean
 * algorithm for finding the greatest common divisor of two positive
 * integers.
 *
 * Proof.
 * If a = c then one of conditions (1) or (2) must be satisfied,
 * according to whether b <= d or b >= d.
 * If a < c, then det = ad - bc = +1 implies that b < d, and
 * condition (1) is satisfied.
 * If a > c, then either
 *   b >= d, in which case condition (2) is satisfied, or
 *   b < d, in which case det = ad - bc >= (c+1)(b+1) - bc
 *           = b + c + 1, which implies b = c = 0, hence a = d = 1.
 * QED
 */

/*
 * First an error check.
 */
if (a < 0 || b < 0 || c < 0 || d < 0)
    uFatalError("bundle_monodromy_to_LR", "punctured_torus_bundles");

/*
 * Take a dry run through the algorithm to count
 * how many factors we'll need.
 */
aa = a;
bb = b;
cc = c;
dd = d;
theNumFactors = 0;
while (aa != 1 || bb != 0 || cc != 0 || dd != 1)
{
    if (aa <= cc && bb <= dd)
    {
        cc -= aa;
        dd -= bb;
        theNumFactors++;
    }
    if (aa >= cc && bb >= dd)
    {
        aa -= cc;
        bb -= dd;
        theNumFactors++;
    }
}

/*
 * Allocate the LRFactorization.

```

```

    */
    *anLRFactorization = alloc_LR_factorization(theNumFactors);

    /*
     * Record the original trace and determinant.
     */
    (*anLRFactorization)->is_available      = TRUE;
    (*anLRFactorization)->negative_determinant = theDeterminantWasNegative;
    (*anLRFactorization)->negative_trace     = theTraceWasNegative;

    /*
     * Repeat the factorization, recording the factors
     * in the LR_factors array.
     */
    theNumFactors = 0;
    while (a != 1 || b != 0 || c != 0 || d != 1)
    {
        if (a <= c && b <= d)
        {
            c -= a;
            d -= b;
            (*anLRFactorization)->LR_factors[theNumFactors++] = 'L';
        }
        if (a >= c && b >= d)
        {
            a -= c;
            b -= d;
            (*anLRFactorization)->LR_factors[theNumFactors++] = 'R';
        }
    }

    /*
     * All done!
     */
}

```

```

LRFactorization *alloc_LR_factorization(
    int aNumFactors)
{
    LRFactorization *anLRFactorization;

    anLRFactorization = NEW_STRUCT(LRFactorization);
    anLRFactorization->num_LR_factors = aNumFactors;
    if (aNumFactors > 0)
        anLRFactorization->LR_factors = NEW_ARRAY(aNumFactors, char);
    else
        anLRFactorization->LR_factors = NULL;

    return anLRFactorization;
}

```

```

void free_LR_factorization(
    LRFactorization *anLRFactorization)
{
    if (anLRFactorization != NULL)
    {
        if (anLRFactorization->LR_factors != NULL)
            my_free(anLRFactorization->LR_factors);

        my_free(anLRFactorization);
    }
}

```

```

Triangulation *triangulate_punctured_torus_bundle(
    LRFactorization *anLRFactorization)
{
    Boolean          theFactorizationContainsAnL,
                    theFactorizationContainsAnR;
    int              n, /* number of tetrahedra */
                    i,
                    j,

```

```
k,
l,
m,
image[4][4],
signed_intersections[2];
TriangulationData *data;
Triangulation      *manifold;
Tetrahedron        *tet;
PeripheralCurve     c;
MatrixInt22         change_matrices[1];
long                m0,
                    m1;
```

```
/*
 * If the LR factorization is not available [because
 * ( $\det(\text{monodromy}) = +1$  and  $| \text{trace}(\text{monodromy}) | < 2$ ) or
 * ( $\det(\text{monodromy}) = -1$  and  $| \text{trace}(\text{monodromy}) | < 1$ ] ,
 * then return NULL.
 */
if (anLRFactorization->is_available == FALSE)
    return NULL;

