

# **Oscillating curves on triangulated 3-manifolds**

An algorithm and implementation for constructing oscillating curves

**Joshua Childs**

**Supervisor: Daniel Mathews**

A thesis presented for the degree of  
Bachelor of Science, Advanced Research (Honours)

School of Mathematics  
Monash University  
Australia

## Abstract

In 2021, Jessica Purcell and Daniel Mathews introduced a geometric interpretation of the vectors dual to the gluing vectors [MP22]. These extend the Neumann-Zagier matrix to one which is a symplectic matrix (up to factor 2).

In 1985, as part of his PhD dissertation [Wee85], Jeffrey Weeks developed SnapPea. This is a collection of algorithms written in C for computing with hyperbolic 3-manifolds [Wee03]. This is currently maintained as the SnapPy [Cul+23] python package on GitHub. We develop and implement an algorithm for constructing oscillating curves dual to the gluing curves and extending the Neumann-Zagier matrix to one which is symplectic (up to factors of 2) in C using SnapPy.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Knot Theory Preliminaries</b>	<b>4</b>
2.1	Tetrahedral Decomposition . . . . .	5
2.2	Hyperbolic Structures . . . . .	7
2.3	Symplectic Form . . . . .	9
<b>3</b>	<b>Lambda Lengths</b>	<b>10</b>
3.1	Lambda Lengths in Hyperbolic 2-space . . . . .	10
3.2	Hyperbolic 3-Space Preliminaries . . . . .	12
3.3	Lambda Lengths in Hyperbolic 3-Space . . . . .	13
<b>4</b>	<b>A Symplectic Basis for 3-Manifold Triangulations</b>	<b>15</b>
4.1	Triangulation . . . . .	16
4.2	Train Tracks . . . . .	16
4.3	End Multi Graph . . . . .	16
4.4	Cusp Triangulation . . . . .	17
4.5	Oscillating Curves . . . . .	17
4.6	Neumann-Zagier Matrix . . . . .	18
<b>5</b>	<b>Algorithm</b>	<b>19</b>
5.1	Cusp Triangles . . . . .	19
5.2	Cusp Regions . . . . .	20
5.3	Adjacent Cusp Regions . . . . .	23
5.4	Cusp Region Graph . . . . .	25
5.5	End Multi Graph . . . . .	26
5.6	Train Lines . . . . .	26
5.7	Oscillating Curves . . . . .	33
5.8	Cusp Region Splitting . . . . .	34
5.9	Neumann-Zagier Matrix . . . . .	36
<b>6</b>	<b>Implementation Details</b>	<b>37</b>
6.1	Cusp Triangulation . . . . .	39
6.2	Cusp Region . . . . .	40
6.3	Cusp Region Graph . . . . .	43
6.4	End Multi Graph . . . . .	44
6.5	Path Nodes and End Points . . . . .	45
6.6	Cusp Structure . . . . .	46
6.7	Train Lines . . . . .	46
6.8	Oscillating Curves . . . . .	47
6.9	Neumann-Zagier Matrix . . . . .	47
<b>7</b>	<b>Examples</b>	<b>48</b>
7.1	Figure-8 Knot . . . . .	48
7.2	L6a4 Link . . . . .	53
<b>8</b>	<b>Conclusion</b>	<b>61</b>
<b>9</b>	<b>Bibliography</b>	<b>61</b>

# 1 Introduction

A 3-manifold is a topological space which locally looks like the familiar 3-dimensional Euclidean space  $\mathbb{R}^3$ . A central topic to the study of 3-manifolds is the geometric structures which can be put on manifold. Thurston's geometrisation conjecture states that certain 3-manifolds each have unique geometric structures which can be associated with them. Thurston proposed this conjecture in 1982 [Thu82]. The analogue for 2-dimensional manifolds states that certain 2-manifolds fall into either having spherical, Euclidean or hyperbolic geometry. For 3-manifolds, Thurston decomposes each manifold in a natural way into 8 types of geometric structure. A key tool for working with 3-manifolds is triangulations, in particular ideal triangulations.

An ideal tetrahedron is a tetrahedron with each vertex removed. An ideal triangulation of a 3-manifold  $M$  is a gluing of ideal tetrahedra which results in a manifold homeomorphic to  $M$ . The removed vertices give triangles at each vertex, called cusp triangles. The cusp triangles correspond to the boundary of  $M$ , and the gluing on the faces of the tetrahedra induce a gluing on the edges of each triangle. This results in a triangulation of the boundary of  $M$ . On each boundary component, we have a collection of curves called edge curves. These curves give rise to vectors which are the coefficients of the gluing equations. Thurston showed a triangulation admits a hyperbolic structure iff the tetrahedra satisfy the gluing equations [Thu80]. We also a collection of simple closed curves which form a basis for the fundamental group of the boundary component. These curves give rise to vectors which are the coefficients of the cusp (or completeness) equations. Thurston also showed that a triangulation which admits a hyperbolic structure, admits a complete hyperbolic structure iff the tetrahedra satisfy the completeness equations [Thu80].

The study of 3-manifolds is closely related to Knot Theory. A knot is a smooth embedding of the circle in the 3-sphere, and a link is a smooth embedding of a disjoint union of circles into the 3-sphere. The complement of a knot or link in the 3-sphere is a 3-manifold. Thurston showed that if a knot is *not* a satellite or torus knot, its complement admits a hyperbolic structure [Thu80]. Thurston also showed that knot complements which admit a hyperbolic structure admit a complete hyperbolic structure [Thu80]. In this case the complete hyperbolic structure is unique, a result known as Mostow-Prasad rigidity [Mos86; Pra73]. Thurston showed that near the complete, hyperbolic structure there is a "neighbourhood" of hyperbolic structures which are not complete except the unique complete one [Thu80]. In the 1980s, Neumann and Zagier were investigating how the volume of the manifold changes in this neighbourhood [NZ85]. They developed a symplectic form, which arises from properties of the triangulation. The Neumann and Zagier also formed a matrix with rows coming from the coefficients of the gluing and completeness equations. They extended this matrix to one which is symplectic up to factors of 2, using the Gram-Schmidt algorithm. In 2022, Mathews and Purcell developed oscillating curves which give rise to vectors that extend the Neumann-Zagier matrix to one which is symplectic up to factors of 2 [MP22]. This symplectic matrix forms an equation which gives integer solutions to the gluing and cusp equations, modulo a factor of 2. This equation was considered by Neumann, who showed integer solutions exist [Neu92]. In 2021, Howie, Mathews and Purcell showed integer solutions can be obtained by adding multiples of explicit vectors [HMP21].

SnapPy [Cul+23] is a python package written in C which can perform various calculations with 3-manifolds. SnapPy can triangulate knot and link complements drawn by the user using the Plink library and contains a large database of 3-manifolds. The aim of this thesis is to design and implement an algorithm in C to construct oscillating curves dual to the edge curves on knot and link complements, and the associated symplectic matrix. We begin by introducing some preliminaries of Knot Theory in Section 2. Then we discuss the notion of Lambda Lengths in Section 3. In Section 4, we give a brief summary of oscillating curves and their properties as described in [MP22]. Section 5 covers our approach to constructing oscillating curves explicitly, which is new to this paper. Section 6 goes into the technical details of representing the objects defined in the process of constructing oscillating curves. Finally, in Section 7 we give an example of the algorithm on the Figure 8 knot and L6a4 Link.

## Acknowledgments

I would like to thank my supervisor, Daniel, for his generous support and time throughout the year. Without your support, none of this would have been possible. Many thanks to my family, friends, and fellow honours students for their support.

## 2 Knot Theory Preliminaries

We begin by outlining the basic definitions of knot theory and tetrahedral decomposition. This exposition and the more technical details can be found in Hyperbolic Knot Theory [Pur20] which is based on the work of Thurston [Thu80].

**Definition 2.1** (Manifold). *Let  $M$  be a hausdorff, second countable topological space.  $M$  is an  $n$ -dimensional manifold if there exists a collection*

$$\{(U_\alpha, \varphi_\alpha) \mid \alpha \in J, U_\alpha \subset M \text{ open}, \varphi_\alpha : M \rightarrow \mathbb{R}^n\},$$

*such that  $\{U_\alpha \mid \alpha \in J\}$  is an open cover for  $M$  and,*

$$\varphi_\beta \circ \varphi_\alpha^{-1} : \varphi_\alpha(U_\alpha \cap U_\beta) \rightarrow \varphi_\beta(U_\alpha \cap U_\beta)$$

*is a homeomorphism.*

We define Euclidean half space by

$$\mathbb{R}_+^n = \{(x_1, \dots, x_n) \in \mathbb{R}^n \mid x^n \geq 0\}.$$

A manifold with boundary is defined similarly to manifolds, where instead locally a manifold with boundary looks like Euclidean half space.

**Definition 2.2** (Manifold with boundary). *Let  $M$  be a hausdorff, second countable topological space.  $M$  is an  $n$ -dimensional manifold with boundary if there exists a collection*

$$\{(U_\alpha, \varphi_\alpha) \mid \alpha \in J, U_\alpha \subset M \text{ open}, \varphi_\alpha : M \rightarrow \mathbb{R}_+^n\},$$

*such that  $\{U_\alpha \mid \alpha \in J\}$  is an open cover for  $M$  and,*

$$\varphi_\beta \circ \varphi_\alpha^{-1} : \varphi_\alpha(U_\alpha \cap U_\beta) \rightarrow \varphi_\beta(U_\alpha \cap U_\beta)$$

*is a homeomorphism.*

We have the boundary of  $\mathbb{R}_+^n$ , denoted  $\partial\mathbb{R}_+^n$ , is given by

$$\partial\mathbb{R}_+^n = \{(x^1, \dots, x^n) \in \mathbb{R}^n \mid x^n = 0\}.$$

The boundary of  $M$ , denoted  $\partial M$ , is defined as

$$\partial M = \bigcup_{\alpha \in J} \varphi_\alpha^{-1}(\partial\mathbb{R}_+^n \cap \varphi_\alpha(U_\alpha)).$$

**Definition 2.3** (Knots and Links). *A knot  $K \subset S^3$  is a subset of points homeomorphic to a circle  $S^1$  under a piecewise linear (PL) homeomorphism.*

*A link is a subset of  $S^3$  PL homeomorphic to a disjoint union of copies of  $S^1$ .*

$S^3$  is the unit sphere in  $\mathbb{R}^4$  consisting of the points  $(x, y, z, w)$  such that  $x^2 + y^2 + z^2 + w^2 = 1$ .

**Definition 2.4** (Knot Complement). *The knot complement of a knot  $K \subset S^3$  is  $S^3 - K$ .*

Notice the knot complement is a 3-manifold with boundary consisting of the knot. We are interested in triangulated 3-manifolds, since they give a concrete model for computation.

## 2.1 Tetrahedral Decomposition

**Definition 2.5.** A *polyhedron* is a closed 3-ball whose boundary is labelled with a finite graph, containing a finite number of vertices and edges, so that complementary regions, which are called *faces*, are simply connected.

An *ideal polyhedron* is a polyhedron with all vertices removed.

**Example 2.6** (Figure-8 Knot). We construct an ideal triangulation of the figure 8 knot complement using polyhedral decomposition. The full details for this construction can be found in [Pur20]. Starting with  $S^3 - K$ , we split it into two polyhedra.

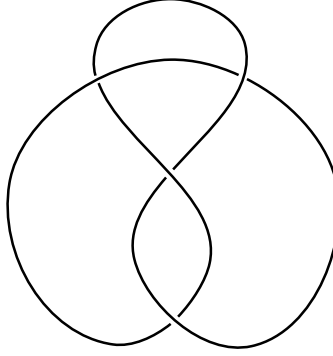


Figure 1: Figure 8 Knot

The polyhedral decomposition of the figure 8 knot contains two polyhedra. We imagine one polyhedron lying above the knot and another below. Then we expand the polyhedra, they will meet at the regions cut out by the knot, labelled below  $A, B, C, D, E$  and  $F$ .

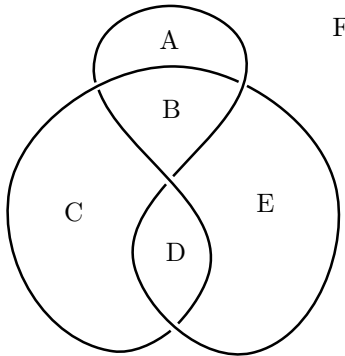


Figure 2: Faces of the figure 8 knot complement

The polyhedra will be represented as a 4-valent graph in the plane. The faces of the polyhedra meet at edges of the knot, and there is one edge in the graph for each crossing of the knot. Edges come from arcs that connect two strands of the diagram at a crossing. These are called *crossing arcs*. Initially we draw 4 edges for each crossing, with the edge going from the overcrossing to undercrossing or vice versa. This will make it easier to visualize which edges will bound the faces. Orientations on the edges can be chosen to run in either direction, as long as we are consistent with the orientations corresponding to the same edge. A choice of orientations is sketched below,

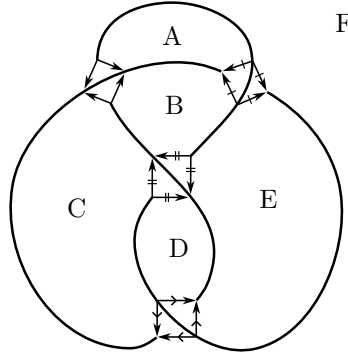


Figure 3: Edges of the figure 8 knot complement

Shrink the knot along ideal vertices on the top polyhedron. Notice the edges of the polyhedron have endpoints on the knot. Since the endpoints are contained in the knot, they are *not* in the knot complement  $S^3 - K$ . Hence, we need ideal polyhedra, which have the vertices removed. Similarly, since the knot is *not* in the manifold, we can shrink along the strands of the knot. We start with the top polyhedron. Focus on a neighbourhood of a crossing. As we shrink the knot between the 4 arrows that go from the overcrossing to the undercrossing or vice versa, the pairs of edges will merge into one edge. This leaves us with two edges running from the overcrossing to the undercrossing or vice versa (Figure 4). Now shrink the knot along the strands between crossings. This collapses the pair of edges into one edge running from one crossing to another. Now we are left with a graph consisting of the edges of the top polyhedron (Figure 5).

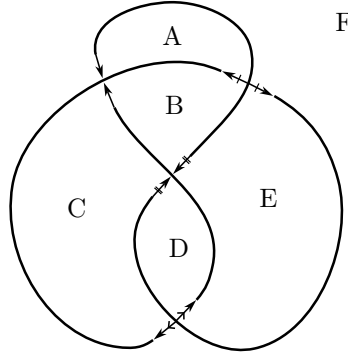


Figure 4: Isotopic edges in the top polyhedron

Following the same method as with the top polyhedron, shrink the knot along ideal vertices on the bottom polyhedron. Since the bottom polyhedron is beneath the knot, we need to flip each crossing of the knot, to view the knot from the inside of the polyhedron. When we flip the knot each overcrossing becomes an undercrossing and vice versa. This means we need to also flip the edge directions, in this case from the overcrossing to the undercrossing. This gives a polyhedral decomposition of the figure 8 knot.

**Definition 2.7** (Ideal Triangulation). *Let  $M$  be a 3-manifold. A topological ideal triangulation of  $M$  is a combinatorial way of gluing ideal tetrahedra (tetrahedra with each vertex removed) so that the result is homeomorphic to  $M$ . The gluing should take edges to edges, and faces to faces.*

We obtain a tetrahedral decomposition from the polyhedral decomposition as follows. Choose a polyhedron and an ideal vertex  $v$  and add an edge between  $v$  and all other vertices on the polyhedron. Between any two edges meeting  $v$  add an ideal triangle meeting  $v$ . Between any three triangles meeting  $v$  add an ideal tetrahedron.

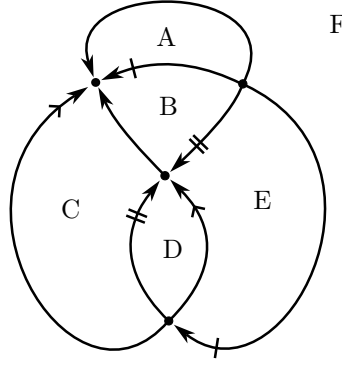


Figure 5: Graph of the top polyhedron

Split off the resulting tetrahedron. This reduces the collection of polyhedra to a collection with one fewer vertex. After repeating a finite number of times, we obtain a collection of ideal tetrahedra which glue to give the original manifold.

In general, 3-manifolds admit ideal triangulations, a result due to the work of Moise and Bing [Bin59; Moi52].

**Definition 2.8** (Cusp Triangulation). *Let  $M$  have topological ideal triangulation. When we remove the vertices, we obtain a collection of triangles lying on the boundary of  $M$ . The gluing on the faces of the tetrahedra induces a gluing of these triangles, called the cusp triangulation.*

## 2.2 Hyperbolic Structures

**Theorem 2.9.** *Let  $M$  be a manifold with boundary consisting of tori, then for any topological ideal triangulation of  $M$ , the number of edges of the triangulation is the number of tetrahedra.*

There is a natural equivalence relation on the edges of the tetrahedra, where two edges are equivalent iff they are identified in the gluing of the tetrahedra. Theorem 2.9 tells us that there are  $n$  equivalence classes, where  $n$  is the number of tetrahedra in the triangulation.

**Definition 2.10** (Edge Class). *Let  $M$  be a triangulated 3-manifold with a triangulation into  $n$  ideal tetrahedra. Index the equivalence classes of edges with an integer  $1 \leq i \leq n$ . The edge class of an edge  $E$  is the index of  $[E]$ .*

Each edge of a tetrahedron has two ends, and each of those ends lies at a vertex of the cusp triangulation. In this way, we have a labelling of each cusp vertex by the edge class which lies at the vertex. When an edge has ends lying on the same cusp, which is the case for all edges in a knot complement, the edge class appears as two distinct vertices in the cusp triangulation of the cusp. To distinguish these vertices we give each cusp vertex an edge index.

**Definition 2.11** (Edge Index). *Let  $M$  be a triangulated 3-manifold. Let  $v$  be a vertex in the cusp triangulation of a boundary component of  $M$ . The edge index of  $v$  is an integer, either 0 or 1, such that two distinct cusp vertices corresponding to the same edge class have different edge indices.*

### 2.2.1 Geometric Structures

**Definition 2.12** (Hyperbolic 3-space). *The upper half plane model of Hyperbolic 3-space is given by,*

$$\mathbb{H}^3 = \{(x, y, z) \in \mathbb{R}^3 \mid z > 0\}$$

*with the metric,*



$$ds^2 = \frac{dx^2 + dy^2 + dz^2}{z}.$$

Let  $M$  be a manifold with an ideal triangulation and a Riemannian metric  $g$ . Then  $M$  admits a hyperbolic structure if every point of the manifold has a neighbourhood isometric to a ball in  $\mathbb{H}^3$ . We would like to know when an ideal triangulation admits a hyperbolic structure. Suppose a 3-manifold  $M$  has an ideal triangulation consisting on  $n$  tetrahedra. We label opposite edges of the  $i$ -th tetrahedron by  $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i$ . This induces a labelling of the vertices of a cusp triangle by the label on the edge the vertex lies in.

**Definition 2.13** (Combinatorial Holonomy). *Let  $\gamma$  be a curve in a cusp triangle which meets distinct side of the triangle. The combinatorial holonomy,  $h(\gamma)$  is edge label,  $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i$  of the cusp triangle which is cut off by the curve. The sign of  $h(\gamma)$  is  $+1$  for a curve travelling anti-clockwise and  $-1$  for a curve travelling clockwise.*

See Figure 6 for an example. We extend this holonomy to an oriented curve on a cusp, as a formal sum of the holonomy for each arc of the curve. Let  $M$  be a manifold with tori boundary and ideal triangulation which admits a hyperbolic structure. Each edge of the triangulation has two ends, each of which lie in a cusp of  $M$ . In the cusp triangulation, this edge corresponds to a vertex of a cusp triangle. The combinatorial holonomy of a curve which encircles this vertex, gives the coefficients of the gluing equations, which we describe below.

Consider an ideal tetrahedron embedded in  $\mathbb{H}^3$ . Choose an edge  $e$  of the tetrahedron, then we can map the vertices of  $e$  to 0 and  $\infty$ . If we send one of the remaining vertices of the tetrahedron to 1, this uniquely determines where the final vertex lies. This final vertex lies on the boundary of  $\mathbb{H}^3$  and not at infinity, so we can consider it as an element of  $\mathbb{C}$ , but we have two different points depending on which vertex we send to 1. One of these points will have positive imaginary part, and the other negative imaginary part, so as a convention we choose the one with positive imaginary part. This is called the edge invariant of  $e$ , denoted  $z(e)$ .

**Definition 2.14** (Edge Invariant). *For an ideal tetrahedron  $T$  embedded in  $\mathbb{H}^3$  and edge  $e$  of that tetrahedron, define the number  $z(e)$  in  $\mathbb{C}$  to be the complex number with positive imaginary part obtained by applying the unique isometry of  $\mathbb{H}^3$  that takes the vertices of  $e$  to 0 and  $\infty$ , takes another vertex to 1 and takes the final vertex of  $T$  to  $z(e)$ . This is called the edge invariant.*

For a hyperbolic structure, we want the tetrahedra around an edge to glue such that walking around the edge brings us back to the same position. This is reflected in the condition, of the following theorem, that the product of edge invariants is 1. The following theorem is due to Thurston, and a proof can be found in [Pur20].

**Theorem 2.15** (Edge Gluing Equations). *Let  $M$  be a 3-manifold which admits a topological ideal triangulation such that each ideal tetrahedron has a hyperbolic structure. The hyperbolic structures on the ideal tetrahedra induce a hyperbolic structure on the gluing  $M$  iff for each edge  $e$ ,*

$$\prod z(e_i) = 1 \quad \text{and} \quad \sum \arg(z(e_i)) = 2\pi.$$

## 2.2.2 Complete Structures

A manifold has a complete structure if it admits a complete metric. Similarly to geometric structures, we would like to know when a triangulation admits a complete structure. Consider a tori boundary component of a manifold  $M$ , and a basis for the fundamental group of this torus  $\mathbf{m}, \mathbf{l}$ . For manifolds which admit a complete hyperbolic structure, the holonomy of these curves gives the coefficients of the completeness equations,  $H(\mathbf{m}) = 1$  and  $H(\mathbf{l}) = 1$ .

**Definition 2.16.** *Suppose  $M$  has a topological ideal triangulation, and let  $T$  be the boundary torus of a cusp of  $M$ . Let  $[\alpha] \in \pi_1(T)$ , so  $\alpha$  is a loop on  $T$  in the homotopy class of  $[\alpha]$ . We associate a complex number*

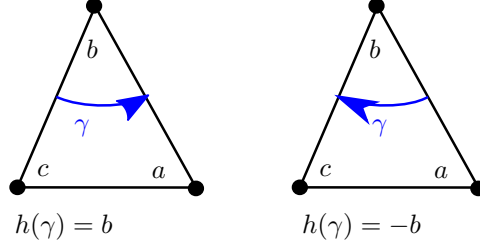


Figure 6: Combinatorial Holonomy [MP22].

$H(\alpha)$  to  $\alpha$  as follows. The loop  $\alpha$  can be homotoped to run through any triangle of the cusp triangulation of  $T$  monotonically, i.e. in such a way that it cuts off a single corner of each triangle it enters. Denote the edge invariants of the corners cut off by  $\alpha$  with  $z_1, \dots, z_n$ . Further associate to each corner a value  $\epsilon_i = \pm 1$ : if the  $i$ -th corner cut off by  $\alpha$  lies to the left of  $\alpha$ , set  $\epsilon_i = +1$ . If the corner lies to the right of  $\alpha$ , set  $\epsilon_i = -1$ . Finally, set the value of  $H(\alpha)$  to be

$$H(\alpha) = \prod_{i=1}^n z_i^{\epsilon_i}.$$

The completeness equations indicate whether a triangulation admits a hyperbolic structure. The following proposition is due to Thurston, a proof can be found in [Pur20].

**Proposition 2.17** (Completeness Equations). *Let  $T$  be the torus boundary of a cusp neighborhood of  $M$ , where  $M$  admits a topological ideal triangulation, and the ideal tetrahedra admit hyperbolic structures that satisfy the edge gluing equations (Theorem 2.15). Let  $\mathfrak{m}, \mathfrak{l}$  be a basis for  $\pi_1(T)$ . If  $H(\mathfrak{m}) = H(\mathfrak{l}) = 1$  on each cusp of  $M$ , then the hyperbolic structure on  $M$  induced by the hyperbolic structure on the tetrahedra will be a complete structure.*

## 2.3 Symplectic Form

We define an abstract intersection number of two curves on a cusp triangulation which will be related to the Neumann-Zagier symplectic form. The intersection number is defined locally on a cusp triangle and extended linearly to the cusp triangulation.