/*
 * If the manifold is nonhyperbolic (because  $\det(\text{monodromy}) = +1$ 
 * and  $| \text{trace}(\text{monodromy}) | = 2$ , in which case the LR factors are all
 * L's or all R's and the manifold splits open along an embedded
 * torus to yield a thrice-punctured sphere cross a circle), then
 * return NULL.
 */
theFactorizationContainsAnL = FALSE;
theFactorizationContainsAnR = FALSE;
for (i = 0; i < anLRFactorization->num_LR_factors; i++)
    switch (anLRFactorization->LR_factors[i])
    {
        case 'L':
            theFactorizationContainsAnL = TRUE;
            break;
        case 'R':
            theFactorizationContainsAnR = TRUE;
            break;
        default:
            uFatalError("triangulate_punctured_torus_bundle", "punctured_torus_bundles");
    }
};

if
(
    anLRFactorization->negative_determinant == TRUE ?
    (
        theFactorizationContainsAnL == FALSE
        && theFactorizationContainsAnR == FALSE
    ) :
    (
        theFactorizationContainsAnL == FALSE
        || theFactorizationContainsAnR == FALSE
    )
)
return NULL;

/*
 * To triangulate the punctured torus bundle, imagine wrapping
 * an almost flattened ideal tetrahedron onto a punctured torus.
```

```
*
 * The tetrahedron covers the punctured torus exactly once.
 * The tetrahedron's ideal vertices coincide with the puncture,
```



```

* and the edges labelled u (resp. v) become identified.
* The homology classes of u and v (directed as shown) define
* the ideal tetrahedron's position on the punctured torus.
*
* Now imagine an infinite stack of such ideal tetrahedra.
* The top surface of tetrahedron i glues to the bottom surface of
* tetrahedron i+1. We want to position the ideal tetrahedra so that
* triangular faces glue to triangular faces. More specifically,
*
*     If LR_factors[i mod n] == 'L'
*
*         u[i+1] = u[i] + v[i]
*         v[i+1] =          v[i]
*
*     If LR_factors[i mod n] == 'R'
*
*         u[i+1] = u[i]
*         v[i+1] = u[i] + v[i]
*
* where n is the number of LR_factors.
*
* When we do these gluings, we get a triangulation for a punctured
* torus cross a line. If we then mod out by the translation
* which takes tetrahedron i to tetrahedron i+n in such a way
* that u[i] → u[i+n] and v[i] → v[i+n], we get a triangulation
* for a punctured torus bundle over the circle. Let's compute
* its monodromy.
*
* If we write the u[i]'s and v[i]'s as the elements of row vectors
* (u[i] v[i]), then we can express the above relations as matrix
* equations.
*
*     If LR_factors[i mod n] == 'L'
*
*         ( u[i+1]  v[i+1] ) = ( u[i]  v[i] ) ( 1  0 )
*                                         ( 1  1 )
*
*     If LR_factors[i mod n] == 'R'
*
*         ( u[i+1]  v[i+1] ) = ( u[i]  v[i] ) ( 1  1 )
*                                         ( 0  1 )
*
* Composing n such relations, we obtain the position of
* tetrahedron i+n as of the position of tetrahedron i times
* the product of the LR_factors.
*
*         ( 1  1 ) ( 1  0 )
* ( u[i+n]  v[i+n] ) = ( u[i]  v[i] ) (      ) (      ) ...
*         ( 0  1 ) ( 1  1 )
*
*         ( product )
* ( u[i+n]  v[i+n] ) = ( u[i]  v[i] ) ( of all )
*         ( LR_factors )
*
* In other words, the first (resp. second) column of the product
* of the LR_factors expresses u[i+n] (resp. v[i+n]) as a linear
* combination of u[i] and v[i]. In other words, the product of
* the LR_factors is the monodromy of the bundle, expressed relative
* to the basis (u[i], v[i]).
*
* The algorithm is easily modified to accomodate negative
* determinant and/or trace.
*
* If the determinant is to be negative, then when i + 1 = 0 (mod n)
* we interchange the usual formulas for u[i+1] and v[i+1].
*
* If the trace is to be negative, then when i + 1 = 0 (mod n)
* we negate the usual formulas for u[i+1] and v[i+1].
*
* These changes still map triangles to triangles, and if both the
* trace and the determinant are negative, we can do them in either
* order. They have the desired effect on the monodromy matrix.

```