**Definition 2.18** (Local Signed Intersection Number). *Let  $\delta$  be a cusp triangle and  $\gamma, \gamma'$  two curves on  $\delta$  which run between distinct sides of the cusp triangle. If  $\gamma$  and  $\gamma'$  are oriented such that they enter  $\delta$  on the same edge, the local signed intersection number is defined as*

$$\gamma \cdot \gamma' = 1, \quad \gamma' \cdot \gamma = -1$$

*with all other local intersection numbers being 0.*

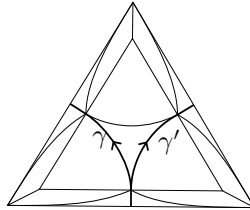


Figure 7: Two curves  $\gamma$  and  $\gamma'$ , which pick up a holonomy as they separate.

This local intersection number is extended linearly to an abstract intersection number on the cusp. We can also consider a curve  $\gamma$  on a cusp triangulation as a vector in  $\mathbb{R}^{3n}$ , given by  $h(\gamma)$ . Consider the vector space  $V$

generated by  $\{a_i, b_i, c_i \mid 1 \leq i \leq n\}$ , consisting of formal sums with coefficients in  $\mathbb{R}$ . Then  $V$  has dimension  $3n$ , and is naturally identified with  $\mathbb{R}^{3n}$ . Neumann and Zagier imposed the relations  $a_i + b_i + c_i = 0$ , which gives a quotient space  $V/\sim$  of dimension  $2n$  spanned by  $\{a_i - c_i, b_i - c_i \mid 1 \leq i \leq n\}$ . Applying the quotient map to  $h(\gamma)$ , we obtain an element of  $V/\sim$ . We define the symplectic form on this quotient space.

**Definition 2.19** (Symplectic Form). *Let  $x = (x_1, \dots, x_{2n}), y = (y_1, \dots, y_{2n}) \in \mathbb{R}^{2n}$ , and define  $\omega : \mathbb{R}^{2n} \times \mathbb{R}^{2n} \rightarrow \mathbb{R}$  by*

$$\omega(x, y) = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \dots + \begin{vmatrix} x_{2n-1} & x_{2n} \\ y_{2n-1} & y_{2n} \end{vmatrix}$$

We will see later that the intersection number of particular curves on a cusp is related to the symplectic form applied to the holonomy of the curves.

### 3 Lambda Lengths

Before introducing oscillating curves on triangulated 3-manifolds, we take a brief detour to describe a related concept known as lambda lengths. In hyperbolic 2-space, lambda lengths satisfy a so called 'Ptolemy Equation', and the following exposition is based on the exposition given by Penner [Pen10]. Lambda lengths and the Ptolemy equation can be generalised to complex lambda lengths in hyperbolic 3-space which satisfy a corresponding Ptolemy equation, for the proofs we refer to Spinors and Horospheres [Mat23].

#### 3.1 Lambda Lengths in Hyperbolic 2-space

We begin by introducing some basic hyperbolic geometry.

**Definition 3.1** (Projective Special Linear Group). *The Mobius group  $\mathrm{PSL}(2, \mathbb{R})$  is the group of  $2 \times 2$  matrices of determinant 1 modulo the relation which identifies  $I$  with  $-I$ .*

$$\mathrm{PSL}(2, \mathbb{R}) = \mathrm{SL}(2, \mathbb{R}) / (I \sim -I).$$

We have three different models of hyperbolic 2-space, known as the conformal disk, upper half plane and the open positive light cone  $L^+$ .

**Definition 3.2** (Conformal Disk Model). *The conformal disk model is a 2-dimensional hyperbolic manifold given by,*

$$\mathbb{D} = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 < 1\}.$$

*together with the following metric which has constant gaussian curvature of  $-1$ .*

$$ds^2 = 4 \frac{dx^2 + dy^2}{1 - \|z\|^2}.$$

Geodesics in the disk are precisely the curves which lie on circles which intersect the boundary of  $\mathbb{D}^2$  orthogonally. We can describe these circles explicitly. Let  $v = v_0 + iv_1, w = w_0 + iw_1 \in \mathbb{D}^2$ . The geodesic passing through  $v$  and  $w$  is given by  $x^2 + y^2 + ax + by + 1 = 0$  with  $(x, y) \in \mathbb{D}^2$  and,

$$a = \frac{v_1(1 + |w|^2) - w_1(1 + |v|^2)}{v_0w_1 - v_1w_0},$$

$$b = \frac{w_0(1 + |v|^2) - v_0(1 + |w|^2)}{v_0w_1 - v_1w_0}.$$

**Definition 3.3** (Upper Half Plane). *The Upper Half Plane is the set*

$$\mathcal{U} = \{(x, y) \in \mathbb{R}^2 \mid y > 0\},$$

together with the metric,

$$ds^2 = \frac{dx^2 + dy^2}{y}.$$

$\mathrm{PSL}(2, \mathbb{R})$  acts on  $\mathcal{U}$  by fractional linear transformations,

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} : z \mapsto \frac{az + b}{cz + d}.$$

Similarly to the Poincare disk, the geodesics in the upper half plane are precisely the curves which lie on circles that intersect the boundary orthogonally. Since the boundary of the upper half plane consists of the x-axis and the point at infinity, we have circles which intersect the x-axis orthogonally or vertical lines. Let  $v = v_0 + iv_1, w = w_0 + iw_1 \in \mathcal{U}$ , where  $v_0 \neq w_0$ . The geodesic passing through  $v$  and  $w$  is given by  $(x - a)^2 + y^2 = r^2$  with  $(x, y) \in \mathcal{U}$  and,

$$a = \frac{|v|^2 - |w|^2}{2(v_0 + w_0)},$$

$$r^2 = (v_0 - a)^2 + v_1^2.$$

A horocycle in  $\mathcal{U}$  is a Euclidean circle tangent to the real line at some point  $p$ , or a horizontal line for a horocycle at infinity. This definition relies on the Euclidean metric, so we have an alternate invariant definition of a horocycle.

**Definition 3.4** (Horocycle). *Choose a point  $p$  in the hyperbolic plane and a tangent direction  $v$  at  $p$ , and consider a family of hyperbolic circles whose radius and center diverge in such a controlled manner as to pass through  $p$  with tangent direction  $v$ . Such a sequence of hyperbolic circles has a well defined limit, defined to be a horocycle.*

**Definition 3.5** (Minkowski 3-Space). *Minkowski 3-space is  $\mathbb{R}^3$  together with the indefinite pairing,*

$$\langle \cdot, \cdot \rangle : \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}, \quad \langle (x, y, z), (x', y', z') \rangle \mapsto xx' + yy' - zz'.$$

There are several characteristic subspaces of Minkowski 3-space, in particular,

$$\mathbb{H} = \{u = (x, y, z) \in \mathbb{R}^3 \mid \langle u, u \rangle = -1 \text{ and } z > 0\},$$

which is the upper sheet of the hyperboloid and is also a model for hyperbolic geometry,

$$L^+ = \{u = (x, y, z) \in \mathbb{R}^3 \mid \langle u, u \rangle = 0 \text{ and } z > 0\},$$

which is the open positive light-cone, and finally,

$$H = \{u = (x, y, z) \in \mathbb{R}^3 \mid \langle u, u \rangle = +1\}.$$

which is the hyperboloid of one sheet. An isometry between  $\mathbb{H}$  and  $\mathbb{D}$  is given by the projection  $\bar{\cdot}$  from the point  $(0, 0, -1)$ .

$$\bar{\cdot} : \mathbb{H} \rightarrow \mathbb{D}, \quad (x, y, z) \mapsto \frac{1}{1+z}(x, y).$$

**Lemma 3.6** (Horocycles [Pen10]). *The assignment,*

$$L^+ \rightarrow \{\text{horocycles in } \mathbb{H}\}, \quad u \mapsto h_u = \{v \in \mathbb{H} \mid \langle u, v \rangle = \frac{1}{\sqrt{2}}\},$$

*establishes an isomorphism between points of  $L^+$  and the collection of all horocycles in  $\mathbb{H}$ . Furthermore, the center of the corresponding horocycle  $\bar{h}_u$  in  $\mathbb{D}$  is  $\bar{u} \in S_\infty^1$ , and the Euclidean radius of  $\bar{h}_u$  in  $\mathbb{D}$  is  $\frac{1}{1+z\sqrt{2}}$ , where  $z = (x, y, z)$ .*

**Definition 3.7** (Lambda Lengths). *Given a pair of horocycles  $h_1, h_2$ , in the Poincare Disk, with distinct centers, consider the geodesic  $\gamma$  in  $\mathbb{D}$  connecting their centers. Let  $\delta$  denote the signed hyperbolic distance between  $h_1 \cap \gamma$  and  $h_2 \cap \gamma$ , where the sign of  $\delta$  is taken to be positive iff  $h_1$  and  $h_2$  are disjoint. This is the length of the curve  $\gamma$  between  $h_1 \cap \gamma$  and  $h_2 \cap \gamma$ . Define the lambda length of  $h_1, h_2$  to be*

$$\lambda(h_1, h_2) = \sqrt{e^\delta}.$$

Figure 8 shows an example of two horospheres  $h_1, h_2$  tangent to the boundary of  $\mathbb{D}$  at  $v$  and  $w$  respectively, and signed distance  $\delta$  between the two horospheres.

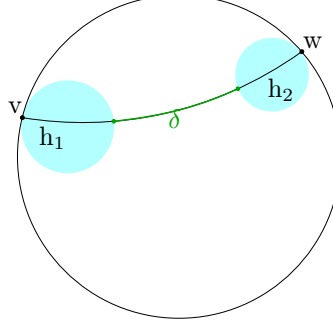


Figure 8: Horospheres and Lambda Lengths in the Poincare Disk.  $h_1$  is a horosphere at  $v$  and  $h_2$  is a horosphere at  $w$ , with  $\delta$  the distance between the horospheres along the geodesic through  $v$  and  $w$ .

**Theorem 3.8** (Ptolemy Equation [Pen10]). *Suppose  $u_1, u_2, u_3, u_4 \in L^+$ , where  $\overline{u_1}, \overline{u_2}, \overline{u_3}, \overline{u_4} \in S^1$  are distinct and occur in this counterclockwise cyclic order, and let  $\lambda_{jk} = \lambda(h_{u_j}, h_{u_k})$  for  $j, k \in \{1, 2, 3, 4\}$ , denote the lambda lengths. Then*

$$\lambda_{13}\lambda_{24} = \lambda_{12}\lambda_{34} + \lambda_{14}\lambda_{23}.$$

This construction of lambda lengths and Ptolemy's equation in hyperbolic 2-space generalises to complex lambda lengths and another Ptolemy equation in hyperbolic 3-space.

### 3.2 Hyperbolic 3-Space Preliminaries

Before defining complex lambda lengths in hyperbolic 3-space, we begin with some preliminaries from differential geometry. The set of invertible  $n \times n$  matrices over a field  $F$  is denoted  $\text{GL}(n, F)$ . We are interested in the subgroup of matrices which preserve the orientation of space, known as special orthogonal matrices.

**Definition 3.9** (Special Orthogonal Group). *The special orthogonal group, denoted  $\text{SO}(n)$ , of real  $n \times n$  matrices is given by,*

$$\text{SO}(n) = \{A \in \text{GL}(n, \mathbb{R}) \mid \det A = 1 \text{ and } AA^T = 1\}.$$

Each matrix  $M \in \text{SO}(3)$  represents a transformation of  $\mathbb{R}^3$ . We can identify this transformation with an axis, represented as a point in  $S^2$ , and a rotation around this axis. Hence, each point  $p$  in the ball of radius  $\pi$  in  $\mathbb{R}^3$  defines an element of  $\text{SO}(3)$  obtained by a rotation of  $\|p\|$  anti-clockwise around the axis through  $p$ . This is not a bijection since anti-podal points on the boundary of the ball define the same element of  $\text{SO}(3)$ . Identifying anti-podal points of the ball we obtain  $\mathbb{RP}^3$  which is in bijection with the elements of  $\text{SO}(3)$ . The anti-podal map gives rise to a covering space action of  $\mathbb{Z}_2$  on  $\mathbb{RP}^3$ . Since  $S^3$  is simply connected, we have  $\pi(\mathbb{RP}^3) = \mathbb{Z}_2$  and  $S^3$  is the universal, double cover of  $\mathbb{RP}^3$ . The lift of  $\text{SO}(3)$  to the universal cover is called the spin group  $\text{Spin}(3)$ .

**Definition 3.10** (Fiber Bundle). A fiber bundle is a structure  $(E, M, \pi, F)$  which consists of manifolds  $E, M, F$ , and a surjective map  $\pi : E \rightarrow M$  such that, for any  $x \in M$ , there exists an open neighbourhood  $U$  and a homeomorphism  $\varphi : \pi^{-1}(U) \rightarrow U \times F$  satisfying

$$\begin{array}{ccc} \pi^{-1}(U) & \xrightarrow{\varphi} & U \times F \\ \downarrow \pi & \swarrow \pi_1 & \\ U & & \end{array}$$

where  $\pi_1$  is the projection onto the first component.

The set  $\pi^{-1}(\{x\})$ , which is homeomorphic to  $F$ , is called the fiber over  $x$  and is denoted  $E_x$ . A fiber bundle can be thought of as attaching this manifold  $F$  to each point  $x \in M$  to obtain a new manifold  $E$ , and locally, i.e. in a neighbourhood of  $x$ , this manifold looks the cartesian product of this neighbourhood of  $x$  with  $F$ .

**Definition 3.11** (Topological Group). A topological group  $G$  is a topological space, which is also a group such that the binary operation and inverse maps,

$$\cdot : G \times G \rightarrow G \quad \text{and} \quad {}^{-1} : G \rightarrow G,$$

are continuous.

**Definition 3.12** (Group Action). Let  $G$  be a group and  $S$  a set. A right action of  $G$  on  $S$  is a map  $S \times G \rightarrow S$  such that,

- a) (Identity)  $x \cdot 1_G = x$ ,
- b) (Compatibility)  $(x \cdot g) \cdot h = x \cdot (gh)$ .

Note in the compatibility condition, the left hand side consists of two operations coming from the group action, compared to the right hand side which contains the group action and the binary operation coming from the group  $G$ . A group action is *transitive* if for any  $x, y \in S$ , there exists  $g \in G$  such that  $xg = y$ . A group action is *free* if  $g \cdot x = x$  for some  $x \in S$  implies  $g = 1_G$ .

**Definition 3.13** (Principal Bundle). Let  $G$  be a topological group and  $M$  a smooth manifold. A principal  $G$ -bundle over  $M$  is fiber bundle  $\pi : E \rightarrow M$  together with a continuous right action  $E \times G \rightarrow E$  such that  $G$  preserves the fibers of  $E$  and acts freely and transitively on them in such a way that for each  $x \in M$ ,  $y \in E_x$ , the map  $G \rightarrow E_x$  defined by  $g \rightarrow yg$  is a homeomorphism.

### 3.3 Lambda Lengths in Hyperbolic 3-Space

Recall hyperbolic 3-space is defined as the set,

$$\mathbb{H}^3 = \{(x, y, z) \in \mathbb{R}^3 \mid z > 0\},$$

together with the metric,

$$ds^2 = \frac{dx^2 + dy^2 + dz^2}{z}.$$

A horosphere in  $\mathbb{H}^3$  is a Euclidean ball tangent at the boundary of  $\mathbb{H}^3$ . Each point in a horosphere  $H \subset \mathbb{H}^3$  has two normal directions, one which points towards the center of the horoball, called the *outward* direction, and the other points away from the center, called the *inward* direction. Thus, we have two unit normal vector fields  $N^{in}$  and  $N^{out}$  on  $H$ . A frame at a point in  $\mathbb{H}^3$  is a right-handed orthonormal frame, namely a triple  $(f_1, f_2, f_3)$  such that  $f_1 \times f_2 = f_3$ . We can associate each frame to a matrix  $M \in \text{SO}(3)$  by writing each  $f_i$  in terms of the standard basis vectors for  $\mathbb{R}^3$ . Then the collection of all frames on  $\mathbb{H}^3$  forms a principal  $\text{SO}(3)$ -bundle over  $\mathbb{H}^3$  which we denote

$$\text{Fr} \rightarrow \mathbb{H}^3.$$

The universal cover of  $\mathrm{SO}(3)$  is  $\mathrm{Spin}(3)$ , so this bundle lifts to a principal  $\mathrm{Spin}(3)$ -bundle over  $\mathbb{H}^3$ , which we denote,

$$\mathrm{Fr}^S \rightarrow \mathbb{H}^3.$$

We refer to points of  $\mathrm{Fr}^S$  as spin frames.

**Definition 3.14** (Decorated Horospheres). *Let  $v$  be a unit parallel tangent vector field on a horosphere  $H$ .*

- *The inward frame field of  $v$  is the frame field on  $H$  given by  $F^{in} = (N^{in}, v, N^{in} \times v)$*
- *The outward frame field of  $v$  is the frame field on  $H$  given by  $F^{out} = (N^{out}, v, N^{out} \times v)$*

*A decorated horosphere is a pair  $(H, F)$  where  $F$  is the pair of frames  $F = (F^{in}, F^{out})$ .*

**Definition 3.15.** *An outward (resp. inward) spin decoration on  $H$  is a continuous lift of an outward (resp. inward) frame field from  $\mathrm{Fr}$  to  $\mathrm{Fr}^S$ .*

Given a unit parallel tangent vector field  $v$ , at each point of a horosphere  $H$ , we can rotate the inward (resp. outward) frame field by  $\pi$  or  $-\pi$  around  $v$ . In either case we obtain the same outward (resp. inward) frame field. On the other hand, given an outward (resp. inward) spin decoration, we can rotate by  $\pi$  or  $-\pi$  around  $v$  but we will obtain distinct inward (resp. outwards) spin decorations on  $H$ , related by a rotation of  $2\pi$ . We make the following convention for associating spin decorations. For an outward spin decoration  $W^{out}$  on  $H$ , the *associated* inward spin decoration is the spin decoration obtained by rotating  $W^{out}$  by  $\pi$  around  $v$  at each point of  $H$ . For an inward spin decoration  $W^{in}$  on  $H$ , the *associated* outward spin decoration is the spin decoration obtained by rotating  $W^{in}$  by  $-\pi$  around  $v$  at each point of  $H$ .

**Definition 3.16** (Spin Decorated Horospheres). *A spin decoration on a horosphere  $H$  is a pair  $W = (W^{in}, W^{out})$  of associated inward and outward spin decorations. We denote a spin-decorated horosphere by  $(H, W)$ , and denote the set of spin-decorated horospheres by  $\mathrm{Hor}^S$ .*

**Definition 3.17.** *Let  $p$  be a point on an oriented geodesic  $\gamma$  in  $\mathbb{H}^3$ . A frame  $F = (f_1, f_2, f_3)$  at  $p$  is adapted to  $\gamma$  if  $f_1$  is positively tangent to  $\gamma$ . A spin frame  $\tilde{F}$  at  $p$  is adapted to  $\gamma$  if it is the lift of a frame adapted to  $\gamma$ .*

Let  $p_1, p_2$  be two points on an oriented geodesic, and frames  $F_i = (f_1^i, f_2^i, f_3^i)$  at  $p_i$ , adapted to  $\gamma$ . Starting at  $p_1$ , parallel transport along  $\gamma$  to  $p_2$  carries  $f_1^1$  to  $f_1^2$  since both frames are adapted to  $\gamma$  at  $p_1$  and  $p_2$ . Then we have a frame  $F' = (f_1', f_2', f_3')$  which is the frame obtain by parallel transport of  $F_1$  along  $\gamma$ . Since  $f_1'$  agrees with  $f_1^2$ ,  $F'$  and  $F^2$  are related by a rotation  $\theta$ . For frames,  $\theta$  is well defined modulo  $2\pi$  and for spin frames  $\theta$  is well defined modulo  $4\pi$ . We can also define a signed distance  $\rho$ , which is the distance between  $p_1$  and  $p_2$  with the sign coming from the orientation on  $\gamma$ .

**Definition 3.18** (Complex Distance). *Let  $F_1, F_2$  be frames (or spin frames) at points  $p_1, p_2$ , on an oriented geodesic  $\gamma$ , adapted to  $\gamma$ . The complex distance from  $F_1$  to  $F_2$  is  $\rho + i\theta$ , where a translation along  $\gamma$  of signed distance  $\rho$ , followed by a rotation about  $\gamma$  of angle  $\theta$ , takes  $F_1$  to  $F_2$ .*

Let  $H_1, H_2$  be two horospheres, with center  $z_1, z_2 \in \partial\mathbb{H}^3$  respectively. Let  $\gamma$  be the oriented geodesic from  $z_1$  to  $z_2$ , and  $p_i = \gamma \cap H_i$ . Note if we have spin decoration  $W_i = (W_i^{in}, W_i^{out})$  on  $H_i$ , then  $W_1^{int}(p_1)$  and  $W_2^{out}(p_2)$  are adapted to  $\gamma$ .

**Definition 3.19** (Complex Lambda Lengths). *Let  $(H_1, W_1)$  and  $(H_2, W_2)$  be spin-decorated horospheres. The complex lambda length from  $(H_1, W_1)$  to  $(H_2, W_2)$  is,*

$$\lambda_{12} = \exp\left\{\frac{d}{2}\right\},$$

*where  $d$  is the complex distance from  $W_1^{in}(p_1)$  to  $W_2^{out}(p_2)$ . When the horospheres  $H_1$  and  $H_2$  have a common center, then the complex lambda length between them is zero.*

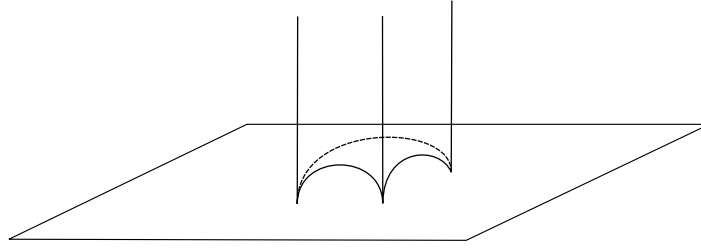


Figure 9: Ideal tetrahedron in hyperbolic 3-space.

Since the imaginary part of  $d$  is well defined modulo  $4\pi$ , the imaginary part of  $\frac{d}{2}$  is well defined modulo  $2\pi$ . Hence,  $\lambda_{12}$  is well defined.

We can embed an ideal tetrahedron in hyperbolic 3-space with three vertices on the  $xy$ -plane, and the fourth at the point at infinity (Figure 9). The sides of the tetrahedron are then geodesics, vertical lines from the points on the  $xy$ -plane to the points at infinity and circles which intersect the  $xy$ -plane at 90 degrees for points on the  $xy$ -plane. We can then place horospheres at each vertex of the tetrahedron, and we have a complex lambda length between horospheres. Theorem 3.8 can be generalised these horospheres (Figure 10), a proof of which can be found in Spinors and Horospheres [Mat23].

**Theorem 3.20** (Ptolemy Equations of a Hyperbolic Tetrahedron [Mat23]). *Let  $T$  be an ideal hyperbolic tetrahedron with vertices numbered 1, 2, 3, 4. Let  $H_i$ ,  $i \in \{1, 2, 3, 4\}$  be spin-decorated horospheres at each vertex of  $T$ . Let  $\lambda_{ij}$  be the complex lambda length between horospheres  $H_i$  and  $H_j$ . Then,*

$$\lambda_{13}\lambda_{24} = \lambda_{12}\lambda_{34} + \lambda_{14}\lambda_{23}.$$

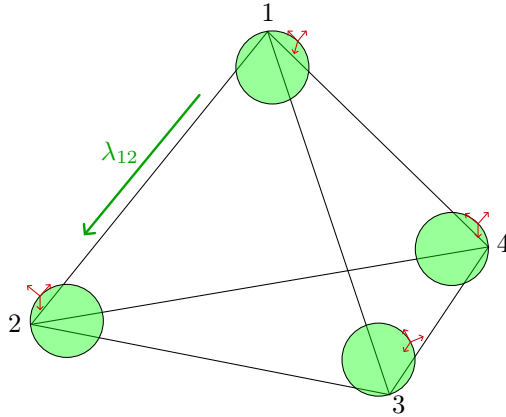


Figure 10: Horospheres and Lambda Lengths on an ideal tetrahedron. Each horosphere is decorated with a frame at each point, given in red.

## 4 A Symplectic Basis for 3-Manifold Triangulations

Now we construct oscillating curves dual to edge curves, which extends the Neumann-Zagier matrix to one which is symplectic (up to factors of 2). For proofs, we refer to A Symplectic Basis for 3-Manifold Triangulations [MP22]. Firstly some naming conventions. An *Edge Curve*  $C_i$  is a curve on a cusp which



encircles the edge with edge class  $i$ , in the triangulation. We prefix an element of the cusp triangulation by "cusp", to distinguish edges in the tetrahedron from edges in the cusp triangulation. The boundary curves on a torus cusp refer to a choice of simple closed curves  $\mathbf{m}_i, \mathbf{l}_i$  forming a basis for the fundamental group of cusp.

## 4.1 Triangulation

Let  $M$  be a 3-manifold with  $n_\epsilon \geq 1$  ends and an ideal triangulation  $\mathcal{T}$ . A decomposition into ideal tetrahedra and the corresponding cusp triangulation of the figure 8 knot is shown in Figure 11.

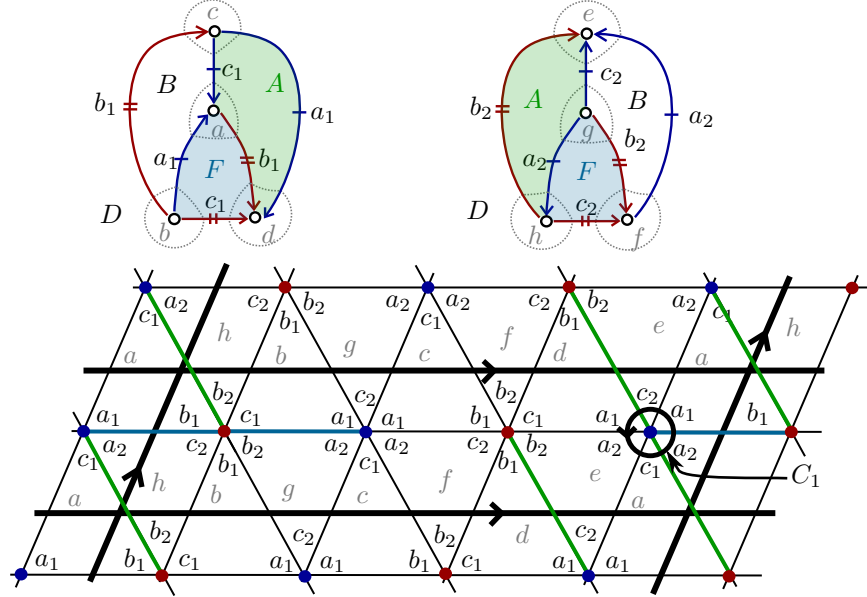


Figure 11: The figure-8 knot complement is made of two tetrahedra, with cusp triangulation shown. A choice of boundary curves  $\mathbf{l}$  and  $\mathbf{m}$  are shown, and one of the edge curves  $C_1$  [MP22].

## 4.2 Train Tracks

To construct an oscillating curve, we define the notion of train tracks for a triangulation. Starting with an ideal tetrahedron, we truncate the tetrahedron further by removing a neighbourhood of each edge. The train tracks lie on the boundary components of the tetrahedron, as indicated in red in Figure 12. More specifically the train tracks lie in the long rectangles (the boundary of the truncated edges) and cusp triangles (which contain short rectangles). Curves which dive through the manifold do so along long rectangles, and contribute no combinatorial holonomy. We say a curve is carried by the train tracks if it is lying on the train track. Curves run along short rectangles in order to dive through the manifold. We denote the train tracks on  $M$  by  $\tau$ . More details on the construction of train tracks can be found in [MP22].

## 4.3 End Multi Graph

Let  $n$  be the number of tetrahedra in the ideal triangulation of  $M$ , and  $n_E$  the number of edges. We also denote  $n_\epsilon$  as the number of ends (boundary components). Define the following graph.

**Definition 4.1** (End Multi Graph). *The end (multi) graph  $G$  of  $\mathcal{T}$  is defined as follows.*

1. The vertices of  $G$  are ends or boundary components of  $M$ .
2.  $G$  has one edge for each  $E$  of  $\mathcal{T}$ , with endpoints in  $G$  joining vertices corresponding to the same ends meeting  $E$ .

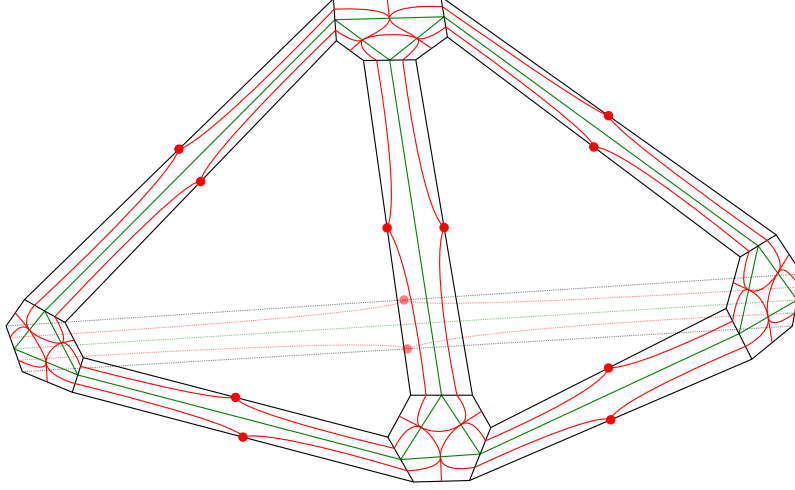


Figure 12: Train tracks on an ideal tetrahedron [MP22]

The end multi graph represents how the different boundary components of  $M$  are connected by edges of the cusp triangulation. We need the following properties of the end multi graph to construct oscillating curves.

**Lemma 4.2** (Mathews & Purcell). *There exists a collection  $\varepsilon'_c$  of  $n_c - 1$  edges of  $\mathcal{T}$ , forming a spanning tree for the end graph, which connects all ends of  $M$ .*

Since all trees are bipartite, we can 2-color the tree.

**Lemma 4.3** (Mathews & Purcell). *There exists an edge  $E_0$  of  $\tau$  which connects ends of the same color.*

The purpose of this extra edge is as follows. Denote  $\varepsilon_c = \varepsilon'_c \cup \{E_0\}$ . Then for any two ends of  $M$ , we can form a path between them using the edges in  $\varepsilon_c$  which has odd length. Aside from the edges  $\varepsilon_c$ , there are  $n_E - n_c$  edges of  $\mathcal{T}$ , which we denote  $E_1, \dots, E_{n_E - n_c}$ .

#### 4.4 Cusp Triangulation

**Lemma 4.4** (Mathews & Purcell). *Let  $e$  be an ideal edge of a triangulation, and suppose a curve carried by  $\tau$  runs through a fixed long rectangle  $R$  adjacent to  $e$ , through both branches on  $R$ . The long rectangle  $R$  lies in a tetrahedron  $\mathbf{t}$ , and is adjacent to a unique face  $F$  of  $\mathbf{t}$ . Then the curve must run through two triangles of the boundary triangulation adjacent to  $e$  in  $\mathbf{t}$ , entering the triangles along the side corresponding to the face  $F$ , and running across this side by way of a short rectangle containing one of the branches of  $\tau$ .*

This lemma allows us to 'dive through the manifold'. A curve can run along a short rectangle (i.e. adjacent to the edge of a cusp triangle). This curve then travels through the manifold along a long rectangle and exits at the other cusp vertex. The face adjacent to the short rectangle the curve runs along to enter the manifold is the same as the face adjacent to the short rectangle the curve exits the manifold along (see Figure 32).

#### 4.5 Oscillating Curves

Oscillating curves which are dual to the edge curves give rise to vectors that extend the Neumann-Zagier matrix to one which is symplectic (up to factors of 2).

**Definition 4.5** (Oscillating Curve). *Let  $\gamma$  be a curve on a boundary component of a 3-manifold  $M$ , which may dive through the manifold as described in Lemma 4.4. If  $\gamma$  is oriented such that  $\gamma$  reverses orientation as it passes through the manifold,  $\gamma$  is called an oscillating curve.*

We say an oscillating curve is dual to an edge curve, if the oscillating curve intersects the edge curve once. Since the curve reverses orientation as it passes through the manifold, the curve consists of an even number

of components. We will use the End Multi Graph, in particular the unique path of odd length, to determine the cusps which these components will lie in. The existence of such curves is a result of the following lemma.

**Lemma 4.6** (Mathews & Purcell). *There exists a collection of disjoint curves  $\Gamma_1, \dots, \Gamma_{n_E - n_c}$ , each disjoint from all  $\mathbf{m}_i$  and  $\mathbf{l}_i$ , with  $\Gamma_i$  meeting  $C_i$  exactly once, and disjoint from  $C_j$  when  $i \neq j$ . Moreover, each such curve runs through an even number of long rectangles along branches in that long rectangle, and runs between distinct sides of each cusp triangle that it meets.*

In the case of knot complements, each oscillating curve  $\Gamma_i$  dives through the manifold twice (once through the edge class for the curve  $E_i$  and once through the common edge class  $E_0$ ). A choice of oscillating curves is drawn in Figure 13.

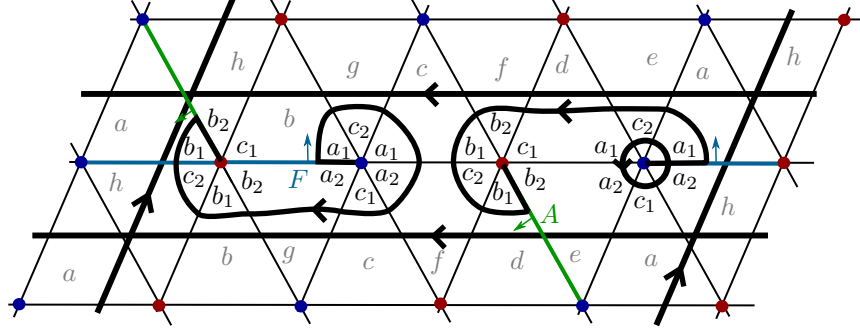


Figure 13: The arcs of the dual oscillating curve  $\Gamma$  for the figure-8 knot [MP22].

## 4.6 Neumann-Zagier Matrix

For a given triangulation, the basis  $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i$  are encoded as vectors:

$$\mathbf{a}_1 = (1, 0, 0, 0, \dots, 0) \quad \mathbf{b}_1 = (0, 1, 0, 0, \dots, 0) \quad \mathbf{c}_1 = (0, 0, 1, 0, \dots, 0)$$

and so on. We calculate the combinatorial holonomy for the curves  $\mathbf{m}_i, \mathbf{l}_i, C_i, \Gamma_i$ , in terms of this basis. For the figure 8 knot and the oscillating curves shown in Figure 13 we obtain the following incidence vectors. The basis  $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i$  is illustrated in Figure 11.

$$\mathbf{m} : (0, 1, 0, 1, 0, 0)$$

$$\mathbf{l} : (0, 2, 2, 0, 2, 2)$$

$$C : (2, 0, 1, 2, 0, 1)$$

$$\Gamma : (1, 1, 1, 1, 1, 1)$$

The Neumann-Zagier matrix consists of the rows  $C, \mathbf{m}$  and  $\mathbf{l}$ , with the relation  $a_i + b_i + c_i = 0$  imposed. So we replace the columns  $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i$  with  $\mathbf{a}_i - \mathbf{c}_i, \mathbf{b}_i - \mathbf{c}_i$ . The  $\Gamma$  row extends this matrix to one which is symplectic (up to factors of 2)  $SY$ .

**Example 4.7.** Imposing the relation  $a_i + b_i + c_i = 0$  on the combinatorial holonomy for the curves found above, we obtain the following matrix  $SY$ .

$$SY = \begin{pmatrix} 0 & -1 & 1 & 0 \\ 2 & 0 & -2 & 0 \\ 1 & -1 & 1 & -1 \\ 0 & 0 & -2 & 0 \end{pmatrix}$$

We evaluate  $\omega$  for pairs of curves (rows of  $SY$ ) obtained from the Figure 8 knot.

$$\begin{aligned}
\omega(h(\mathbf{m}), h(\mathbf{l})) &= \begin{vmatrix} 0 & -1 \\ 2 & 0 \end{vmatrix} + \begin{vmatrix} 1 & 0 \\ -2 & 0 \end{vmatrix} \\
&= 2 \\
\omega(h(\Gamma), h(C)) &= \begin{vmatrix} 0 & 0 \\ 1 & -1 \end{vmatrix} + \begin{vmatrix} -2 & 0 \\ 1 & -1 \end{vmatrix} \\
&= 2 \\
\omega(h(\mathbf{m}), h(\Gamma)) &= \begin{vmatrix} 0 & -1 \\ 0 & 0 \end{vmatrix} + \begin{vmatrix} 1 & 0 \\ -2 & 0 \end{vmatrix} \\
&= 0
\end{aligned}$$

This leads to the main theorem [MP22], let  $g = \sum_{i=0}^n g_i$  be the sum of genera for each boundary component of  $M$ .

**Theorem 4.8** (Mathews & Purcell). *There exists a constructive algorithm to determine oscillating curves  $C_1, \dots, C_{n_E - n_C}$  associated with edge gluings, and dual oscillating curves  $\Gamma_1, \dots, \Gamma_{n_E - n_C}$ , so that these curves along with a standard basis for  $H_1(M)$ ,  $\mathbf{m}_1, \mathbf{l}_1, \dots, \mathbf{m}_g, \mathbf{l}_g$  form a symplectic basis, in the sense that*

$$\begin{aligned}
\omega(h(C_i), h(\Gamma_j)) &= 2C_i \cdot \Gamma_j = 2\delta_{ij} \\
\omega(h(\mathbf{m}_i), h(\mathbf{l}_j)) &= 2\mathbf{m}_i \cdot \mathbf{l}_j = 2\delta_{ij}
\end{aligned}$$

and  $\omega$ , evaluated on the combinatorial holonomy of any other pair of curves, is zero.

## 5 Algorithm

Theorem 4.8 gives the existence of an algorithm to construct oscillating curves on triangulated 3-manifolds. We develop an algorithm to construct oscillating curves dual to the edge curves, which is new to this paper, and furthermore we implement the algorithm in C within the SnapPy kernel. There are 4 main steps to our algorithm,

1. Initialise cusp triangles and cusp regions (Sections 5.1 & 5.2),
2. Construct the end multi graph (Section 5.5),
3. Construct train lines on each cusp (Section 5.6),
4. Construct oscillating curves (Section 5.7).

Finally, we calculate the combinatorial holonomy of the oscillating curves dual to the edge curves and piece together the rows into the final matrix  $SY$ . This section describes the algorithm at a high level and the different choices it makes to construct the oscillating curves. The details of each of the structures are described in Section 6.

### 5.1 Cusp Triangles

For a manifold with an ideal triangulation, the removed vertices of an ideal tetrahedron give a collection of triangles, called *cusp triangles*, which form a triangulation of each boundary component. The `CuspTriangle`'s are derived from the `Triangulation` object provided by SnapPy. Each `CuspTriangle` stores the information about the triangle such as the neighbouring `CuspTriangle`'s and a `CuspVertex` at each cusp vertex. Each edge of the cusp triangle lies in a face of a tetrahedron, and each vertex lies in an edge of a tetrahedron. So at each cusp vertex we have an edge class, the edge class of the edge of the triangulation the vertex lies in. Each cusp vertex also has an associated edge index. When we want to dive through the manifold along an edge, the approach will be to find a cusp triangle which has a vertex with edge class of the target edge.

## 5.2 Cusp Regions

Our general approach to constructing oscillating curves will be to construct a graph representing how we can move around the cusp, and then using breadth first search to find a path through the cusp. If we used the `CuspTriangles` to construct this graph, the curve we construct may intersect with the boundary curves  $\mathbf{m}_i, \mathbf{l}_i$ , or with previously constructed oscillating curves. One approach to tackle this problem could be to construct a graph dual to the cusp triangles initially, and then altering the graph at each iteration to take into account the new curves. The downside to this approach is the complexity of altering the graph from the sheer number of configurations possible after constructing a curve. Instead, we build an intermediate object which handles these changes locally, so the program can work out for itself how the graph has changed after constructing a curve. This intermediate object is called a `CuspRegion`.

**Definition 5.1** (Cusp Region). *Let  $\mathcal{T}$  be a torus with a triangulation, and  $\gamma_1, \dots, \gamma_n$  curves which are either simple closed curves, or begin and end at vertices of the triangulation, and which intersect edges of the triangulation transversally. A Cusp Region is a connected component of the torus with the curves and the edges of each triangle removed.*

In other words, curves on a `CuspTriangle` subdivide the cusp triangle into distinct `CuspRegions`. An example of two different cusp regions are shown in Figure 15. Each `CuspRegion` keeps track of various attributes associated to the region, such as the cusp triangle it lies in, the cusp edges the region meets, the number of curves on each cusp edge (used for adjacent cusp regions, Section 5.3), and the short rectangles an oscillating curve in the cusp region can dive along. Initially, the cusps contain the curves  $\mathbf{m}, \mathbf{l}$  which form a basis for the fundamental group of the cusp. The boundary curves,  $\mathbf{m}, \mathbf{l}$ , intersect at a point, once and on the interior of a cusp triangle. The construction of the `CuspRegions` when the cusp only contains the boundary curves is split into those on the cusp triangle with the intersection point, and those not on this cusp triangle. We describe the initialisation of the cusp regions for these two types of cusp triangles in Section 5.2.1 and 5.2.2. After constructing a curve on a cusp, we split cusp regions locally at each cusp triangle, which is described later in Section 5.8.

### 5.2.1 Intersection Cusp Triangle

To simplify the initialisation of the `CuspRegion`'s coming from the curves  $\mathbf{m}, \mathbf{l}$ , we make use of the following lemma.

**Lemma 5.2** (Boundary Curves). *Let  $\mathcal{T}$  be a torus with a triangulation. The function `peripheral_curves()` contained in the SnapPy kernel places curves  $\mathbf{m}, \mathbf{l}$  on  $\mathcal{T}$  such that,*

- i)  $\mathbf{m}, \mathbf{l}$  are simple closed curves on  $\mathcal{T}$ ,
- ii)  $\mathbf{m}, \mathbf{l}$  generate the fundamental group of  $\mathcal{T}$ ,
- iii) The intersection point of  $\mathbf{m}$  and  $\mathbf{l}$  lies on the interior of a triangle,
- iv) The curves enter the triangle containing the intersection point once.

SnapPy already guarantees conditions i-iii, the new property described by this lemma is part iv. Property iv is not necessary for the construction of oscillating curves, rather it simplifies the construction of the cusp regions when the cusp contains only the boundary curves. Moreover, the rest of the algorithm does not depend on this assumption, so one could replace this initialisation function with a general one. We analyse the functions `pick_base_tet`, `set_up_perimeter` and `expand_perimeter`, which are contained in `kernel/kernel_code/peripheral_curves.c` in the SnapPy source code [Cul+23]. These are called by `peripheral_curves()` to construct the boundary curves. In essence, `peripheral_curves()` constructs a graph on each cusp, which forms a spanning tree of a fundamental domain. The curves are found by traversing this tree from leaves of the tree to the root, in such a way that the result is a pair of curves which generate the fundamental group. The root of the tree then becomes the intersection of the curves, which is the node we are concerned with.

*Proof.* We give the function `pick_base_tet` a triangulation of a 3-manifold and a cusp. The function looks through the tetrahedra of the triangulation in order, and within each tetrahedron it looks through its ideal vertices in order, until it finds an ideal vertex at the given cusp, and stores the tetrahedron and vertex in `base_tet` and `base_vertex`.

```
static void pick_base_tet(
    Triangulation *manifold,
    Cusp *cusp,
    Tetrahedron **base_tet,
    VertexIndex *base_vertex)
{
    Tetrahedron *tet;
    VertexIndex v;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
        for (v = 0; v < 4; v++)
            if (tet->cusp[v] == cusp)
            {
                *base_tet = tet;
                *base_vertex = v;
                return;
            }

    /*
     * If pick_base_tet() didn't find any vertex belonging
     * to the specified cusp, we're in big trouble.
     */
    uFatalError("pick_base_tet", "peripheral_curves");
}
```

The function `set_up_perimeter()` initialises the data structures needed for the search. We give `set_up_perimeter` the previously found `base_tet` and `base_vertex`. The `PerimeterPiece` object is used by SnapPy to keep track of the current search region of the cusp, and later to store the tree of cusp triangles. Each tetrahedron is given an array of 4 `extra` structs, which represent the cusp triangle coming from each vertex of the tetrahedron. The key point is the `visited` attribute of the starting cusp triangle, stored in the `extra->visited` attribute on each tetrahedron, is set to `TRUE`. We will see in the next function that `visited` is responsible for the control flow of the search.

```
static void set_up_perimeter(
    Tetrahedron *base_tet,
    VertexIndex base_vertex,
    PerimeterPiece **perimeter_anchor)
{
    int i;
    PerimeterPiece *pp[3];

    base_tet->extra[base_vertex].visited = TRUE;
    base_tet->extra[base_vertex].parent_tet = NULL;
    base_tet->extra[base_vertex].orientation = right_handed;

    for (i = 0; i < 3; i++)
        pp[i] = NEW_SIRUCT(PerimeterPiece);

    for (i = 0; i < 3; i++)
```

```

{
    pp[i]->tet          = base_tet;
    pp[i]->vertex       = base_vertex;
    pp[i]->face         = vt_side[base_vertex][i];
    pp[i]->orientation  = right_handed;
    pp[i]->checked      = FALSE;
    pp[i]->next         = pp[(i+1)%3];
    pp[i]->prev         = pp[(i+2)%3];
}

*perimeter_anchor = pp[0];
}

```

The function `expand_perimeter()` expands the search of the cusp. SnapPy uses the `PerimeterPiece` object to keep track of the current search area, which is stored as a tree with root at the base cusp triangle. If the `visited` attribute of a cusp triangle is set to `TRUE`, the search continues to a different cusp triangle, ignoring this visited one. If the `visited` attribute is set to `FALSE`, the search expands into this cusp triangle, a vertex and edge are added to the tree. But in either case, the `visited` parameter is only set to `TRUE`, with the search only expanding into a cusp triangle when `visited == FALSE`. Since we set `visited` to `TRUE` for the base cusp triangle in `set_up_perimeter`, we never revisit the starting cusp triangle and therefore the only curve sections on the intersection cusp triangle are the sections which intersect.

```

static void expand_perimeter(PerimeterPiece *perimeter_anchor) {
    int num_unchecked_pieces;
    PerimeterPiece *pp,
                  *new_piece;

    Permutation    gluing;
    Tetrahedron    *nbr_tet;
    VertexIndex    nbr_vertex;
    FaceIndex      nbr_back_face,
                  nbr_left_face,
                  nbr_right_face;
    Orientation     nbr_orientation;

    for (num_unchecked_pieces = 3, pp = perimeter_anchor;
         num_unchecked_pieces;
         pp = pp->next)

        if (pp->checked == FALSE)
        {
            gluing      = pp->tet->gluing[pp->face];
            nbr_tet      = pp->tet->neighbor[pp->face];
            nbr_vertex    = EVALUATE(gluing, pp->vertex);
            if (nbr_tet->extra[nbr_vertex].visited)
            {
                pp->checked = TRUE;
                num_unchecked_pieces--;
            }
            else
            {
                /*
                 * Extend the tree to the neighboring vertex.
                 */

                ...
            }
        }
}

```

```

    nbr_tet->extra[nbr_vertex].visited      = TRUE;
    nbr_tet->extra[nbr_vertex].parent_tet    = pp->tet;
    ...
  }
}

```

□

In the case of tori boundary components, we have two different ways the boundary curves can intersect, shown in Figure 14. While we could hard code the cusp region attributes for these two cases, rather we have implemented a general function which finds the cusp regions on a triangle containing an arbitrary number of intersecting curves.

We have two distinct types of cusp regions on the intersection cusp triangle. The regions which are incident to a cusp vertex and those which are not. An example of these regions is shown in Figure 14, with a region that meets a cusp vertex shown on the left, and one which meets one cusp edge shown on the right. We use the number of curves which pass around each vertex to enumerate the cusp regions which meet one cusp edge. For a cusp triangle at vertex  $v$  of a tetrahedron, we find the number of cusp regions at cusp edge  $e$ . The cusp triangle contains two other edges,  $e_1, e_2$ . Then  $\text{Flow}(e, e_1) + \text{Flow}(e, e_2) + 1$  is the number of cusp regions which meet cusp edge  $e$ , where  $\text{Flow}(e_1, e_2)$  is the number of curves passing between edge  $e_1$  and edge  $e_2$ .

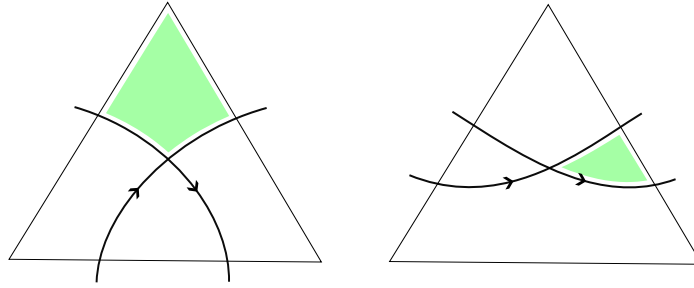


Figure 14: Two types of intersection cusp triangles.