```

* For example, if both the determinant and the trace are to be
* negative, the formula for (u[n] v[n]) becomes
*
*
*      ( product ) ( 0 1 ) (-1 0 )
* ( u[n] v[n] ) = ( u[0] v[0] ) ( of all ) ( ) ( )
*      ( LR_factors ) ( 1 0 ) ( 0 -1 )
*
* Comment. The topology of the bundle depends only on the cyclic
* word of LR_factors (including the possible factors for negative
* determinant and trace), even though the monodromy matrix itself
* (the matrix product on the right hand side above) depends on how
* you break the cyclic word into a linear word. The different
* monodromy matrices are of course conjugates of one another.
*/

/*
* The plan is to describe the bundle as a TriangulationData structure,
* and then let data_to_triangulation() create the Triangulation
* itself.
*/

/*
* Let n be the number of tetrahedra.
*/
n = anLRFactorization->num_LR_factors;

/*
* Set up the header.
*/
data = NEW_STRUCT(TriangulationData);
data->name = NULL;
data->num_tetrahedra = n;
data->solution_type = not_attempted;
data->volume = 0.0;
data->orientability = (anLRFactorization->negative_determinant == TRUE) ?
nonorientable_manifold : oriented_manifold;
data->CS_value_is_known = FALSE;
data->CS_value = -1.0;
data->num_or_cusps = (anLRFactorization->negative_determinant == TRUE) ? 0 : 1;
data->num_nonor_cusps = (anLRFactorization->negative_determinant == TRUE) ? 1 : 0;
data->cusp_data = NULL;
data->tetrahedron_data = NULL;

/*
* Set up the name.
*/
if (anLRFactorization->num_LR_factors <= MAX_NAME_LENGTH - 3)
{
    data->name = NEW_ARRAY(3 + n + 1, char);
    data->name[0] = 'b';
    data->name[1] = (anLRFactorization->negative_determinant == TRUE) ? '-' : '+';
    data->name[2] = (anLRFactorization->negative_trace == TRUE) ? '-' : '+';
    for (i = 0; i < n; i++)
        data->name[3 + i] = anLRFactorization->LR_factors[i];
    data->name[3 + n] = 0;
}
else
{
    data->name = NEW_ARRAY(strlen(BIG_BUNDLE_NAME) + 1, char);
    strcpy(data->name, BIG_BUNDLE_NAME);
}

/*
* Set up the cusp.
*/
data->cusp_data = NEW_ARRAY(1, CuspData);
data->cusp_data->topology = (anLRFactorization->negative_determinant == TRUE) ?
Klein_cusp : torus_cusp;
data->cusp_data->m = 0.0;
data->cusp_data->l = 0.0;

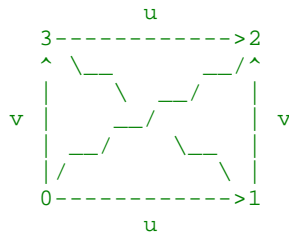
/*
* Set up the tetrahedra.
*

```

```

* Label the vertices {0,1,2,3} as shown.
* Each face gets the index of the opposite vertex.
* This indexing scheme gives each tetrahedron
* the right_handed Orientation.

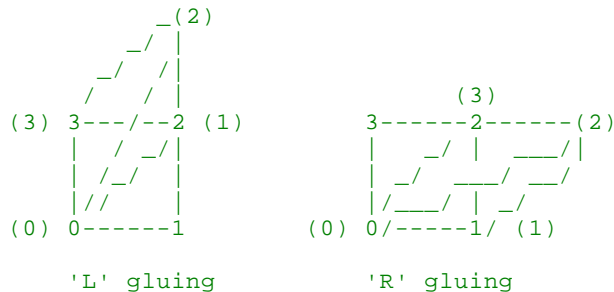
```



```

* To understand the gluings, it's helpful to draw tetrahedron i
* on a piece of paper using one color ink, and then draw
* tetrahedron i+1 on top of it using another color ink.
* You'll have to make two such pictures, one for an 'L'
* gluing and one for an 'R' gluing. The following illustrations
* show the general idea. The vertex indices for tetrahedron i+1
* are shown in parentheses -- in your own picture you can write
* the indices in the second color ink and omit the parentheses.

```