### 5.2.2 Remaining Cusp Triangles

The remaining cusp regions contain arcs of boundary curves, each of which are disjoint, for which we do not make any assumption about the number of curves on each cusp triangle. Similarly to the intersection cusp triangle, we have two distinct types of regions on the remaining cusp triangles. Each arc of a curve on the cusp runs between distinct sides of the cusp triangle, so we have two distinct types of cusp regions. One type of region lies "within" the arcs on a cusp triangle. These regions meet two cusp edges, and we enumerate them using the number of arcs which pass around the vertex. The other region is the "center" cusp region which meets all three cusp edges of the cusp triangle. An example of these two regions is show in Figure 15.

## 5.3 Adjacent Cusp Regions

We define the notion of adjacent cusp regions, which will be used to construct the cusp region graph in Section 5.4.

**Definition 5.3** (Adjacent Cusp Region). *Let  $\mathcal{T}$  be a torus with a triangulation, and  $\gamma_1, \dots, \gamma_i$  curves which are either simple closed curves, or begin and end at vertices of the triangulation. Two cusp regions  $r, r'$  are adjacent if there exists a curve  $\gamma$  on  $\mathcal{T}$  such that,*



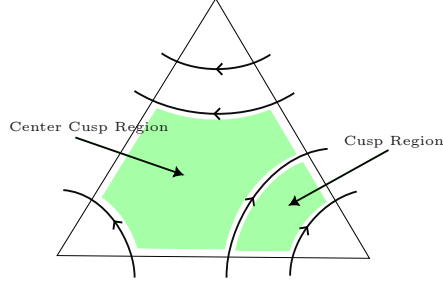


Figure 15: Cusp regions of cusp triangle without intersecting curves.

- i)  $\gamma$  meets exactly two cusp regions,  $r$  and  $r'$ ,
- ii)  $\gamma$  is disjoint from  $\gamma_1, \dots, \gamma_i$ , and the vertices of the triangulation.

Two cusp regions  $r, r'$  are adjacent across an edge  $e$  of the triangulation if there exists a curve  $\gamma$  satisfying the above properties and  $\gamma$  intersects the edge  $e$  transversally.

An example of adjacent cusp regions is shown in Figure 16. This definition differs slightly from natural notion of adjacency, which would consider  $r_3$  and  $r_5$  in Figure 16 adjacent.

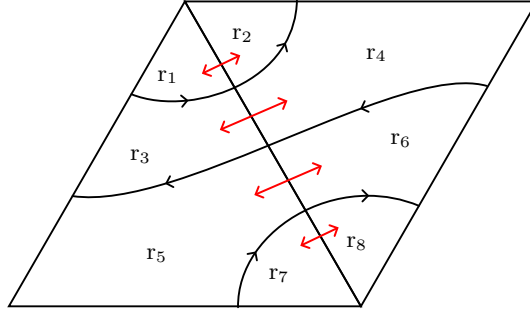


Figure 16: Adjacent cusp regions are indicated by red arrows.

For a given cusp triangle, we can determine which cusp triangles are adjacent across cusp edges. This is derived from the gluing of the tetrahedra, but this is not enough to know which cusp regions are adjacent, as there may be multiple cusp regions which meet the edge of the cusp triangle. Given two adjacent cusp triangles,  $t_1$  and  $t_2$ , let  $e$  be an edge of  $t_1$  which is adjacent to  $t_2$ . The edge  $e$  has two cusp vertices, one at either end, which we label  $x, y$ . The following proposition gives a combinatorial method for determining if two cusp regions are adjacent.

**Proposition 5.4.** Suppose  $r_1, r_2$  are two cusp regions, lying on cusp triangles  $t_1, t_2$  respectively.  $r_1$  is adjacent to  $r_2$  across the edge  $e$  iff all the following conditions are satisfied,

- i)  $t_1$  is adjacent to  $t_2$  across edge  $e$ ,
- ii)  $r_1$  and  $r_2$  meet the edge  $e$ ,
- iii) the number of curves which cross  $e$  between  $x$  and  $r_1$  equals the number of curves which cross  $e$  between  $x$  and  $r_2$ ,
- iv) the number of curves which cross  $e$  between  $y$  and  $r_1$  equals the number of curves which cross  $e$  between  $y$  and  $r_2$ .

*Proof.* Clearly these conditions are necessary for two cusp regions to be adjacent. Suppose  $r_1, r_2$  satisfy condition (i) – (iv). Conditions (i), (ii) give the cusp regions meet a common edge of cusp triangulation. Consider this edge as an interval  $(0, 1)$ . Let  $n$  be the curves which cross  $e$  between  $x$  and  $r_1$ , and  $n'$  curves which cross  $e$  between  $y$  and  $r_1$ . Hence, there are a total  $n + n'$  curves which cross  $e$ , so we subdivide  $(0, 1)$  into  $n + n' + 1$  sub-intervals,

$$\left(0, \frac{1}{n + n'}\right), \dots, \left(\frac{n + n' - 1}{n + n'}, 1\right)$$

Since curves run between distinct sides of a cusp triangle,  $r_1$  meets  $e$  on exactly one sub-interval  $(\frac{j}{n+n'}, \frac{j+1}{n+n'})$ . Condition (iii) implies  $j = n$ . The cusp region  $r_2$  is meets  $e$  in the same partition, and hence the cusp regions are adjacent across the edge  $e$  of cusp triangle  $t_1$ .  $\square$

We only use condition (iv) to determine the total number of curves which cross the edge of the cusp triangle. Instead of storing the number of curves towards each vertex along the cusp edge, we could store the number of curves towards one vertex along the cusp edge and the total number of curves which cross each edge. In practice, the former is simpler when working with the curves, both for finding adjacent regions and splitting the cusp regions (Section 5.8).

## 5.4 Cusp Region Graph

We define the cusp region graph, which we use to construct curves on a cusp which are disjoint from other curves on the cusp.

**Definition 5.5** (Cusp Region Graph). *The Cusp Region graph  $G_r$ , of a cusp  $\mathcal{C}$  with a collection of curves  $\gamma_1, \dots, \gamma_i$  with are either simple closed curves or begin and end at vertices of a cusp triangle, is defined as follows.*

- a) *The vertices of  $G_r$  are the cusp regions of  $\mathcal{C}$ .*
- b)  *$G_r$  has an edge between two regions  $r_0, r_1$  if they are adjacent cusp regions.*

The Cusp Region graph is a subgraph of the graph dual to the cusp regions. The Cusp Region graph simplifies the task of constructing a curve on a cusp which is disjoint from other curves on the cusp, to the problem of graph pathfinding. An example of the Cusp Region graph is shown in Figure 17.

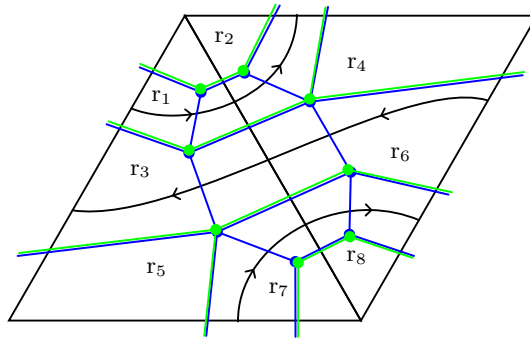


Figure 17: The blue graph is dual to the cusp regions, the green graph is the cusp region graph for the curves shown.

In order to construct various curves on the cusp, we build up a curve from a path through the cusp region graph as follows. Consider a path through the cusp region graph  $r_1, \dots, r_n$ . Each  $r_i$  represents a cusp region, so we identify  $r_i$  with an arbitrary point in this cusp region. Since we have an edge  $r_i \rightarrow r_{i+1}$ , these cusp regions are adjacent. Hence, there exists a curve  $\beta_i$  from  $r_i$  to  $r_{i+1}$ . Piecing together each of these  $\beta_i$ 's, we obtain a curve on the cusp which is disjoint from other curves on the cusp. The algorithm for constructing the Cusp Region graph is fairly straightforward, stated in Algorithm 1.

---

**Algorithm 1:** Cusp Region graph

---

```
input : Cusp Regions
output: Cusp Region graph
1  $g \leftarrow \text{Graph}(\text{num\_vertices} = \text{cusp.num\_cusp\_regions});$ 
2 for each cusp region  $r$  do
3   for each edge  $e$  of the cusp triangle  $r$  is contained in do
4     if  $r$  does not meet  $e$  then
5       | continue
6     end
7      $r' \leftarrow$  cusp region adjacent to  $r$  across edge  $e$ ;
8     if  $g$  does not contain the edge  $(r, r')$  then
9       |  $g.\text{insert\_edge}(r, r')$ ;
10    end
11  end
12 end
13 return  $g$ ;
```

---

## 5.5 End Multi Graph

The end multi graph describes how the cusps of  $M$  connect via edges in the triangulation. We have three computations to perform

- i) Construct the end multi graph, as defined in Definition 4.1.
- ii) Find a spanning tree for this graph,
- iii) Find an edge in the end multi graph such that when added to the spanning tree, forms a cycle of odd length.

To find a spanning tree, we use breadth first search, since we have an unweighted graph. To find the edge which forms a cycle of odd length, we 2-color the spanning tree. Then we find an edge of the end multi graph which connects vertices of the same color. Figure 18 shows an example of the cusps for a manifold, and the end multi graph.

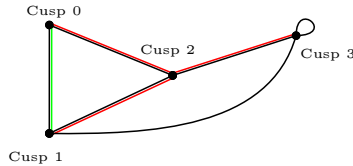


Figure 18: Example end multi graph of a manifold with 4 cusps and 6 edge classes. A spanning tree for the end multi graph is indicated in red and a choice of  $E_0$  is shown in green.

Algorithm 2 describes the construction of the end multi graph, and Algorithm 3 finds the edge  $E_0$ .

## 5.6 Train Lines

We motivate train lines with the following example.

**Example 5.6.** Consider the link L11a467, shown in Figure 19. This link has 3 components, so the link complement in  $S^3$  has 3 tori boundary components, and SnapPy gives the link a triangulation consisting of 17 tetrahedra.

---

**Algorithm 2:** End Multi Graph

---

```
input : Triangulation
output: End Multi Graph
1  $g \leftarrow \text{Graph}(\text{num\_vertices} = \text{manifold.num\_cusps});$ 
2 for each  $tet$  in  $\text{manifold.tet\_list}$  do
3   for each pair of vertices  $v1, v2$  of  $tet$ ,  $v1 \neq v2$  do
4      $/* tet.cusp[v]$  is the cusp at vertex  $v$  of  $tet$ . */
5      $g.insert\_edge(tet.cusp[v1], tet.cusp[v2]);$ 
6   end
7 end
8 return  $\text{spanning\_tree}(g);$ 
```

---

---

**Algorithm 3:** Find edge  $E_0$ 

---

```
input : Graph and a spanning tree of the graph
output: An edge of the graph which forms an odd length cycle in the tree
1  $q \leftarrow \text{Queue}();$ 
2  $q.enqueue(0);$ 
3 while  $!q.is\_empty()$  do
4    $v1 \leftarrow q.dequeue();$ 
5   for  $v2$  adjacent to  $v1$  in the spanning tree do
6     if  $v2$  is not colored then
7        $v2.color \leftarrow \text{reverse}(v1.color);$ 
8        $q.enqueue(v2);$ 
9     end
10  end
11 end
12 for  $v1$  in the end multi graph do
13   for  $v2$  adjacent to  $v1$  do
14     if  $v1.color == v2.color$  then
15       return  $\text{edge\_class}(v1, v2);$ 
16     end
17   end
18 end
```

---

The triangulation includes an edge from cusp 0 to cusp 0, which we choose to be  $E_0$ , an edge from cusp 0 to cusp 1 and cusp 0 to cusp 2 which we label  $E$  and  $E'$  respectively. There are also edges  $E_1$  from cusp 1 to cusp 1 and  $E_2$  from cusp 2 to cusp 2. We take  $\{E, E'\}$  to form a spanning tree for the end multi graph. Suppose we are constructing oscillating curves on this link. We naively find oscillating curves which intersect the curves  $C_1$  and  $C_2$  which run around the  $E_1$  and  $E_2$  respectively, shown in Figure 20.

Notice on cusp 0, we have a region colored in blue which has become disconnected from the rest of the cusp. In order to construct oscillating curves, we require a path connected cusp.

We may still be able to construct the remaining oscillating curves through careful consideration of each cusp, but it suggests the more pathological example.

**Example 5.7.** Consider a hypothetical link complement with a cusp in the end multi graph of degree 6, shown on the left of Figure 21. Suppose we have 9 oscillating curves to construct which pass through this cusp indicated by the red, green and blue lines on the left of Figure 21. Cusp 0, drawn on the right of Figure 21, then contains 6 vertices, one for each edge class, and we are trying to find 9 curves between these vertices. This is not possible as the complete, bipartite graph  $K_{3,3}$  is not a planar graph, a result also known as the three-utilities problem.

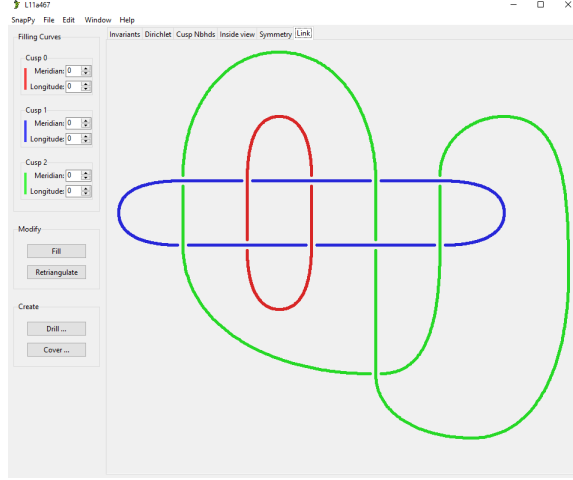


Figure 19: Link L11a467.

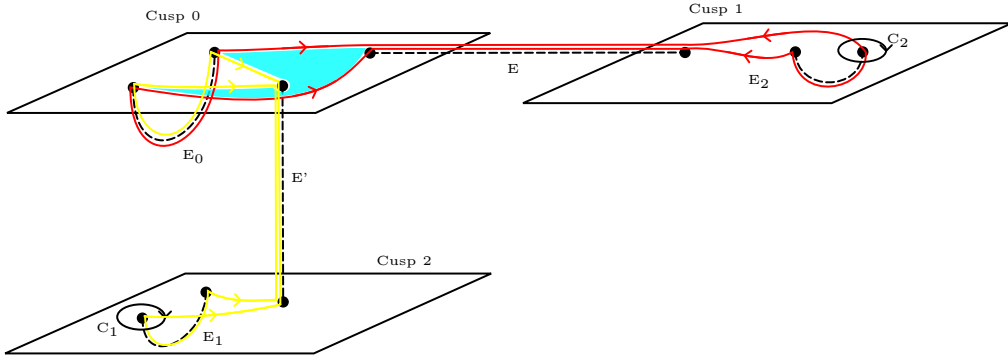


Figure 20: Two oscillating curves on L11a467, one in red and the other in yellow.

Our solution to this problem involves constructing a single 'template' curve, called a train line, which we place the oscillating curves on. The curve will be constructed in a way such that we can dive into the manifold along any edge class in the end multi graph, without picking up holonomy for any pair of curves on the train line.

**Definition 5.8** (Train Line). *A train line on a cusp  $\mathcal{C}$  with a triangulation, for a collection of edge classes  $i_1, \dots, i_k$  is a curve  $\gamma$  on the cusp  $\mathcal{C}$ , with boundary curves  $\mathbf{m}_i, \mathbf{l}_i$ , such that*

- i)  $\gamma$  is disjoint from the boundary curves,*
- ii)  $\gamma$  runs between distinct sides of each cusp triangle it enters,*
- iii)  $\gamma$  intersects cusp edges transversally,*
- iv) For  $1 \leq j \leq k$ , the curve  $\gamma$  meets a cusp edge incident to a cusp vertex corresponding to  $i_j$ , with no other curves between the intersection point and the cusp vertex.*

**Definition 5.9** (Train Line Siding). *Let  $\mathcal{C}$  be a cusp with a triangulation, for a collection of edge class  $i_1, \dots, i_k$ , and  $\gamma$  a train line for this collection. A train line siding for an edge class  $i_j$  is a curve  $\beta : [0, 1] \rightarrow \mathcal{C}$ , such that*

- i)  $\beta(0)$  is a cusp vertex corresponding to edge class  $i_j$ ,*

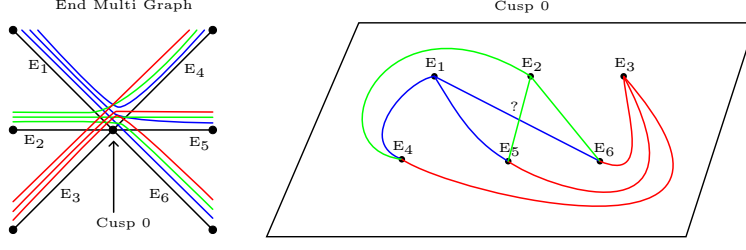


Figure 21: Part of a hypothetical link complement.

- ii)  $\beta$  is tangent to a cusp edge  $e$  at  $t = 0$ ,
- iii)  $\beta(1)$  lies on the curve  $\gamma$ , and this arc of  $\gamma$  intersects the cusp edge  $e$ ,
- iv)  $\beta$  is disjoint from all curves except at  $\beta(1)$ ,
- v)  $\beta$  meets exactly one cusp region  $r$ ,

A train line with sidings for each edge class in the collection is called a *train line with sidings*. In order to construct cusp regions on a cusp we require curves which start and end at cusp vertices. For a train line with sidings, as currently defined the ends of the train line curve,  $\gamma$ , does not start and end at cusp vertices. So we shrink the curve  $\gamma$ , so it begins at the end of the first siding, and ends at the end of the last siding. Then to construct the cusp regions, we consider the curve  $\gamma$ , combined with the train line sidings  $\beta_1, \dots, \beta_k$  as one curve. An example of a train line with sidings is shown in Figure 22. We will construct a train line on each cusp of a triangulated 3-manifold, with compatible sidings (such that the curves forming the sidings satisfy Lemma 4.4), using the following steps.

- i) Find the collection of edge classes for the train lines (Section 5.6.1),
- ii) Choose a siding for each edge class (Section 5.6.2),
- iii) On each cusp, find curves between the sidings to form the train line (Section 5.6.3).

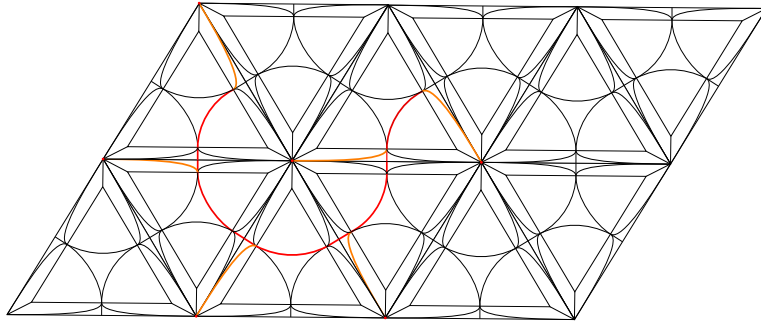


Figure 22: Example of a train line with sidings, the train line is the red curve, with sidings in orange.

### 5.6.1 Edge Classes

Firstly we build a collection of edge classes which the train lines will use. We have a collection of edges which make up the spanning tree for the end multi graph  $\varepsilon'_c$ , and  $\varepsilon_c = \varepsilon'_c \cup \{E_0\}$ . For each cusp  $C_i$ , we have a collection of edges

$$H(\mathcal{C}_i) = \{j \in \{1, \dots, n\} \mid \text{the edge with edge class } j \text{ has an end which lies in } \mathcal{C}_i\}.$$

Then we define the collection

$$K(\mathcal{C}_i) = H(\mathcal{C}_i) \cap \varepsilon_c.$$

Consider an edge class  $i \notin \varepsilon_c$ . The edge with edge class  $i$  has two ends, each of which lie in a cusp of the end multi graph,  $\mathcal{C}_j, \mathcal{C}_k$ . The graph consisting of edges from  $\varepsilon_c$  contains a unique path of odd length between any two vertices. Let  $l(i)$  be the length of the unique path of odd length from  $\mathcal{C}_j$  to  $\mathcal{C}_k$ . Then we define a new collection,

$$L(\mathcal{C}_i) = K(\mathcal{C}_i) \cup \{j \in H(\mathcal{C}_i) - \varepsilon_c \mid l(j) > 2\}.$$

This final collection gives, on each cusp, the edge classes in the spanning tree for the end multi graph,  $E_0$  and the edge class associated to an oscillating curve with more than 2 curve components.

### 5.6.2 Train Line Sidings

The sidings will be used by oscillating curves to dive into the manifold (Lemma 4.4). We find the sidings before the train lines, so we have two choices to make.

- i) Which cusp region does  $\beta$  enter?
- ii) Which cusp edge is  $\beta(0)$  tangent to?

We use the following conjecture to place sidings on distinct cusp triangles.

**Conjecture 5.10.** *Let  $M$  be a triangulated 3-manifold consisting of  $n$  tetrahedra. Let  $G$  be a graph defined as follows,*

- i)  *$G$  contains a vertex for each tetrahedron and each edge class of  $M$ .*
- ii)  *$G$  contains an edge  $(t, i)$  if an edge of tetrahedron  $t$  has edge class  $i$ .*

*Then  $G$  has a bipartite matching*

This is plausible since we have  $n$  tetrahedra and  $n$  edge classes, with each edge class coming from an edge which lies in a number of tetrahedra. In practice this conjecture seems to hold, and we only require the result for a subset of edge classes. A similar result holds for knots, consider a knot as a 4-valent graph in the plane. Kauffman showed there exists a pairing between crossings of the knot and regions bound by knot which are incident with the crossing, with no two crossings assigned to the same region [Kau83]. Rather, we would like a pairing between vertices in a cusp triangulation and the incident cusp triangles. This conjecture assigns edge classes in  $M$  to tetrahedra in the triangulation. Each train line siding is associated to an edge class, and a cusp region. This cusp region is contained in a cusp triangle, which in turn is contained in a tetrahedron. This assignment sets the tetrahedron we use for the siding. Since we have a bipartite matching, the cusp regions for each train line siding are contained in cusp triangles from different tetrahedra, and hence distinct.

When we choose a train line siding on one cusp, this induces a train line siding on another cusp. This induced siding is given by Lemma 4.4, and an example of which is shown in Figure 32. The approach we take to consistently choosing the train line sidings is as follows, also described in Algorithm 4. For each cusp create a list of the edge classes in  $L(\mathcal{C}_i)$ . Then for each cusp, find train line sidings for all edge classes in the list on this cusp. Ensure each of the train line sidings lies in a cusp region which matches the bipartite matching. For each edge class we found a train line siding for, this edge class is associated to an edge of the triangulation, and there is a cusp at the other end of the edge. Find the induced train line siding on the other cusp and remove the edge class from the list of edge classes on the current cusp and the other cusp.

---

**Algorithm 4:** Train Lines

---

```
1 Triangulated 3-manifold manifold output: Train Line Sidings
2  $M \leftarrow \text{Matrix}(\text{num\_rows} = \text{manifold.num\_cusps}, \text{num\_cols} = \text{num\_edge\_classes});$ 
3 for each cusp  $C_i$  in the manifold do
4   for  $j \in \{1, \dots, \text{num\_edge\_classes}\}$  do
5     if  $j \in L(C_i)$  then
6        $M[i][j] \leftarrow \text{True}$ 
7     else
8        $M[i][j] \leftarrow \text{False}$ 
9     end
10  end
11 end
12 for each cusp  $C_i$  in the manifold do
13   for  $\text{edge\_class} \in \{1, \dots, \text{num\_edge\_classes}\}$  do
14     if  $M[i][\text{edge\_class}]$  is False then
15       continue;
16     end
17     Find a train line siding for  $\text{edge\_class}$  on  $C_i$ ;
18      $C_j \leftarrow$  index of the other cusp  $\text{edge\_class}$  lies in;
19     Find a train line siding for  $\text{edge\_class}$  on  $C_j$  compatible with the previous train line siding;
20      $M[i][\text{edge\_class}] \leftarrow \text{False}$ ;
21      $M[j][\text{edge\_class}] \leftarrow \text{False}$ ;
22   end
23 end
```

---

### 5.6.3 Curve Components

We build the train line  $\gamma$  incrementally, as a collection curves  $\gamma_1, \dots, \gamma_i$  such that  $\gamma_j$  connects  $\beta_j(1)$  and  $\beta_{j+1}(1)$ . The train line is the curves  $\gamma_i$  joined together, with sidings  $\beta_i$ . Firstly order the edge classes in  $L(C_i)$ ,  $(\alpha_1, \dots, \alpha_{|L(C_i)|})$ . Each edge class  $\alpha_j$  has an associated siding, constructed previously, which defines the **CuspRegion**,  $R(\alpha_j)$ , where the curves will terminate. We find curves between the first and second edge class, then second and third and so on.

**Proposition 5.11** (Train Line Curve Components). *Let  $\mathcal{C}$  be a cusp of a 3-manifold triangulation with boundary curves  $\mathbf{m}_k, \mathbf{l}_k$ . Let  $m = |L(\mathcal{C})|$  and  $\beta_1, \dots, \beta_m$  the collection of sidings on the cusp  $\mathcal{C}$  found previously. There exists a collection of curves  $\gamma_1, \dots, \gamma_{m-1} : [0, 1] \rightarrow \mathcal{C}$  such that,*

- i) Each  $\gamma_i$  is disjoint from  $\mathbf{m}_k$  and  $\mathbf{l}_k$ , and disjoint from  $\gamma_j((0, 1))$  for  $j \neq i$ ,*
- ii) For  $1 \leq i \leq m - 1$ ,  $\gamma_i$  is a curve running between distinct sides of each triangle it enters,*
- iii) For  $1 \leq i \leq m - 1$ ,  $\gamma_i(0) = \beta_i(1)$  and  $\gamma_i(1) = \beta_{i+1}(1)$ .*
- iv) For  $1 \leq i \leq m - 2$ , the final arc of  $\gamma_i$  and the first arc of  $\gamma_{i+1}$  lie on the same cusp triangle, and they enter this cusp triangle from distinct sides, with one of these sides the edge which  $\beta_{i+1}(0)$  is tangent to.*

*Proof.* We begin with  $\gamma_1$ . Construct the cusp region graph on the cusp  $\mathcal{C}$  for the curves  $\mathbf{m}_k$  and  $\mathbf{l}_k$ . Let  $R(\alpha_j)$  be the cusp region  $\beta_j$  is contained in. Let  $\alpha$  be an edge class, with  $\alpha \neq 1, 2$ .  $\alpha$  is not one of the edge classes we are finding a curve  $\gamma_1$  between, rather it is used to give the curve  $\gamma_1$  a certain property. In the Cusp Region graph, the cusp vertex corresponding to  $\alpha$  will induce a cycle consisting of the cusp regions around the cusp vertex. Let  $[r_1, \dots, r_a]$  be this cycle of cusp regions. Find a path  $\gamma$  through the Cusp Region graph from  $R(\alpha_1)$  to a region in  $\{r_1, \dots, r_a\}$ , and a path  $\gamma'$  from  $R(\alpha_2)$  to a region in  $\{r_1, \dots, r_a\}$ . The siding  $\beta_2$  runs tangent to an edge  $e$  at  $t = 0$ .



If  $\gamma'$  leaves  $R(\alpha_2)$  in the first step across edge  $e$ , set  $v_1$  to be the region adjacent across this edge and  $s_1$  to be the region  $R(\alpha_2)$ . Otherwise, set  $v_1$  to the region  $R(\alpha_2)$  and  $s_1$  to be the region adjacent to  $R(\alpha_2)$  across  $e$ . This edge  $(s_1, v_1)$  in the cusp region graph cannot be used in the next step of curve finding. There are two different paths  $\sigma_1, \sigma_2$  around the cycle  $[r_1, \dots, r_a]$  to join  $\gamma$  to  $\gamma'$ . Compute the cusp region graph with the curve  $\gamma + \sigma_1 + \gamma'$  and  $\gamma + \sigma_2 + \gamma'$ . Since the cusp region graph is path connected, after removing the edge  $(s_1, v_1)$  from each graph, we have at most 2 path connected components on each graph. One of the graphs has a connected component which contains both the cycle  $[r_1, \dots, r_m]$  and the cusp region  $s_1$ . Let  $\gamma_1$  be the curve  $\gamma + \sigma_j + \gamma'$  which induced this cusp region graph.

Now we proceed inductively on  $j < m-1$ . Construct the cusp region graph on  $\mathcal{C}_i$  for the curves  $\mathbf{m}_i, \mathbf{l}_i, \beta_1 + \gamma_1 + \dots + \beta_{j-1} + \gamma_{j-1} + \beta_j$ . We have an edge  $(s_{j-1}, v_{j-1})$  which we find while constructing  $\gamma_{j-1}$ , which is defined in the same way as  $(s_1, v_1)$ . Assume by induction, after removing  $(s_{j-1}, v_{j-1})$ , we have a connected subgraph containing  $s_{j-1}$  and a cycle  $[t_1, \dots, t_b]$ . Let  $\alpha'$  be an edge class on the cusp  $\mathcal{C}$ , with  $\alpha' \neq \alpha_1, \dots, \alpha_{j+1}$ . Then  $\alpha'$  does not have a curve which dives into the cusp vertex corresponding to this edge class, and hence we have a cycle  $[u_1, \dots, u_c]$  in the cusp region graph around this cusp vertex. In the same manner as  $\gamma_1$ , find a path  $\gamma_j$  through the cusp region graph, which goes around the cycle  $[u_1, \dots, u_c]$  in the direction which leaves a cycle in the subgraph of the cusp region graph containing  $s_j$  after removing  $(s_j, v_j)$ , from  $s_{j-1}$  to  $R_{j+1}$ . It could be the case that after removing the edge  $(s_{j-1}, v_{j-1})$ , we can't find a path from  $s_{j-1}$  to the target points. In this case, find a path from  $s_{j-1}$  to a point in the cycle from the previous iteration, pass around the cycle  $[t_1, \dots, t_m]$  and then back along the path to  $s_{j-1}$ . Then we can use the edge  $(s_{j-1}, v_{j-1})$  to cross to the other connected component of the cusp region graph.

The final curve  $\gamma_{m-1}$  can be found in the same manner as the inductive case, with the exception that we don't need to preserve a cycle in the subgraph contain  $s_{m-1}$  after removing the edge  $(s_{m-1}, v_{m-1})$  from the cusp region graph.  $\square$

Figure 23 shows how we incrementally construct the curve components. We start with the two sidings  $\beta_1$  and  $\beta_2$ , and we find a cycle consisting of  $[r_1, r_2, \dots, r_6]$ . The first curve component  $\gamma_1$  connects  $\beta_1$  to  $\beta_2$ , and passes the correct way around the cycle. Then we find a path from  $\beta_2$  to  $\beta_3$ , which is the curve  $\gamma_2$ , utilising this cycle to pass back along through the regions  $(v_1, s_1)$ . Without the cycle coming from  $[r_1, r_2, \dots, r_6]$  in the subgraph containing  $s_1$ , we would be stuck in the region enclosed by  $\gamma_1$  due to the condition that the curve must run between distinct sides of each cusp triangle it enters.

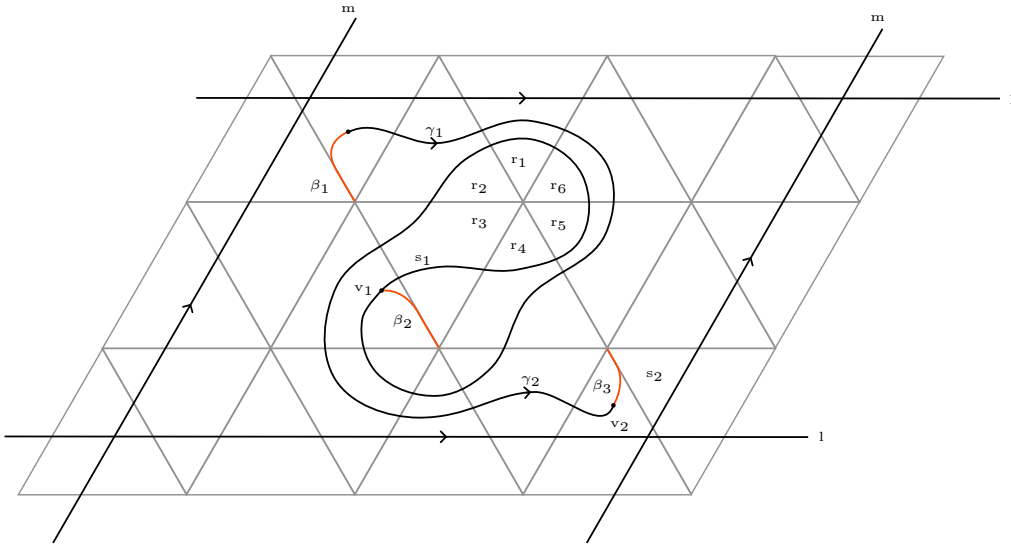


Figure 23: Two train line curve components  $\gamma_1, \gamma_2$  for the train line sidings  $\beta_1, \beta_2$  and  $\beta_3$ .

In each iteration we find two cusp regions  $s_j$  and  $v_j$ . The cusp region  $s_j$  is the start region, the next curve starts here and can go to any adjacent cusp region except  $v_j$ . The cusp region  $v_j$  is the visited region, in

order to preserve the condition that the final curve runs between distinct sides of each cusp triangle it enters, we cannot visit this cusp region in the first step. The choice of  $s$  and  $v$  also ensures condition *iv* holds. Each curve  $\gamma_i$  is disjoint from the other curves  $\gamma_j$  and the boundary curves by construction, since they are composed of paths in the cusp region graph. An example of how the curves  $\gamma_i, \gamma_{i+1}, \beta_{i+1}$  meet is shown in Section 5.8.3.

**Conjecture 5.12.** *Let  $\mathcal{C}$  be a cusp of a triangulated 3-manifold, which contains a train with sidings for a collection of edge class  $\{i_1, \dots, i_k\}$ . If  $\gamma, \gamma'$  are oriented curves on  $\mathcal{C}$  which start and end at cusp vertices corresponding to edge classes in  $\{i_1, \dots, i_k\}$  and lie on the train line, then*

$$\omega(h(\gamma), h(\gamma')) = 0$$

This conjecture is the key point to the train line. The cusp is initially path connected, so we can find the train line. Then oscillating curves which may disconnect the cusp can instead be made to run along the train line, and leave the cusp path connected. A proof of this conjecture is beyond the scope of this thesis. Instead, we justify the conjecture using the notion of an abstract oscillating curve, as defined in [MP22]. We call the vertex of the train track which lies on a long rectangle a station.

**Definition 5.13** (Abstract Oscillating Curve). *An abstract oscillating curve on an oriented enhanced train track  $\tau$  is a labeling of each edge  $\gamma$  of  $\tau$  by an integer  $n_\gamma$ , such that the following compatibility conditions are satisfied at each vertex  $v$ .*

i) *If  $v$  has degree  $k + 1$ , i.e.  $k + 1$  incident edges  $\gamma_0, \dots, \gamma_k$ , then*

$$\epsilon_{\gamma_0} n_{\gamma_0} + \dots + \epsilon_{\gamma_k} n_{\gamma_k} = 0$$

ii) *If  $v$  is a station with incident edges  $\gamma, \hat{\gamma}$  at one end and  $\delta = \bar{\gamma}, \hat{\delta}$  at the other end, then*

$$\epsilon_\gamma n_\gamma + \epsilon_{\hat{\gamma}} n_{\hat{\gamma}} = \epsilon_\delta n_\delta + \epsilon_{\hat{\delta}} n_{\hat{\delta}}$$

Abstract oscillating curves have the property that they only pick up combinatorial holonomy where they meet or diverge on the interior of cusp triangles. By considering the curves on a train line as an abstract oscillating curve, their intersection numbers will be 0, since when they diverge they dive along a short rectangle into the manifold, which does not pick up combinatorial holonomy. This also gives the reason for the additional edges to the sets  $K(\mathcal{C}_i)$ , since when we place the 'interior' components of an oscillating curves, we need to place the first and last components on the train line also. Section 7.2 illustrates the use of train lines with sidings, and the potential issues with them.

## 5.7 Oscillating Curves

We want to construct oscillating curves dual to the curve  $C_i$ , which corresponds to an edge  $E$  with `edge_class`  $i$  which the curve  $C_i$  circles. We construct an oscillating curve as a collection of curves which lie on cusps, called curve components. Each curve component start and end at cusp vertices. In order to orient the curve to be oscillating, we require the number of these components to be even. We look for a path of odd length, consisting of edges in  $\varepsilon_c$ , from the cusp containing one end of  $E$  to the cusp containing the other end of  $E$ . This gives a cycle of even length when we add the edge  $E$  to form a cycle. This path is found using Algorithm 5 as illustrated below.

While the algorithm finds the path through the end multi graph, it keeps track of the edge class of each edge it visits. To construct each oscillating curve, we need to extract the information about each curve component from this path, which we make explicit in the following example.

**Example 5.14.** Suppose we are constructing an oscillating curve  $\Gamma$  which is dual to edge class 1. The edge of the triangulation corresponding to edge class 1 lies in two cusps, say cusp 0 and cusp 3. We give this edge class, along with end multi graph and the collection of edges  $\varepsilon_c$  to Algorithm 5, and suppose it returns the path

---

**Algorithm 5:** End Multi Graph Path

---

**input** : An edge  $E$  with edge class  $i$ ,  $E_0$  the extra edge added to the spanning tree for the end multi graph.  
**output**: A path through the end multi graph of odd length

- 1 The ends of  $E$  lie at two cusps,  $m, n$ ;
- 2 Find a path through the spanning tree we picked for the end multi graph from  $m$  to  $n$ ;
- 3 **if** *path length is odd* **then**
- 4     return path;
- 5 **else**
- 6      $usp_0 \leftarrow$  cusp index of one end of  $E_0$ ;
- 7      $usp_1 \leftarrow$  cusp index of the other end of  $E_0$ ;
- 8      $\gamma_0 \leftarrow$  path from  $m$  to  $usp_0$ ;
- 9      $\gamma_1 \leftarrow$  path consisting of a single edge from  $usp_0$  to  $usp_1$ ;
- 10     $\gamma_2 \leftarrow$  path from  $usp_1$  to  $n$ ;
- 11    return  $\gamma_0 + \gamma_1 + \gamma_2$ ;
- 12 **end**

---

$$\text{Cusp } 0 \xrightarrow{E_2} \text{Cusp } 2 \xrightarrow{E_0} \text{Cusp } 2 \xrightarrow{E_4} \text{Cusp } 3$$

So we have 4 components to this oscillating curve, oriented such that the curves reverse orientation after diving through the manifold.

- i) Curve 1: Lies on Cusp 0, starts at edge class 1 and ends at edge class 2.
- ii) Curve 2: Lies on Cusp 2, starts at edge class 0 and ends at edge class 2.
- iii) Curve 3: Lies on Cusp 2, starts at edge class 0 and ends at edge class 4.
- iv) Curve 4: Lies on Cusp 3, starts at edge class 1 and ends at edge class 4.

We proceed with the general construction of components of oscillating curves. If the curve component starts and ends at edge class in the train line with sidings on the cusp, found previously, the curve lies completely on the train line. To find the other curve components, we start by deciding how the curve will dive into the manifold at either end. We look for a cusp region which allows us to dive along a cusp edge into the cusp vertex corresponding to the target edge class. The subtle detail is we need to dive in the way described in Lemma 4.4 at either end of the edge of the triangulation corresponding to the edge class. So the first curve component of the current curve we are constructing to use the edge picks the first cusp region which allows us to dive along a cusp edge into a cusp vertex corresponding to the target edge class. For the next curve component to use the edge, we pick the cusp region which matches the curve chosen previously for this edge class. For each curve component, we use breadth first search on the cusp region graph to find a path between the cusp regions at the start and end of the curve component.

## 5.8 Cusp Region Splitting

At any point in the algorithm, on a cusp of the manifold, we will have a number of cusp regions coming from the curves  $\gamma_1, \dots, \gamma_i$  currently lying on the cusp. Suppose we find a new curve  $\gamma_{i+1}$  on the cusp, to reconstruct the cusp region graph we need to find the cusp regions on the cusp for curves  $\gamma_1, \dots, \gamma_{i+1}$ . Most of the cusp regions information is still useful, so we don't throw away the previously computed cusp regions. Instead, we 'split' each cusp region the curve passes through into two cusp regions, and then update the other cusp regions on the relevant cusp triangles to account for this new curve. This gives a number of cases to consider.

### 5.8.1 Interior Arcs

The simplest case is an arc of a curve which meets distinct sides of the cusp triangle. For oscillating curves, these are the arcs of the curve on the cusp except the first and last one which dive into the manifold. Figure 24 shows an example which contains two arcs on the left. Then we add an arc on cusp region  $r_0$  which splits this region into two new regions  $r_3$  and  $r_4$ . In Section 6, we will describe explicitly how we represent cusp regions as objects in  $\mathcal{C}$ , and compute an example of region splitting for an arc which meets distinct sides of the cusp triangle in Example 6.2.

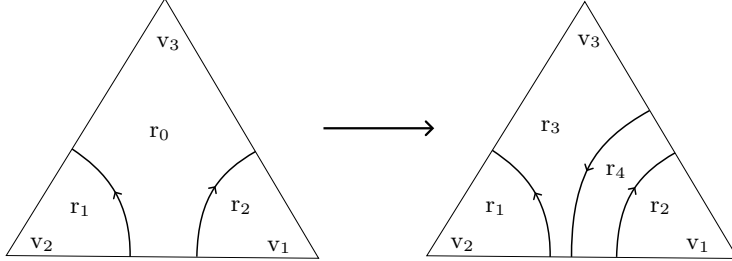


Figure 24: Adding an arc on cusp region  $r_0$  which splits it into  $r_3$  and  $r_4$ .

### 5.8.2 Cusp Vertex Arcs

Arcs of an oscillating curve which meet cusp vertices need to be handled differently to arcs which run between distinct sides of the cusp triangle. After crossing an edge of a cusp triangle, there are essentially 3 different ways the triangle can dive into a vertex of the cusp triangle. Either by 'passing around a vertex' and along a face opposite to the one crossed (left of Figure 25), by diving into the vertex opposite the edge which the curve crossed (middle of Figure 25), or finally by diving along the edge the curve crossed (right of Figure 25). For each of these cases we have different cusp regions obtained after splitting along the arc.

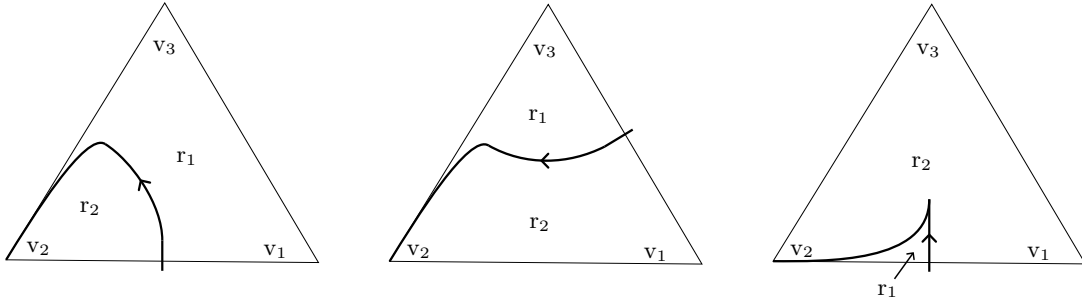


Figure 25: Splitting of an endpoint cusp region.

### 5.8.3 Train Line Arcs

The final type of splitting occurs while constructing the train line with sidings. In Section 5.6, we described the construction of the train lines with sidings by first choosing a set of sidings  $\beta_1, \dots, \beta_{i+1}$ , and then incrementally finding curves  $\gamma_1, \dots, \gamma_i$  which make up the train line. When we construct the train line, in each iteration we need to reconstruct the cusp region graph, so we split along the train line as follows.

- In the first iteration we only have the curves  $\mathbf{m}, \mathbf{l}$ , so we construct the cusp region as described in Section 5.2.
- After finding the curve  $\gamma_1$ , we can split along the curve  $\beta_1 + \gamma_1 + \beta_2$  using the process described above (Section 5.8.1 and 5.8.2).

- For the remaining  $\gamma_i$ , we split along the curve  $\gamma_i + \beta_{i+1}$  using the process described above, except for the first arc of the curve. In this case we have 3 curves meeting at a point, so we need a separate process which we describe below.

There are 4 different ways the train lines can meet at  $\beta(1)$ . Figure 26 shows the two cases for joining to a train line where the previous segment dived through the manifold along the last edge it crossed. Figure 27 shows the remaining two cases, where the previous segment has dived through the manifold by either 'passing around the vertex' or crossed the face opposite the vertex it dives through.

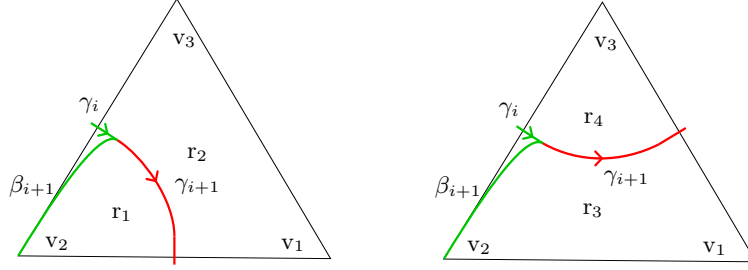


Figure 26: Join of two train line segments. Previous curve  $\gamma_i + \beta_{i+1}$  shown in green, next curve  $\gamma_{i+1}$  shown in red.

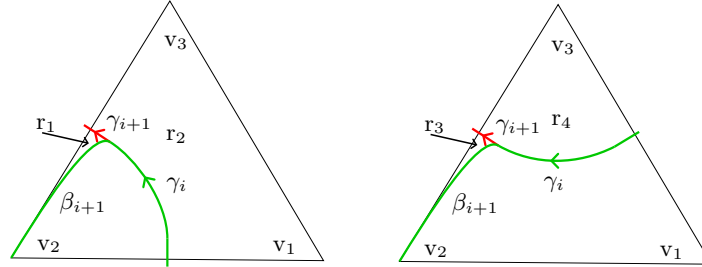


Figure 27: Join of two train line segments. Previous curve  $\gamma_i + \beta_{i+1}$  shown in green, next curve  $\gamma_{i+1}$  shown in red.

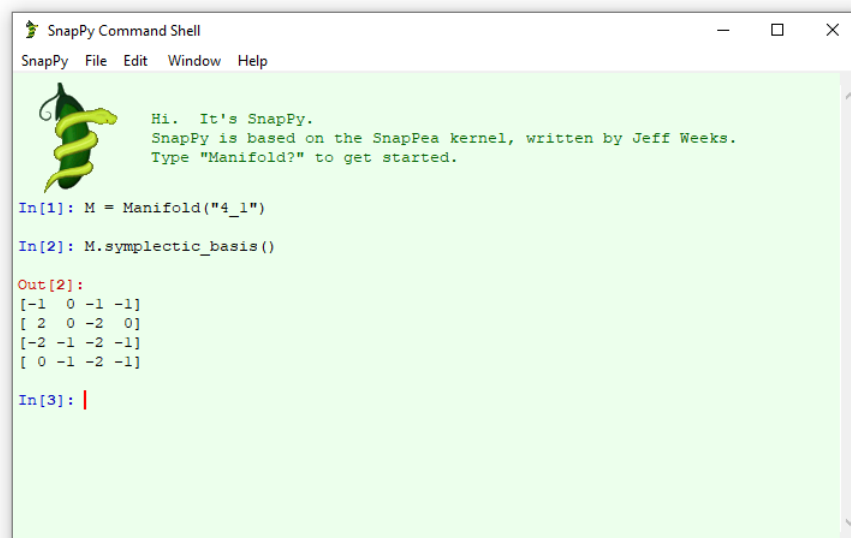
## 5.9 Neumann-Zagier Matrix

All the work done previously makes calculating the Neumann-Zagier matrix relatively straightforward. SnapPy currently contains functions to combinatorial holonomy of the curves  $C_i$ ,  $m_j$  and  $l_j$ . We store the intersection numbers of the oscillating curves with each cusp edge, this is identical to how SnapPy stores the curves  $m_j$  and  $l_j$ . So to find the holonomy of the oscillating curves, we simply use the same algorithm as SnapPy, with a change of arrays to the ones containing the oscillating curves. To find the holonomy corresponding to the curves  $C_i$ , SnapPy walks around the edge class corresponding to  $C_i$  and adds up the basis vectors  $\mathbf{a}_i$ ,  $\mathbf{b}_i$  or  $\mathbf{c}_i$  on the inside cusp vertex. This function is contained in `kernel/addl_code/get_gluings_equations.c` in the function `get_gluings_equations()`. To find the vectors corresponding to the other curves, SnapPy looks at every cusp triangle, adding up the contribution from the arcs of the curve lying on the cusp triangle. The function to do this for  $m$  and  $l$  is contained in `kernel/addl_code/get_gluings_equations.c` in the function `get_cusp_equation()`.