```

*/
data->tetrahedron_data = NEW_ARRAY(n, TetrahedronData);
for (i = 0; i < n; i++)
{
    data->tetrahedron_data[i].neighbor_index[0] = (i + (n-1)) % n;
    data->tetrahedron_data[i].neighbor_index[1] = (i + 1) % n;
    data->tetrahedron_data[i].neighbor_index[2] = (i + (n-1)) % n;
    data->tetrahedron_data[i].neighbor_index[3] = (i + 1) % n;

    /*
     * Set up the gluings assuming det = +1 and trace > 0, and then
     * if necessary factor in det = -1 and/or trace < 0 later.
     */
    if (anLRFactorization->LR_factors[i] == 'L')
    {
        data->tetrahedron_data[i].gluing[1][0] = 0;
        data->tetrahedron_data[i].gluing[1][1] = 2;
        data->tetrahedron_data[i].gluing[1][2] = 1;
        data->tetrahedron_data[i].gluing[1][3] = 3;

        data->tetrahedron_data[(i+1)%n].gluing[2][0] = 0;
        data->tetrahedron_data[(i+1)%n].gluing[2][1] = 2;
        data->tetrahedron_data[(i+1)%n].gluing[2][2] = 1;
        data->tetrahedron_data[(i+1)%n].gluing[2][3] = 3;

        data->tetrahedron_data[i].gluing[3][0] = 3;
        data->tetrahedron_data[i].gluing[3][1] = 1;
        data->tetrahedron_data[i].gluing[3][2] = 2;
        data->tetrahedron_data[i].gluing[3][3] = 0;

        data->tetrahedron_data[(i+1)%n].gluing[0][0] = 3;
        data->tetrahedron_data[(i+1)%n].gluing[0][1] = 1;
        data->tetrahedron_data[(i+1)%n].gluing[0][2] = 2;
        data->tetrahedron_data[(i+1)%n].gluing[0][3] = 0;
    }
    else /* 'R' */
    {
        data->tetrahedron_data[i].gluing[1][0] = 1;
        data->tetrahedron_data[i].gluing[1][1] = 0;
    }
}

```

```

    data->tetrahedron_data[i].gluing[1][2] = 2;
    data->tetrahedron_data[i].gluing[1][3] = 3;

    data->tetrahedron_data[(i+1)%n].gluing[0][0] = 1;
    data->tetrahedron_data[(i+1)%n].gluing[0][1] = 0;
    data->tetrahedron_data[(i+1)%n].gluing[0][2] = 2;
    data->tetrahedron_data[(i+1)%n].gluing[0][3] = 3;

    data->tetrahedron_data[i].gluing[3][0] = 0;
    data->tetrahedron_data[i].gluing[3][1] = 1;
    data->tetrahedron_data[i].gluing[3][2] = 3;
    data->tetrahedron_data[i].gluing[3][3] = 2;

    data->tetrahedron_data[(i+1)%n].gluing[2][0] = 0;
    data->tetrahedron_data[(i+1)%n].gluing[2][1] = 1;
    data->tetrahedron_data[(i+1)%n].gluing[2][2] = 3;
    data->tetrahedron_data[(i+1)%n].gluing[2][3] = 2;
}

for (j = 0; j < 4; j++)
    data->tetrahedron_data[i].cusp_index[j] = 0;

/*
 * Set the peripheral curves to all zeros. This will force
 * data_to_triangulation() to provide default curves.
 * We'll then find linear combinations of the default curves
 * which provide the desired meridian and longitude.
 */
for (j = 0; j < 2; j++)
    for (k = 0; k < 2; k++)
        for (l = 0; l < 4; l++)
            for (m = 0; m < 4; m++)
                data->tetrahedron_data[i].curve[j][k][l][m] = 0;