We have implemented the algorithm described above in C, within the SnapPy kernel. The algorithm has been tested on the knots in the SnapPy `HTLinkExteriors` database, which contains 313,230 knots, with all manifolds giving matrices which are symplectic (up to factors of 2). The algorithm has been merged into the SnapPy source code [Cul+23], with the main algorithm contained in

kernel/kernel\_code/symplectic\_basis.c. We add the method `symplectic_basis()` for `Manifolds` and `Triangulations`, which will be available within SnapPy package after the next release. Figure 28 gives an example of how to use the function, with the output of the form

$$\left[ h(\mathbf{m}_1), h(\mathbf{l}_1), \dots, h(\mathbf{m}_g), h(\mathbf{m}_g), h(C_1), h(\Gamma_1), \dots, h(C_{n-n_c}), h(\Gamma_{n-n_c}) \right]$$



```

SnapPy Command Shell
SnapPy File Edit Window Help

Hi. It's SnapPy.
SnapPy is based on the SnapPea kernel, written by Jeff Weeks.
Type "Manifold?" to get started.

In[1]: M = Manifold("4_1")
In[2]: M.symplectic_basis()

Out[2]:
[[-1  0 -1 -1]
 [ 2  0 -2  0]
 [-2 -1 -2 -1]
 [ 0 -1 -2 -1]]

In[3]: |

```

Figure 28: Symplectic Basis for the Figure-8 knot using SnapPy.

For link complements, we have a partial implementation, which is missing the full train line functionality. Although the current implementation can find train lines on most link complements, on some it finds matrices which are not symplectic (up to factors of 2). An example of this is described in Section 7.2.

## 6 Implementation Details

In the previous section we outlined an algorithm for constructing oscillating curves dual to edge curves on a triangulated 3-manifold. For clarity, numerous details about the structures used were omitted, which we detail here. SnapPy constructs the `Triangulation`, `Tetrahedron`, `EdgeClass`, and `Cusp` objects as it triangulates a 3-manifold. The structs

- `CuspTriangle` and `CuspVertex` (Section 6.1),
- `CuspRegion` (Section 6.2),
- `EndMultiGraph` and `CuspNode` (Section 6.4),
- `PathNode` and `PathEndPoint` (Section 6.5),
- `CuspStructure` (Section 6.6),
- `CurveComponent` and `OscillatingCurve` (Section 6.8)

we construct as part of the algorithm, and are new to this paper. We also use `EdgeNode` and `Graph` structs (Section 6.3) for various different graphs, which is a well established data structure. SnapPy has a large database of knot and link exterior triangulations, and can triangulate a knot drawn by the user using the PLink library. These triangulations are stored as the `Triangulation` structure. The `triangulation.h`

header file contains the type declarations of **Triangulation**, **EdgeClass**, and **Cusp**, along with comments for their interpretation [Cul+23].

```

struct Triangulation {
    char                *name;
    int                 num_tetrahedra;
    ...
    int                 num_cusps;
    ...
    Tetrahedron         tet_list_begin, tet_list_end;
    EdgeClass            edge_list_begin, edge_list_end;
    Cusp                 cusp_list_begin, cusp_list_end;
};

```

The **Triangulation** struct contains all the triangulation information for a triangulated 3-manifold. The tetrahedra of the triangulation are stored as a doubly linked list of **Tetrahedron** structs in **tet\_list\_begin**. Similarly, for **edge\_list\_begin**, which contains information about the edges of the triangulation and **cusp\_list\_begin**, which contains information about each cusp of the manifold.

```

struct Tetrahedron {
    Tetrahedron         *neighbor[4];
    Permutation          gluing[4];
    Cusp                 *cusp[4];
    int                 curve[2][2][4][4];
    EdgeClass            *edge_class[6];
    ...
};

```

Each vertex of a **Tetrahedron** is given a label  $\{0, 1, 2, 3\}$ , which gives a natural labelling of the faces (the face opposite a vertex). The attribute **neighbour**[*i*] is a pointer to the tetrahedron which glues to face *i*. The attribute **gluing**[*i*] is a permutation of  $\{0, 1, 2, 3\}$ , which describes the gluing of the tetrahedra using the following scheme. Let  $\sigma \in \text{Sym}(\{0, 1, 2, 3\})$  be the *i*-th entry in the **gluing** array. Then  $\sigma(i)$  is the face of **neighbour**[*i*] which glues to face *i*. Additionally, for  $k \in \{0, 1, 2, 3\}$  and  $k \neq i$  then  $\sigma(k)$  is a vertex on the tetrahedron **neighbour**[*i*] which is glues to vertex *k* when the face *i* on the tetrahedron is glued to face  $\sigma(i)$  on **neighbour**[*i*]. The **gluing** permutations are used to calculate which cusp triangle is adjacent across an edge of a cusp triangle. The attribute **curve**[*M*][0][*v*][*e*] is a 4 dimensional array consisting of the net number of times the meridian crosses edge *e* of the cusp triangle at vertex *v*. The constant *M* refers to the meridian curve, which can be changed to *L* for the longitude curve. SnapPy substitutes these constants with 0 and 1 to index into the curve array. In order to handle the case of non orientable cusps, SnapPy considers two cusp triangles at each vertex to place the boundary curves on. The second dimension of the array, which is set to 0, is used to access the two cusp triangles. For our purposes, we are only considering tori boundary components, for which we only need one cusp triangle.

```

struct Cusp {
    int                 index;
    ...
}

```

The **cusp**[*i*] attribute of a tetrahedron is a pointer to the **Cusp** struct which the cusp triangle at vertex *i* of the tetrahedron lies in. For each boundary component of the manifold, SnapPy creates a **Cusp** struct which stores information about the cusp. We use the **index** attribute of the cusp struct to identify which boundary components each cusp triangle belongs to.

```

struct EdgeClass {
    int                 index;
    ...
}

```

The `edge_class[i]` attribute of a tetrahedron is a pointer to the `EdgeClass` struct corresponding to the edge of triangulation on edge  $i$  of the tetrahedron. The `index` attribute is the edge class, from Definition 2.10, of the edge. SnapPy labels the edges of a tetrahedron  $0, \dots, 5$  by which vertices they lie between. The following comment is contained in `edge_classes.c` in the SnapPy kernel [Cul+23].

```

/*
 * The edges of a tetrahedron are indexed
 * according to the following table:
 *
 *
 *      edge      lies      lies
 *              between between
 *              faces  vertices
 *
 *      0          0,1      2,3
 *      1          0,2      1,3
 *      2          0,3      1,2
 *      3          1,2      0,3
 *      4          1,3      0,2
 *      5          2,3      0,1
 *
 */

```

## 6.1 Cusp Triangulation

The first data structure we construct is a doubly linked list of cusp triangles, which stores information about each cusp triangle. All the information comes from the `Triangulation` data. Each tetrahedron contributes 4 cusp triangles. Each edge of a cusp triangle lies in a face of the tetrahedron which it comes from, so we have a labelling of the cusp edges by the faces of the tetrahedron. This labelling gives a labelling of the vertices of the cusp triangle, by the edge opposite the vertex.

```

typedef struct CuspTriangle {
    Tetrahedron      *tet;
    int              tet_index;
    int              tet_vertex;
    CuspVertex       vertices[4];
    struct CuspTriangle *neighbours[4];
    struct CuspTriangle *next;
    struct CuspTriangle *prev;
} CuspTriangle;

```

The attribute `tet` is a pointer to the tetrahedron the cusp triangle comes from. The attribute `tet_index` is `tet->index`, so the index is easier to access. The attribute `tet_vertex` is the vertex of the tetrahedron cut off by the cusp triangle. The attribute `vertices[v]` is the `CuspVertex` vertex  $v$ , with `vertices[tet_vertex]` set to `NULL`. The attribute `neighbours[e]` is a pointer to the `CuspTriangle` adjacent across edge  $e$ , with `neighbours[tet_vertex]` set to `NULL`. The attributes `next` and `prev` are used as part of the linked list data structure.

```

typedef struct CuspVertex {
    int      edge_class;
    int      edge_index;
    EdgeClass *edge;
    int      v1;
    int      v2;
} CuspVertex;

```

Each vertex of a cusp triangle is given a `CuspVertex` object. Note however, that we have multiple, different `CuspVertex` objects for the same *vertex* in the cusp triangulation. The main purpose of the `CuspVertex` is



the keep track of the edge class and edge index of each cusp vertex. Each cusp vertex lies on an edge, `edge`, of a tetrahedron which in turn lies on an edge of the triangulation. The edge connects two vertices of the tetrahedron, `v1` and `v2`. For `tri->vertex[v]` on a cusp triangle, we set `v1 = tri->tet_vertex` and `v2 = v`. The attribute `edge` stores `tet->edge_class[i]` which  $i$  is the index of the edge lying between `v1` and `v2`. This edge belongs to an edge class, `edge_class`, as described in Definition 2.10. If the ends of an edge lie in the same cusp, then `edge_index` is used to distinguish these two ends in the cusp triangulation.

## 6.2 Cusp Region

The Cusp Regions are connected components of a torus cusp minus the edges of the triangulation, the boundary curves, and any curves constructed on the cusp such as train lines or oscillating curves, as defined previously in Definition 5.1. One of the significant runtime costs of the algorithm is updating and finding cusp regions, so to reduce this cost we store all the cusp regions on a given cusp triangle in a doubly linked list, keeping a linked list for each cusp triangle. Depending on the triangulation and the number of cusps, the number of cusp regions can explode into the hundreds of thousands with only 20 oscillating curves. When running the algorithm on a test set of 60,000 knots and links, this optimisation cut the run time from weeks down to hours. We define a cusp region as follows.

```
typedef struct CuspRegion {
    CuspTriangle      *tri;
    int               tet_index;
    int               tet_vertex;
    int               index;
    bool              adj_cusp_triangle[4];
    int               curve[4][4];
    bool              dive[4][4];
    int               num_adj_curves[4][4];
    int               temp_adj_curves[4][4];
    struct CuspRegion *adj_cusp_regions[4];
    struct CuspRegion *next;
    struct CuspRegion *prev;
} CuspRegion;
```

Each cusp region lies in a single cusp triangle, since we remove the edges of the triangulation in the cusp region definition. The attribute `tri` stores a pointer to the cusp triangle the region lies in. The attributes `tet_index` and `tet_vertex` come from the `CuspTriangle`, `tet_index = tri->tet_index` and `tet_vertex = tri->tet_vertex`. The attribute `adj_cusp_triangle[e]` is `True` if the cusp region meets edge  $e$  of the cusp triangle and `False` otherwise. If the cusp region meets edge  $e$  of the cusp triangle it lies in, then the attribute `curve[e][v]` is the number of curves between the cusp region and vertex  $v$  along the edge  $e$ , otherwise it is set to 0. The attribute `dive[e][v]` is `True` if there exists a curve in this `CuspRegion` which can dive along cusp edge  $e$  into cusp vertex  $v$ , otherwise `dive[e][v]` is `False`. The attributes `num_adj_curves` and `temp_adj_curves` are used to keep track of the number of curves between the cusp region and an edge, `num_adj_curves[e][v]` is the number of curves which dive along cusp edge  $e$  into cusp vertex  $v$  between the cusp region and cusp edge  $e$ . The attribute `adj_cusp_regions[e]` is a pointer to the cusp region adjacent across edge  $e$ , or `NULL` if this region does not exist. We are only adjacent to at most 3 regions across edges of the cusp triangle since the oscillating curves pass between distinct sides of each cusp triangle (Lemma 4.6).

**Example 6.1.** Consider the cusp region indicated in green on Figure 29.

We will initialise cusp region, `region`, explicitly. This region lies in some cusp triangle `tri`, so we set `region.tri = tri`. We transfer some attributes from the cusp triangle up to the cusp region, `region.tet_index = tri.tet_index` and `region.tet_vertex = tri.tet_vertex`. We keep track of the number of cusp regions we have currently initialised,  $i$ , set `region.index = i` and increment  $i$  by 1. Since the cusp region lies between two curves passing around vertex 2,

```
region.adj_cusp_triangle = [false, true, false, true]
```

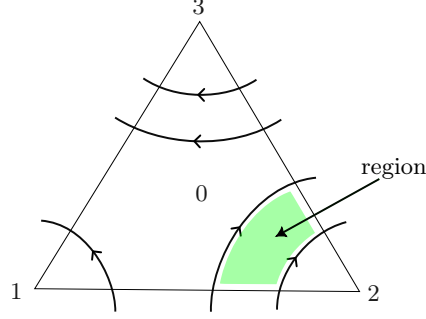


Figure 29: Cusp Region.

We define a function **Flow** which takes 2 cusp triangle edges and returns the number of curves between these two edges. So  $\text{Flow}(1, 3) = 2$ , there are 2 curves passing between edges 1 and 3. The target cusp region has 1 curve between the region and vertex 2. So we find the number of curves towards vertex 2,

$$\text{region.curve}[1][2] = 1, \quad \text{region.curve}[3][2] = 1.$$

Next we find the number of curves along edge 1 towards vertex 3. The flow between edges 1 and 2,  $\text{Flow}(1, 2)$ , is 2, so

$$\text{region.curve}[1][3] = \text{Flow}(1, 3) + \text{Flow}(1, 2) - 1 = 3$$

Finally, the number of curves along edge 3 towards vertex 1. The flow between edges 2 and 3,  $\text{Flow}(2, 3)$ , is 1, so

$$\text{region.curve}[3][1] = \text{Flow}(1, 3) + \text{Flow}(1, 2) - 1 = 2$$

So we have the complete curve matrix,

```
region.curve = {
    {0, 0, 0, 0}, {0, 0, 1, 3}, {0, 0, 0, 0}, {0, 2, 1, 0}
};
```

This region cannot dive along any face towards a vertex so the entire dive matrix is **False**.

```
region.dive = {
    {False, False, False, False},
    {False, False, False, False},
    {False, False, False, False},
    {False, False, False, False}
};
```

The arrays **num\_adj\_curves** and **temp\_adj\_curves** initialised the arrays to 0. Note since we cannot dive into any vertex of this cusp triangle, we will not use these arrays for this particular cusp region. The array **adj\_cusp\_regions** cannot be found until we have all the cusp regions, so we leave this array uninitialised.

**Example 6.2.** Consider the cusp triangle shown in Figure 30.

For simplicity, we will start with no curves on the cusp triangle, so we have one cusp region, **region**. Suppose the cusp region lies in the cusp triangle **tri**, which has **tet\_vertex** = 0, with adjacent cusp regions across face 1, 2 and 3, **r1**, **r2** and **r3** respectively.

```
region.tri = tri;
region.tet_index = tet_index;
region.tet_vertex = 0;
region.index = index;
region.curve = {
```

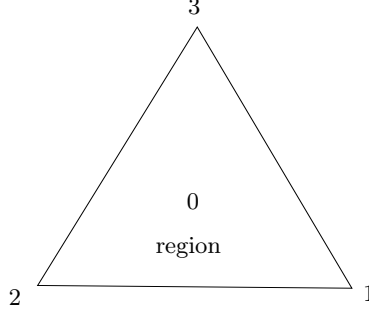


Figure 30: Cusp Region.

```

    {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}
};
region.adj_cusp_triangle = {false, true, true, true};
region.dive = {
    {False, False, False, False},
    {False, False, True, True},
    {False, True, False, True},
    {False, True, True, False}
};
region.num_adj_curves = {
    {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}
};
region.adj_cusp_regions = {NULL, r1, r2, r3};

```

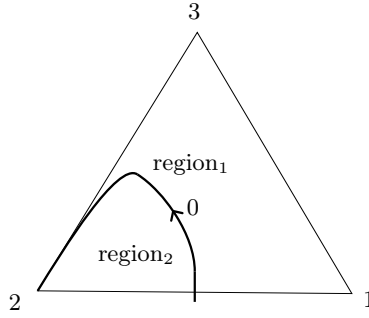


Figure 31: Cusp Region.

Now we draw the curve show on Figure 31. Then we need to create two new cusp regions **region1**, **region2**. Because the curve crosses edge 3, **region1** sees an extra curve on edge 3 towards vertex 2, and **region2** sees an extra curve on edge 3 towards vertex 1.

$$\begin{aligned} \text{region1.curve}[3][2] &= \text{region.curve}[3][2] + 1 = 1 \\ \text{region2.curve}[3][1] &= \text{region.curve}[3][1] + 1 = 1 \end{aligned}$$

Since the curve runs around the vertex 2, and dives through the manifold along face 1, **region2** is only adjacent to face 3. The curve also contributes to the array **num\_adj\_curves**, since the curve dives along face 1 into vertex 2, we set

$$\text{region2.num\_adj\_curves}[1][2] = \text{region.num\_adj\_curves}[1][2] + 1 = 1$$

This curve will split the region **r3** into two regions **r3'** and **r3''**. In the algorithm, these are not calculated until after all regions have been split.

```

region1.tri = tri;
region1.tet_index = tet_index;
region1.tet_vertex = 0;
region1.index = num_cusp_regions + 1;
region1.curve = {
    {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 1, 0}
};
region1.adj_cusp_triangle = {False, True, True, True};
region1.dive = {
    {False, False, False, False},
    {False, False, True, True},
    {False, True, False, True},
    {False, True, False, False}
};
region1.num_adj_curves = {
    {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}
};
region1.adj_cusp_regions = {NULL, r1, r2, r3''};

region2.tri = tri;
region2.tet_index = tet_index;
region2.tet_vertex = 0;
region2.index = num_cusp_regions + 2;
region2.curve = {
    {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 1, 0, 0}
};
region2.adj_cusp_triangle = {False, False, False, True};
region2.dive = {
    {False, False, False, False},
    {False, False, True, False},
    {False, False, False, False},
    {False, False, True, False}
};
region2.num_adj_curves = {
    {0, 0, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}
};
region2.adj_cusp_regions = {NULL, NULL, NULL, r3'};

```

### 6.3 Cusp Region Graph

We have a local view of the cusp regions, coming from the **adj\_cusp\_regions** array of each cusp region. The purpose of the Cusp Region graph is to construct a global view of the cusp regions, which allows us to use breadth first search for path finding. Each cusp region is given a vertex in the graph, labelled by the **index** attribute of the cusp region. For each cusp region, we insert an edge to each of the adjacent regions into the Cusp Region graph. The graph is undirected and is stored using an adjacency list of **EdgeNodes**.

```

typedef struct EdgeNode {
    int y;
    struct EdgeNode *next;
    struct EdgeNode *prev;
} EdgeNode;

```

The attribute `y` is a vertex of the graph. The attributes `next` and `prev` are used as part of the doubly linked list data structure.

```
typedef struct Graph {
    EdgeNode      *edge_list_begin;
    EdgeNode      *edge_list_end;
    int           *degree;
    int           *color;
    int           num_vertices;
} Graph;
```

The attributes `edge_list_begin` and `edge_list_end` store the adjacency list of edges as follows. An `EdgeNode` `edge` in the doubly linked list `edge_list_begin[v]` represents an edge of the graph (`v`, `edge->y`). The attribute `degree[v]` is the degree of the vertex `v`. The attribute `color` is an array used for bipartite coloring of the end multi graph. The attribute `num_vertices` is the number of vertices in the graph.

## 6.4 End Multi Graph

The end multi graph consists of a graph where the vertices are cusps of the manifold and edges are edges of the triangulation. The cusps are indexed by SnapPy, so we reuse this indexing for the graph structure. The edges of the triangulation are also indexed by the edge class, and we keep track of the edge class associated to each edge in the end multi graph since we may have multiple edges between a pair of cusps.

```
typedef struct EndMultiGraph {
    int           e0;
    int           num_edge_classes;
    int           num_cusps;
    int           **edges;
    Boolean       *edge_classes;
    Graph         *multi_graph;
} EndMultiGraph;
```

The attribute `e0` is the edge class we have chosen to be  $E_0$ . The attribute `multi_graph` is a pointer to a spanning tree for the end multi graph. The attribute `edges[u][v]` is the edge class of the edge  $(u, v)$  in the spanning tree for the end multi graph, where  $u$  and  $v$  are cusps of the manifold. The attribute `edge_classes[edge_class]` is `True` if `edge_class` is in the set of edges  $\varepsilon_c$ , the edges in the spanning tree for the end multi graph and  $E_0$ , and is `False` otherwise.

When we construct an oscillating curve, we need a cycle of even length through the end multi graph. We firstly find a path of odd length using graph in `multi_graph` plus the edge `e0`, which gives us the path as a sequence of cusp indices (using `EdgeNodes`). To construct oscillating curves dual to the curves  $C_i$ , we need some more information. We convert the sequence of `EdgeNodes` into a sequence of `CuspNodes`, a process we described in Example 5.14.

```
typedef struct CuspNode {
    int           cusp_index;
    int           edge_class[2];
    struct CuspNode *next;
    struct CuspNode *prev;
} CuspNode;
```

A `CuspNode` represents a template for a single curve component of an oscillating curve. We use the `edges` matrix to find the edge class associated to each edge in the path of `PathNodes`. The attribute `edge_class[START]` is the edge class the curve starts at, with `edge_class[FINISH]` is the edge class the curve finishes at. The attribute `cusp_index` is the index of the cusp the node lies in. The attributes `next` and `prev` are pointers to the next and previous `CuspNodes` for the doubly linked list. The start and finish edge classes are set such that when each curve component is oriented from start to finish, we obtain an oscillating curve.

## 6.5 Path Nodes and End Points

We represent curves on a cusp using **PathNodes** and **PathEndPoints**. The oscillating curves and train lines with sidings are made up of curves which begin and end cusp vertices, and pass through a number of cusp regions. The cusp regions the curve passes through are stored as a doubly linked list of **PathNodes**.

```
typedef struct PathNode {
    int                cusp_region_index;
    int                next_edge;
    int                prev_edge;
    int                inside_vertex;
    struct CuspTriangle *tri;
    struct PathNode    *next;
    struct PathNode    *prev;
} PathNode;
```

The **cusp\_region\_index** attribute is the index of the **CuspRegion** the **PathNode** is contained in. Each curve is oriented, **next\_face** (resp. **prev\_face**) is the edge of the cusp triangle the exits (resp. enters) the cusp triangle across. The **PathNode** can be considered as an arc of the curve which is contained in a cusp triangle, and this arc cuts off a vertex **inside\_vertex** of the cusp triangle. The **tri** attribute is a pointer to the **CuspTriangle** the **PathNode** lies in. The **next** and **prev** attributes are used for the doubly linked list.

The **PathNodes** give the path from starting cusp region to the final one. Consider a **PathNode** as a point on a cusp region, a **PathEndPoint** represent a curve from the **PathNode** to a cusp vertex. Each cusp region keeps track of whether or not we can dive into a vertex of the cusp triangle along a given edge. We illustrate how a curve dives through the manifold along an edge of a tetrahedron. The curve lies on the cusp triangle of vertex  $v_1$ , passes along the edge  $F$  of the top cusp triangle, down the vertex  $v_2$  and then enters the lower cusp triangle along the face  $F$ . Figure 32 shows this oscillating curve as viewed on a slice of a tetrahedron (left) and the corresponding curve on the cusp triangulation (right).

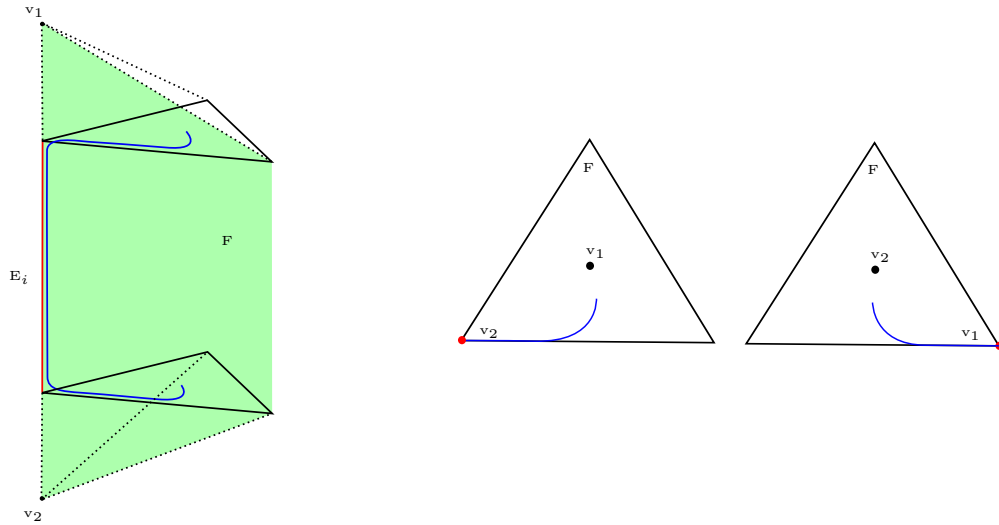


Figure 32: Oscillating curve on a slice of a tetrahedron (left), and the oscillating curve on the cusp triangles at  $v_1$  and  $v_2$  (right).

```
typedef struct PathEndPoint {
    int                edge;
    int                vertex;
}
```

```

    int                cusp_region_index;
    int                num_adj_curves;
    struct PathNode    *node;
    struct CuspRegion  *region;
    struct CuspTriangle *tri;
} PathEndPoint;

```

The **edge** attribute is the cusp edge the curve runs adjacent to. The **vertex** attribute is the cusp vertex which the curve enters. The **cusp\_region\_index** attribute is the index of the **CuspRegion** the curve is contained in. The **num\_adj\_curves** attribute is the number of curves which run through the cusp triangle containing the **PathEndPoint**, between the **PathEndPoint** curve and cusp edge **edge**, into the cusp vertex **vertex**. The **node** attribute is a pointer to a **PathNode** which this **PathEndPoint** joins up to. The **region** and **tri** attributes are pointers to the cusp region and cusp triangle the path end point lies in.

In the process of constructing an oscillating curve, we find a path through the cusp region graph, which gives a sequence of **EdgeNodes**. We convert this path to a path of **PathNodes**. To find the next curve on the cusp, we add this curve to the curves used to construct the cusp regions. Now the **PathNodes** and **PathEndPoints** are not contained in a cusp region, so the **region** pointers are no longer needed. In the end, we want to find the combinatorial holonomy of each arc, and the **next\_edge**, **prev\_edge**, **tri** attributes are sufficient to find the holonomy.

## 6.6 Cusp Structure

The **CuspStructure** struct is a collection of the various objects contained on a cusp of the manifold. The program uses an array of **manifold->num\_cusps()** **CuspStructure** structs, to keep track of the entire manifold.

```

typedef struct CuspStructure {
    int                intersect_tet_index;
    int                intersect_tet_vertex;
    int                num_edge_classes;
    int                num_cusp_triangles;
    int                num_cusp_regions;
    Triangulation      *manifold;
    Cusp               *cusp;
    Graph              *cusp_region_graph;
    CuspTriangle       cusp_triangle_begin;
    CuspTriangle       cusp_triangle_end;
    CuspRegion         *cusp_region_begin;
    CuspRegion         *cusp_region_end;
    PathNode           train_line_path_begin;
    PathNode           train_line_path_end;
    PathEndPoint       *train_line_endpoint[2];
} CuspStructure;

```

The attributes **intersect\_tet\_index** and **intersect\_tet\_vertex** refer to the cusp triangle at vertex **intersect\_tet\_vertex** of the tetrahedron with index **intersect\_tet\_index** which **peripheral\_curves()** chose to place the intersection of the boundary curves on.

## 6.7 Train Lines

The train line is a path on each cusp which interconnects a certain collection of edge classes on the cusp. The train line is stored as a doubly linked list of **PathNodes**, and two arrays of **PathEndPoints**. The cusp triangles the curve passes through is stored by the **PathNodes** and how the curve dives into the manifold is stored in the **PathEndPoints**. The **train\_line\_endpoint[edge\_index][edge\_class]** attribute in **CuspStructure** is the **PathEndPoint** associated to edge index **edge\_index** and edge class **edge\_class**. For **PathEndPoints** not

in use, we set the `tri` attribute to `NULL`. The curves which are placed on the train line change the orientation of each curve of the train line it uses in order to have the desired orientation.

## 6.8 Oscillating Curves

Each oscillating curve is broken up into `CurveComponents`, with each component consisting of a section of an oscillating curve lying on a single cusp.

```
typedef struct CurveComponent{
    int                edge_class[2];
    int                cusp_index;
    PathNode           path_begin;
    PathNode           path_end;
    PathEndPoint       endpoints[2];
    struct CurveComponent *next;
    struct CurveComponent *prev;
} CurveComponent;
```

Each curve component dives into the manifold at either end, and the edge class it dives into is stored in `edge_class[START]` and `edge_class[FINISH]`, with the curve oriented to run from `START` to `FINISH`. The `cusp_index` attribute stores the `cusp->index` attribute from the `Cusp` the curve lies in. The path of the curve is stored as a doubly linked list of `PathNodes`, with the header in `path_begin`. The attribute `endpoints` stores a `PathEndPoint` for the start and end of the curve. The attributes `next` and `prev` are used in the doubly linked list data structure.

The `OscillatingCurve` struct stores all the oscillating curves on the manifold. The number of oscillating curves is the number of tetrahedra in the triangulation minus the number of cusps.

```
typedef struct OscillatingCurves {
    int                num_curves;
    int                *edge_class;
    CurveComponent     *curve_begin;
    CurveComponent     *curve_end;
} OscillatingCurves;
```

Each oscillating curve intersects an edge curve  $C$  which encircles a cusp vertex with an edge class, which we associate to this oscillating curve. The attribute `edge_class[i]` is the edge class associated with the  $i$ -th oscillating curve. Each oscillating curve is stored as a doubly linked list of `CurveComponents`. The attribute `curve_begin[i]` is the header node of a doubly linked list of `CurveComponents` for the  $i$ -th oscillating curve. In order to calculate the combinatorial holonomy of each curve, we store the intersection numbers of each curve with each cusp edge, in a similar fashion to the boundary curves. SnapPy provides a generic `extra` pointer on each `Tetrahedron` which we are free to declare the implementation of and use.

```
struct extra {
    int curve[4][4];
};
```

We add an array of `manifold->num_tetrahedra` `extra` structs to each `Tetrahedron`. On a tetrahedron `tet`, `tet->extra[edge_class].curve[v][e]` is the net number of the oscillating curve associated with edge class `edge_class`, crosses the cusp edge `e` of the cusp triangle at vertex `v` of tetrahedron `tet`.

## 6.9 Neumann-Zagier Matrix

We describe explicitly the process SnapPy uses to reconstruct the combinatorial holonomy from the intersection numbers of a curve. The function to do this for `m` and `l` is contained in `kernel/addl_code/get_gluing_equations.c` in the function `get_cusp_equation()`, and described in Algorithm 6. The process applies to the oscillating curves, replacing `tet.curve[c][right_handed]` with `tet.extra[edge_class].curve`. The arrays `remaining_face` and `edge3_between_faces` are tables provided by SnapPy for working with tetrahedra. SnapPy contains the following comment explaining the table:



"Given two faces  $i$  and  $j$ , `remaining_face[i][j]` tells you the index of one of the remaining faces. For a `right_handed` tetrahedron the faces  $i$ ,  $j$ , and `remaining_face[i][j]` are arranged in a counterclockwise order around their common ideal vertex. Thus, for two faces  $i$  and  $j$ , `remaining_face[i][j]` gives one of the remaining faces and `remaining_face[j][i]` gives the other." [Cul+23]. The definition of a right handed tetrahedron is not important, all tetrahedra in a triangulation of a manifold with tori boundary components are right handed. We use the remaining face table to ensure we pick up a negative sign when we go clockwise around a vertex, and a positive sign when we go anti-clockwise. The array `edge3_between_faces[i][j]` gives the index of the complex edge parameter associated with the edge of the triangulation lying between faces  $i$  and  $j$ . This gives us the basis vector  $\mathbf{a}$ ,  $\mathbf{b}$  or  $\mathbf{c}$  lying between  $i$  and  $j$ . Finally, the function `FLOW` reconstructs simple closed curves from the homological information. More specifically "Let  $A$  be the number of times a curve intersects one side of a triangle, and  $B$  be the number of times it intersects a different side (of the same triangle). Then `FLOW(A,B)` is the number of strands of the curve passing from the first side to the second." [Cul+23].

---

**Algorithm 6:** Combinatorial Holonomy

---

**input :** Triangulation  $M$ , and boundary curve  $c$ .  
**output:** The combinatorial holonomy of the boundary curve  $c$ .

```

1  $eqn \leftarrow [0] * (3 * manifold.num\_tetrahedra);$ 
2 for  $tet \in manifold.tetrahedra[]$  do
3   for  $0 \leq v \leq 3$  do
4     for  $0 \leq f_1 \leq 3$  and  $f_1 \neq v$  do
5        $f_2 \leftarrow remaining\_face[vertex][f_1];$ 
6        $eqn[3 * tet.index + edge3\_between\_faces[f_1][f_2]] +=$ 
          $FLOW(tet.curve[c][right\_handed][v][f_1], tet.curve[c][right\_handed][v][f_2]);$ 
7     end
8   end
9 end
10 return  $eqn;$ 
```

---

## 7 Examples

### 7.1 Figure-8 Knot

To illustrate the algorithm, we find oscillating curves for the figure-8 knot, which turn out to be different to the ones found in Section 4. SnapPy gives the figure-8 knot a triangulation consisting of 2 tetrahedra. To begin, we construct a small picture of the single cusp, Figure 33, using the gluing information provided by SnapPy.

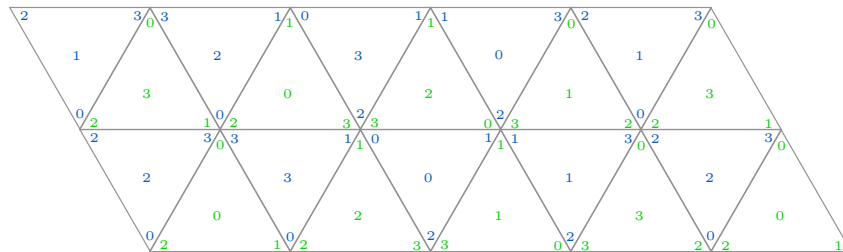


Figure 33: Cusp Triangulation of the Figure 8 knot. Cusp triangles from tetrahedron 0 are coloured blue, cusp triangles from tetrahedron 1 are colored green. The central number of each cusp triangle is the vertex of the tetrahedron truncated by the cusp triangle. The number at a vertex of the cusp triangle is the label on the cusp edge opposite the vertex.

Then we extract the intersection numbers of the meridian and longitude curves with the edges of each cusp triangle, which is provided by SnapPy, and draw a candidate curve on the cusp, Figure 34 and Figure 35.

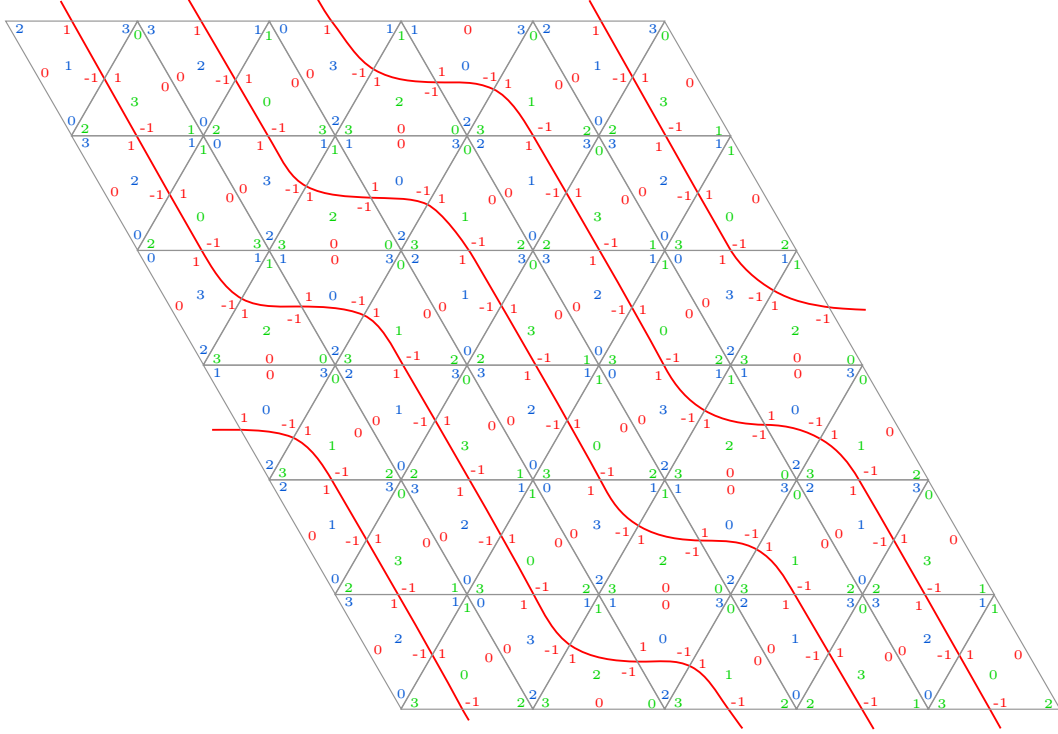


Figure 34: Meridian Curves

The function `peripheral_curve()` picks out the cusp triangle coming from vertex 0 of tetrahedron 0 as the intersection cusp triangle. So we combine the meridian and longitude curves onto one picture, placing the intersection on this cusp triangle, Figure 36.

Next we construct the cusp regions on this cusp with the boundary curves, and the cusp region graph (Figure 37).

The end multi graph consists of only a single vertex and edge which loops on the vertex. Hence, an oscillating curve consists of two components, both of which start at edge class 0 and finish at edge class 1. Searching through the cusp regions sequentially, we find cusp region 2 as the first cusp region which can dive along face 2, into the cusp vertex 3 which corresponds to edge class 0 and edge index 0. Similarly, we find cusp region 2 as a region which can dive along face 3 into the cusp vertex 1 which corresponds to edge class 1 and edge index 0. Using breadth first search we find a path from vertex 2 to vertex 2, which consists of a single node. Then we place a curve on cusp region 2 which dives into the two cusp vertices, and split cusp region 2 by modifying the attributes of cusp region 2 and adding cusp region 25, Figure 38.

The choice of cusp region 2 to dive along face 2 into vertex 3 induces the choice of cusp region 6, diving along face 1 into vertex 0. Similarly, the choice of cusp region 2 to dive along face 3 into vertex 1 induces the choice of cusp region 12, diving along face 3 into vertex 0. Then we find a path through the Cusp Region graph from vertex 6 to vertex 12, which gives the path  $[6, 16, 3, 21, 12]$  (Figure 39).

Finally, we calculate the combinatorial holonomy for the meridian and longitude curve, the curve going around edge class 1,  $C_1$ , and the oscillating curve.

$$SY = \begin{pmatrix} 0 & 0 & -2 & 4 \\ -1 & 0 & -2 & -3 \\ -2 & -1 & 1 & 2 \\ 0 & -1 & 1 & 2 \end{pmatrix}$$

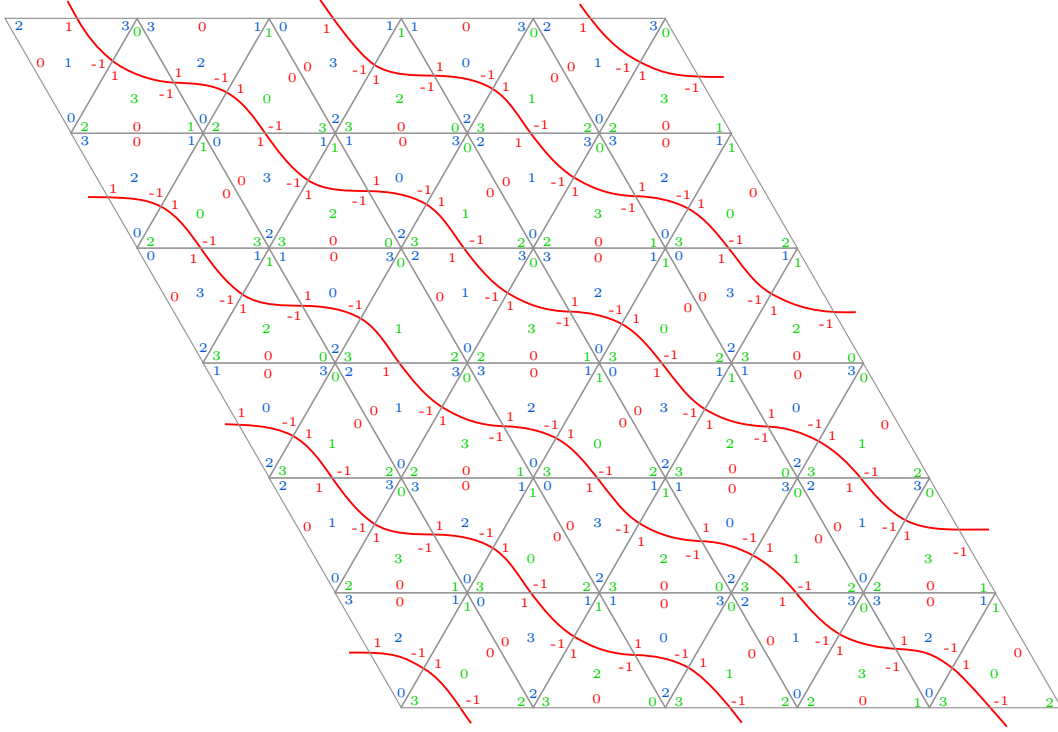


Figure 35: Longitude

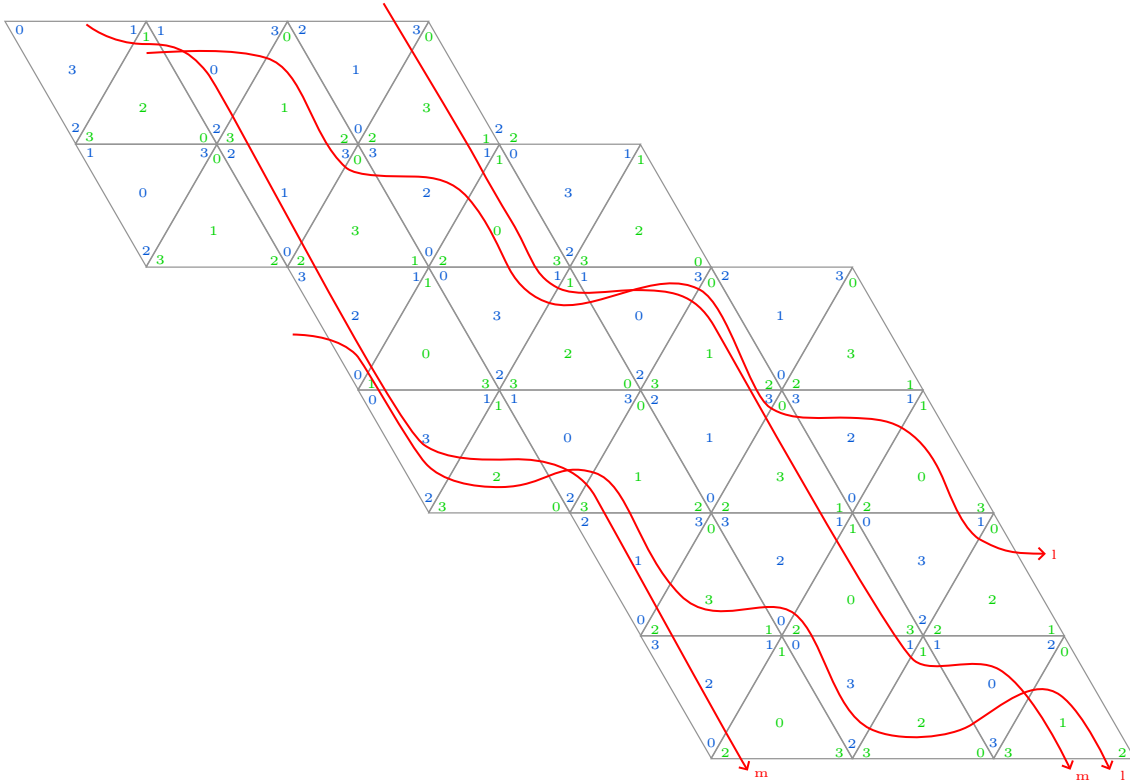


Figure 36: Meridian and longitude curves, with the intersection placed on the cusp triangle at vertex 0 of tetrahedron 0.

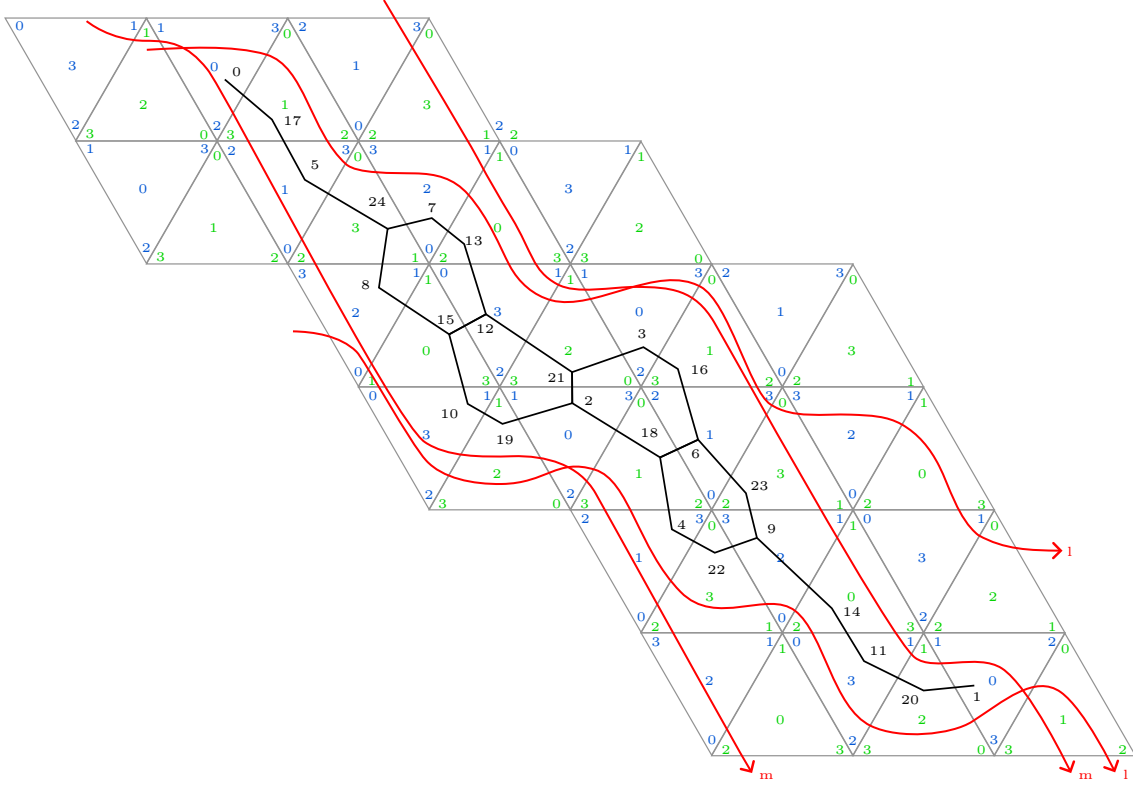


Figure 37: Cusp Region graph.

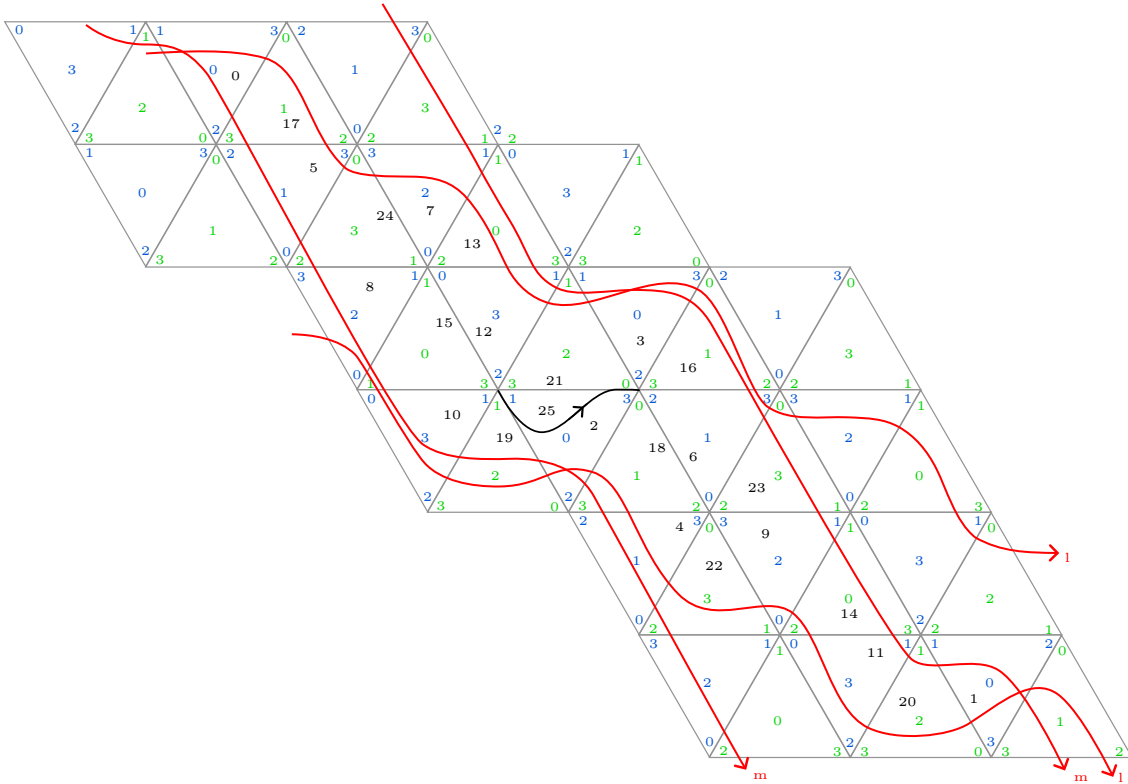


Figure 38: First half of the oscillating curve.