/*
 * The shape will be ignored.
 */
data->tetrahedron_data[i].filled_shape = Zero;
}

/*
 * If the determinant is to be negative, interchange u[0] and v[0]
 * relative to u[n-1] and v[n-1].
 */
if (anLRFactorization->negative_determinant == TRUE)
{
    for (i = 0; i < 4; i++)
    {
        image[0][i] = data->tetrahedron_data[0].gluing[0][i];
        image[2][i] = data->tetrahedron_data[0].gluing[2][i];
    }

    data->tetrahedron_data[0].gluing[0][1] = image[0][3];
    data->tetrahedron_data[0].gluing[0][3] = image[0][1];

    data->tetrahedron_data[0].gluing[2][1] = image[2][3];
    data->tetrahedron_data[0].gluing[2][3] = image[2][1];

    data->tetrahedron_data[n-1].gluing[image[0][0]][image[0][1]] = 3;
    data->tetrahedron_data[n-1].gluing[image[0][0]][image[0][3]] = 1;

    data->tetrahedron_data[n-1].gluing[image[2][2]][image[2][1]] = 3;
    data->tetrahedron_data[n-1].gluing[image[2][2]][image[2][3]] = 1;
}

/*
 * If the trace is to be negative, negate u[0] and v[0]
 * relative to u[n-1] and v[n-1].
 */
if (anLRFactorization->negative_trace == TRUE)
{
    for (i = 0; i < 4; i++)
    {
        image[0][i] = data->tetrahedron_data[0].gluing[0][i];

```

```

        image[2][i] = data->tetrahedron_data[0].gluing[2][i];
    }

    data->tetrahedron_data[0].gluing[0][0] = image[2][2];
    data->tetrahedron_data[0].gluing[0][1] = image[2][3];
    data->tetrahedron_data[0].gluing[0][2] = image[2][0];
    data->tetrahedron_data[0].gluing[0][3] = image[2][1];

    data->tetrahedron_data[0].gluing[2][0] = image[0][2];
    data->tetrahedron_data[0].gluing[2][1] = image[0][3];
    data->tetrahedron_data[0].gluing[2][2] = image[0][0];
    data->tetrahedron_data[0].gluing[2][3] = image[0][1];

    data->tetrahedron_data[n-1].gluing[image[0][0]][image[0][0]] = 2;
    data->tetrahedron_data[n-1].gluing[image[0][0]][image[0][1]] = 3;
    data->tetrahedron_data[n-1].gluing[image[0][0]][image[0][2]] = 0;
    data->tetrahedron_data[n-1].gluing[image[0][0]][image[0][3]] = 1;

    data->tetrahedron_data[n-1].gluing[image[2][2]][image[2][0]] = 2;
    data->tetrahedron_data[n-1].gluing[image[2][2]][image[2][1]] = 3;
    data->tetrahedron_data[n-1].gluing[image[2][2]][image[2][2]] = 0;
    data->tetrahedron_data[n-1].gluing[image[2][2]][image[2][3]] = 1;
}

/*
 * Create the Triangulation.
 */
data_to_triangulation(data, &manifold);

/*
 * We no longer need the data.
 */
free_triangulation_data(data);

/*
 * We want the meridian to be homotopic to the boundary of the
 * punctured torus fiber, and the longitude to be any curve
 * transverse to it. (Yes, I know that the boundary of the fiber
 * is usually called the longitude, but that conflicts with SnapPea's
 * convention for curves on Klein bottles, which says that the
 * meridian is a nonseparating orientation preserving simple closed
 * curve, and the longitude is an orientation reversing simple
 * closed curve.)
 *
 * Consider any ideal tetrahedron in the triangulation we just
 * created. The triangular cross sections of its four ideal vertices
 * sit like this in the cusp.
 *
 *
 *      0 _____ 2
 *     / \         / \
 *    /   \       /   \
 *   /_____ \ /_____ \
 *  /           \ /           \
 * /             \|             \
 * \             /|             /
 *  \           / \           /
 *   \   3  / \   1  / \
 *    \_____/   \_____/
 *
 * The horizontal line running across the picture is the desired
 * meridian. We'll compute the intersection numbers of the desired
 * meridian with the default meridian and longitude which
 * data_to_triangulation() provided, and use the information to
 * express the desired peripheral curves as linear combinations
 * of the default ones.
 *
 * Note: The desired meridian is completely well defined, but
 * the desired longitude is defined only up the addition of some
 * multiple of the meridian. In other words, the longitude might
 * twist around the cusp.
 */

/*
 * Let tet be any Tetrahedron in the Triangulation.
 */
tet = manifold->tet_list_begin.next;

/*
 * We expect the indexing of the Tetrahedra to be the same as

```

```

* the indexing of the data we provided, i.e.
*
*           3-----2
*           | \   _/ |
*           |  _/   \|
*           | /    \ |
*           0-----1
*
* but we don't want to rely on this assumption. To verify it,
* we check that the edge from vertex 0 to 1 is identified in the
* manifold to the edge from vertex 2 to vertex 3, and similarly
* that the edge from vertex 1 to vertex 2 is identified to the
* edge from vertex 3 to vertex 0. We do, however, assume that
* the Triangulation is combinatorially equivalent to the one
* specified in the data.
*/
if
(
    tet->edge_class[edge_between_vertices[0][1]]
    != tet->edge_class[edge_between_vertices[2][3]]
||
    tet->edge_class[edge_between_vertices[1][2]]
    != tet->edge_class[edge_between_vertices[3][0]]
)
{
    /*
    * I don't think this situation will ever arise,
    * so I haven't written any code to handle it.
    * If necessary, it would be very easy to write code
    * to discover the indexing scheme on the tet.
    */
    uFatalError("triangulate_punctured_torus_bundle", "punctured_torus_bundle");
}

/*
* Count the number of times the default meridian and longitude
* pass upwards through the intended meridian. Use only
* the right_handed sheet -- this will give correct results on
* both a torus cusp and a Klein bottle cusp.
*/
for (c = 0; c < 2; c++) /* c = M, L */
    signed_intersections[c] = tet->curve[c][right_handed][0][2]
                             - tet->curve[c][right_handed][1][3]
                             + tet->curve[c][right_handed][2][0]
                             - tet->curve[c][right_handed][3][1];

/*
* The desired meridian and longitude will be linear combinations
* of the default meridian and longitude
*
* desired meridian = m00*(default meridian) + m01*(default longitude)
* desired longitude = m10*(default meridian) + m11*(default longitude)
*
* The desired meridian has intersection number zero with itself, so
*
*     m00*signed_intersections[M] + m01*signed_intersections[L] = 0
*
* The desired longitude has intersection number +1 with the
* desired meridian, so
*
*     m10*signed_intersections[M] + m11*signed_intersections[L] = +1
*/

/*
* Try
*
*     m00 = signed_intersections[L]
*     m01 = - signed_intersections[M]
*
* Later we may have to negate these coefficients if the direction
* is wrong.
*/
change_matrices[0][0][0] = signed_intersections[L];
change_matrices[0][0][1] = - signed_intersections[M];

```

```
/*
 * Use the Euclidean algorithm to solve for M10 and M11.
 */
if (euclidean_algorithm(signed_intersections[M],
                        signed_intersections[L],
                        &m10,
                        &m11) != 1)
    uFatalError("triangulate_punctured_torus_bundle", "punctured_torus_bundle");
change_matrices[0][1][0] = m10;
change_matrices[0][1][1] = m11;

/*
 * If change_matrices[0] has determinant -1, reverse the meridian.
 */
switch (DET2(change_matrices[0]))
{
    case +1:
        /*
         * The meridian is correct.
         */
        break;

    case -1:
        /*
         * Reverse the meridian.
         */
        change_matrices[0][0][0] = - change_matrices[0][0][0];
        change_matrices[0][0][1] = - change_matrices[0][0][1];
        break;

    default:
        uFatalError("triangulate_punctured_torus_bundle", "punctured_torus_bundle");
}

/*
 * Change the peripheral curves.
 */
change_peripheral_curves(manifold, change_matrices);

/*
 * Done!
 */
return manifold;
}
```