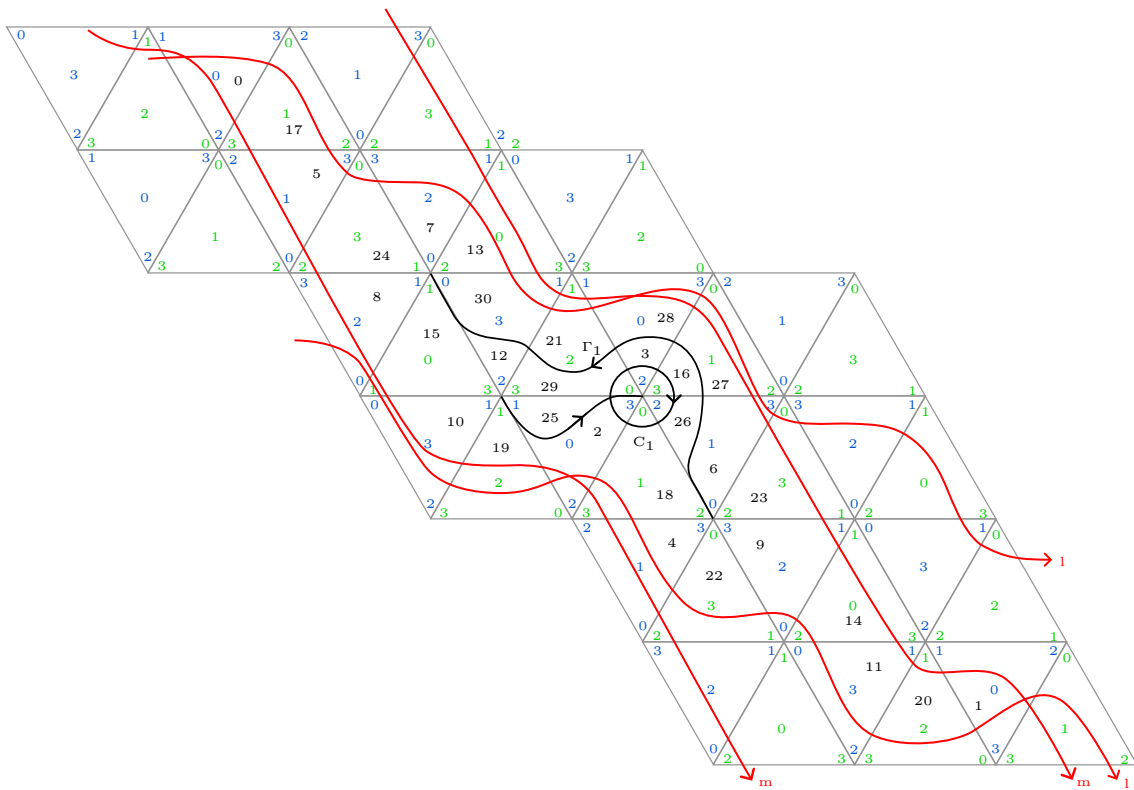


Figure 39: Second half of the oscillating curve.

## 7.2 L6a4 Link

Consider the L6a4 link, shown in Figure 40. The link complement of L6a4 has 3 cusps, and SnapPy gives the complement a triangulation consisting of 8 tetrahedra [Cul+23]. Show in Figure 41, 42, and 43 are the cusp triangulations for each cusp of L6a4. We chose a spanning tree for the end multi graph consisting of edge class 4, which connects cusp 0 to cusp 1, and edge class 2, which connects cusp 0 to cusp 2. We also pick edge class 0, which connects cusp 1 to cusp 2, to be  $E_0$ . For the train lines with sidings, we have 3 collections of edge classes,

$$L(\mathcal{C}_0) = \{2, 4\}, \quad L(\mathcal{C}_1) = \{0, 1, 4, 6\}, \quad L(\mathcal{C}_2) = \{0, 2\}.$$

We have 5 oscillating curves, which are made up the following components,

- $\Gamma_0$ : Corresponds to edge class 1, which connects cusp 1 to cusp 1.
  - Cusp 1: Starts at edge class 1, edge index 1 and ends at edge class 0, edge index 0.
  - Cusp 2: Starts at edge class 2, edge index 0 and ends at edge class 0, edge index 0.
  - Cusp 0: Starts at edge class 2, edge index 0 and ends at edge class 4, edge index 0.
  - Cusp 1: Starts at edge class 1, edge index 0 and ends at edge class 4, edge index 0.
- $\Gamma_1$ : Corresponds to edge class 3, which connects cusp 1 to cusp 2.
  - Cusp 1: Starts at edge class 3, edge index 0 and ends at edge class 4, edge index 0.
  - Cusp 2: Starts at edge class 3, edge index 0 and ends at edge class 4, edge index 0.
- $\Gamma_2$ : Corresponds to edge class 5, which connects cusp 1 to cusp 2.
  - Cusp 1: Starts at edge class 5, edge index 0 and ends at edge class 2, edge index 0.
  - Cusp 2: Starts at edge class 5, edge index 0 and ends at edge class 2, edge index 0.
- $\Gamma_3$ : Corresponds to edge class 6, which connects cusp 1 to cusp 1.
  - Cusp 1: Starts at edge class 6, edge index 1 and ends at edge class 0, edge index 0.
  - Cusp 2: Starts at edge class 2, edge index 0 and ends at edge class 0, edge index 0.
  - Cusp 0: Starts at edge class 2, edge index 0 and ends at edge class 4, edge index 0.
  - Cusp 1: Starts at edge class 6, edge index 0 and ends at edge class 4, edge index 0.
- $\Gamma_4$ : Corresponds to edge class 7, which connects cusp 1 to cusp 2.
  - Cusp 1: Starts at edge class 7, edge index 0 and ends at edge class 0, edge index 0.
  - Cusp 2: Starts at edge class 7, edge index 0 and ends at edge class 0, edge index 0.

The algorithm finds the train lines with sidings and oscillating curves shown in Figure 44, 45, and 46. Calculating the combinatorial holonomy of each boundary curve, edge curve and oscillating curve we obtain the following matrix.

$$\begin{aligned}
\mathbf{m}_0 : & ( 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1 ) \\
\mathbf{l}_0 : & ( -1, -1, 0, 1, 1, 1, -1, 0, 0, 0, 0, 0, -1, -1, 0, 0, 0 ) \\
\mathbf{m}_1 : & ( 0, -1, 1, 1, 1, 0, 0, -1, 0, 1, 1, 0, 1, 1, -1, -1 ) \\
\mathbf{l}_1 : & ( -1, -1, 0, 0, 0, -1, 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0 ) \\
\mathbf{m}_2 : & ( 0, -1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0 ) \\
\mathbf{l}_2 : & ( 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0 ) \\
C_0 : & ( -1, -1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, -1 ) \\
\Gamma_0 : & ( 0, -1, 0, -1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0 ) \\
C_1 : & ( 0, 1, -1, -1, 0, -1, 0, 0, -1, -1, 1, 1, 0, 0, 0, 0, 1 ) \\
\Gamma_1 : & ( 0, 0, 0, 0, 1, 1, -1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0 ) \\
C_2 : & ( 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, -1, -1, -1 ) \\
\Gamma_2 : & ( 1, 1, 0, 0, 0, -1, 1, 0, 1, 0, 0, 1, 1, -1, 1, 0, 0 ) \\
C_3 : & ( 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, -1, -1, -1, -1, 0, 0, 0 ) \\
\Gamma_3 : & ( -1, -2, 1, 2, -1, 0, 0, 1, 0, 0, 1, 1, 2, 1, -1, -1, -1 ) \\
C_4 : & ( 0, 0, 0, 0, -1, -1, 1, 0, 0, -1, 0, 1, 1, 0, 1, 1, 1 ) \\
\Gamma_4 : & ( 1, 0, 0, 0, -1, 0, 1, 2, 1, -1, -1, 0, 0, -1, 1, 1, 1 )
\end{aligned}$$

This matrix is not symplectic (up to factors of 2) since,

$$\omega(\Gamma_0, \Gamma_3) = 2.$$

It is currently unknown why these curves have non zero holonomy, since they have the same holonomy for each cusp triangle they enter.

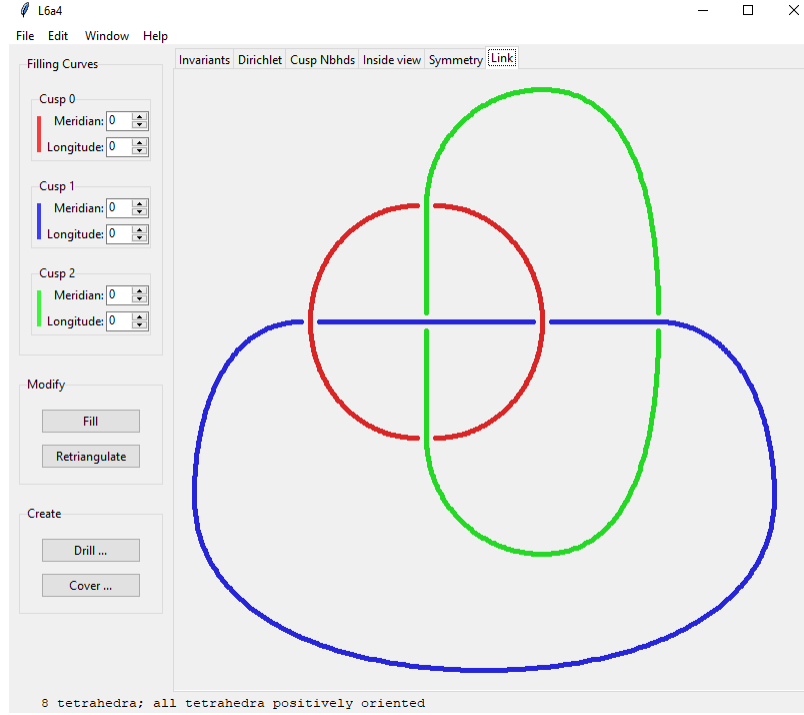


Figure 40: L6a4 link [Cul+23].

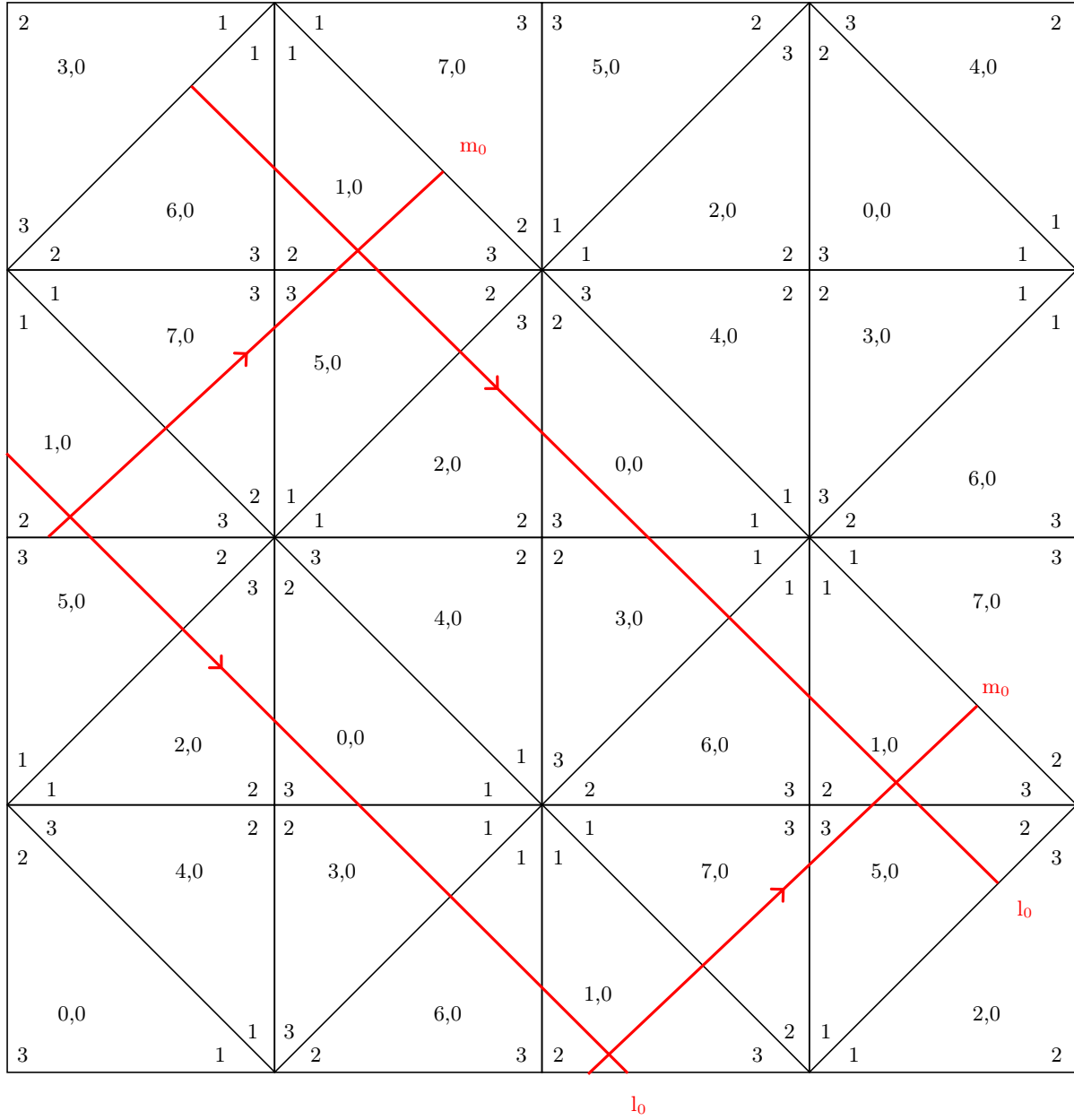


Figure 41: Cusp 0 of L6a4 link complement. The center of each cusp triangle contains the `tet_index` and `tet_vertex` of the cusp triangle.



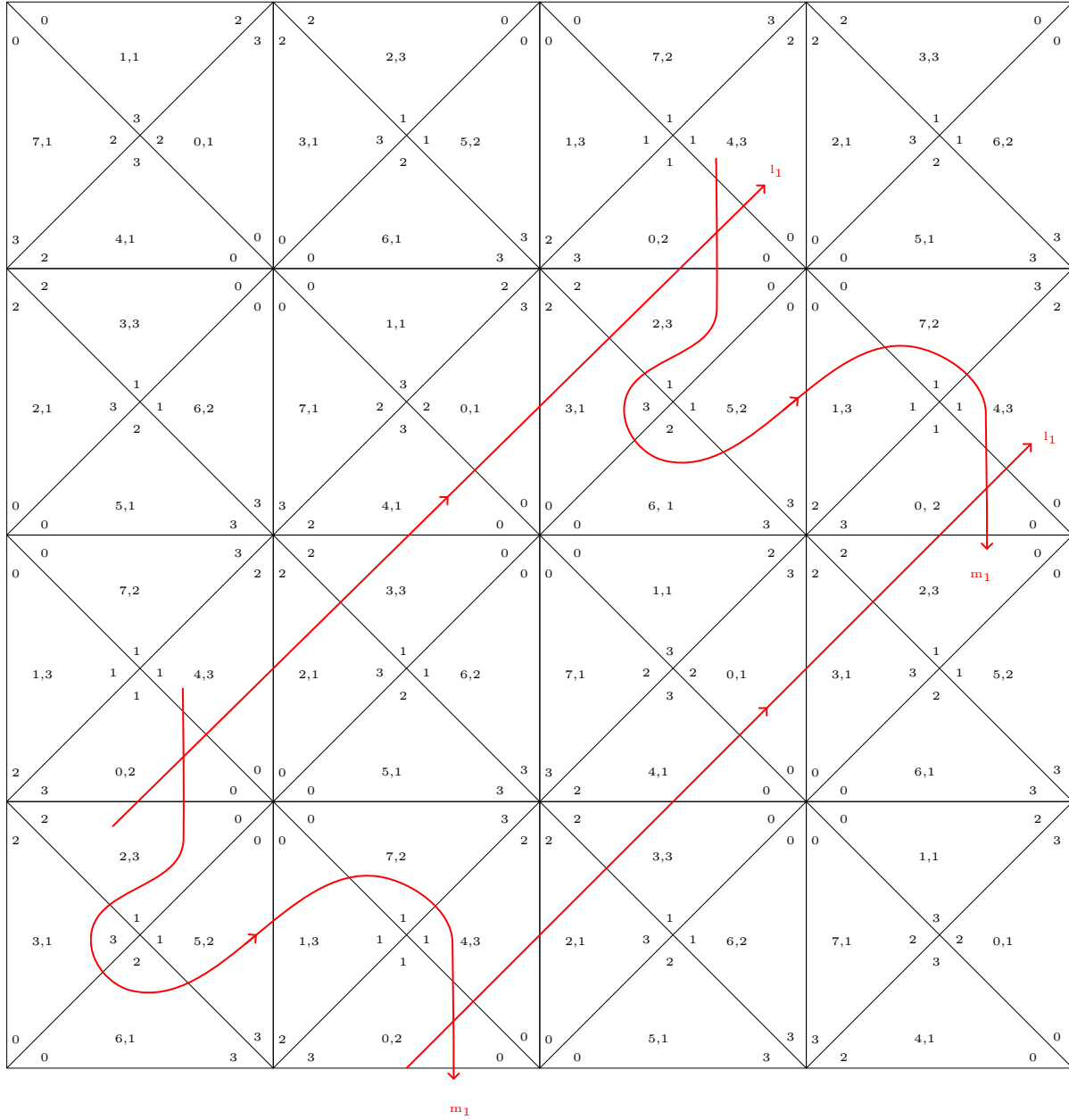


Figure 42: Cusp 1 of L6a4 link complement. The center of each cusp triangle contains the `tet_index` and `tet_vertex` of the cusp triangle.

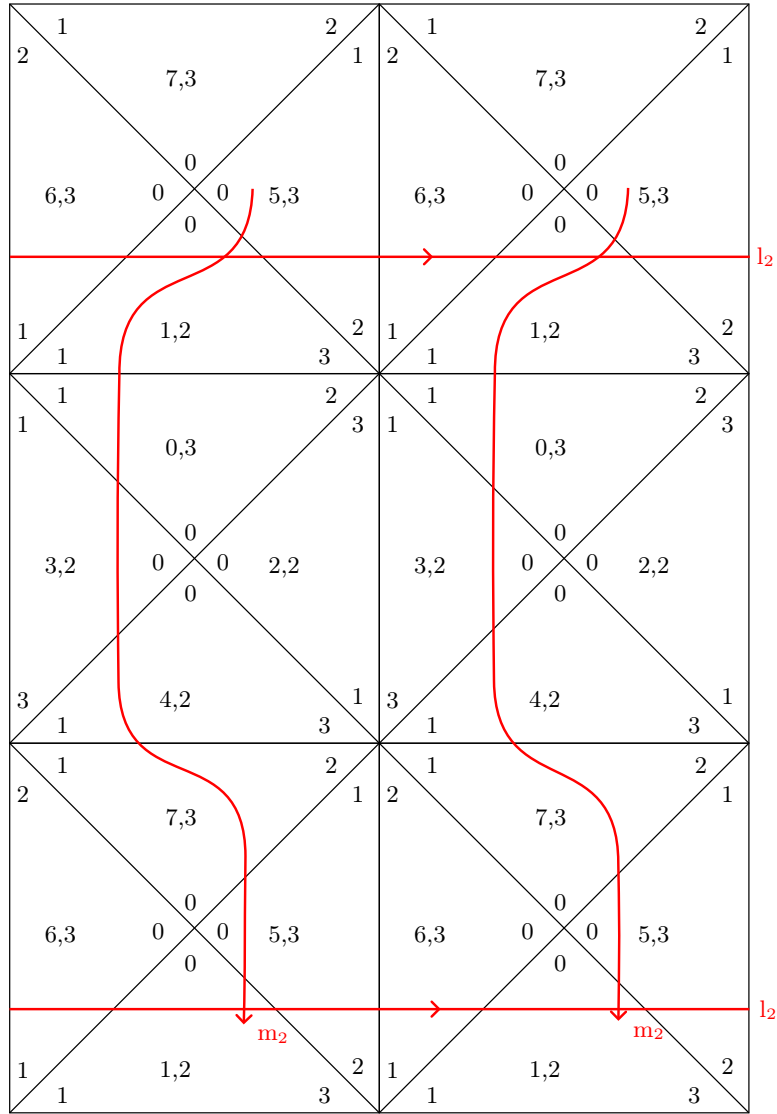
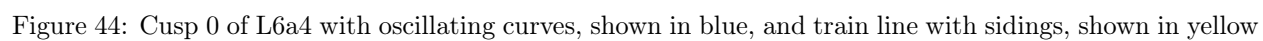


Figure 43: Cusp 2 of L6a4 link complement. The center of each cusp triangle contains the `tet_index` and `tet_vertex` of the cusp triangle.



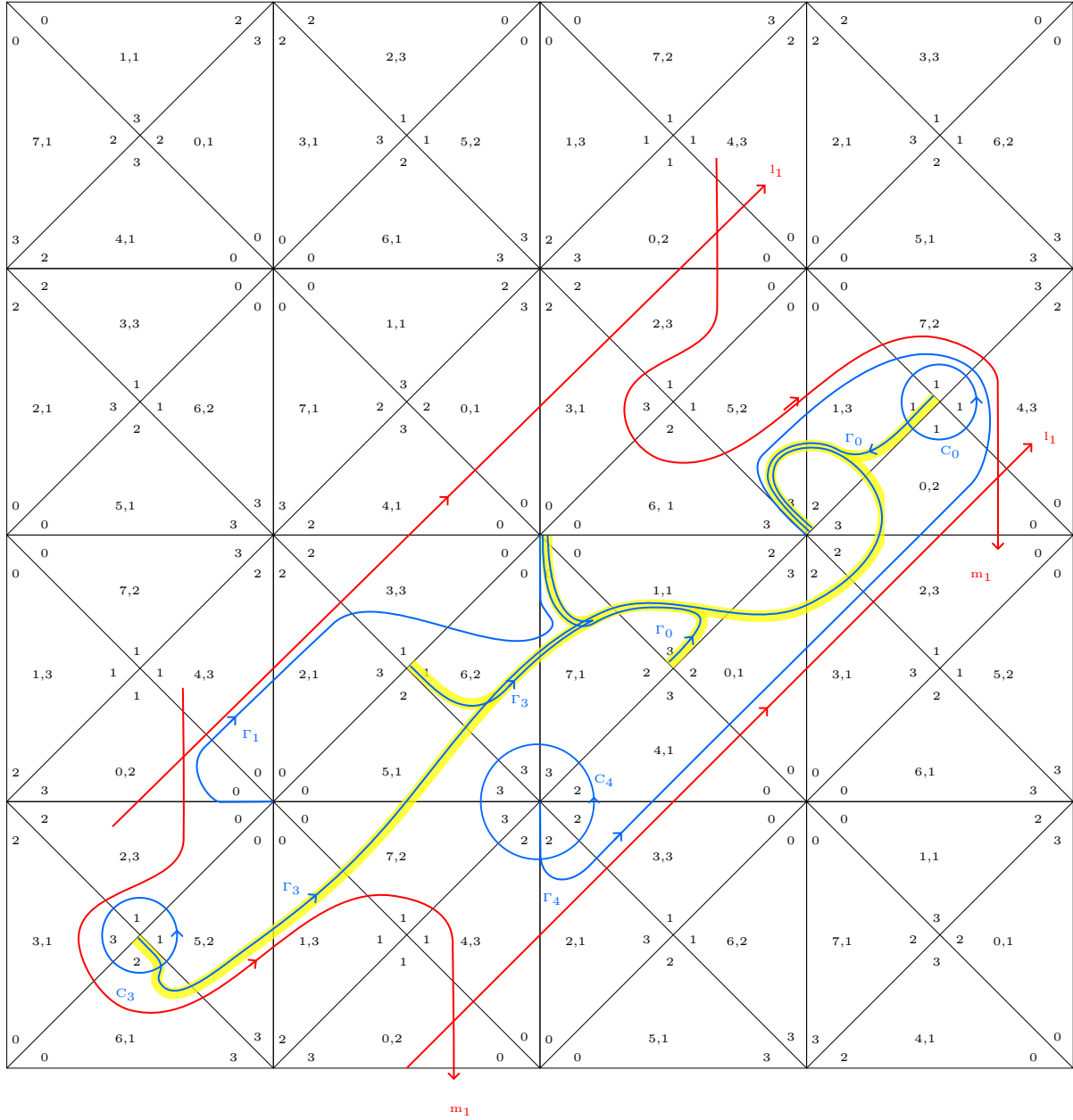


Figure 45: Cusp 1 of L6a4 with oscillating curves, shown in blue, and train line with sidings, shown in yellow

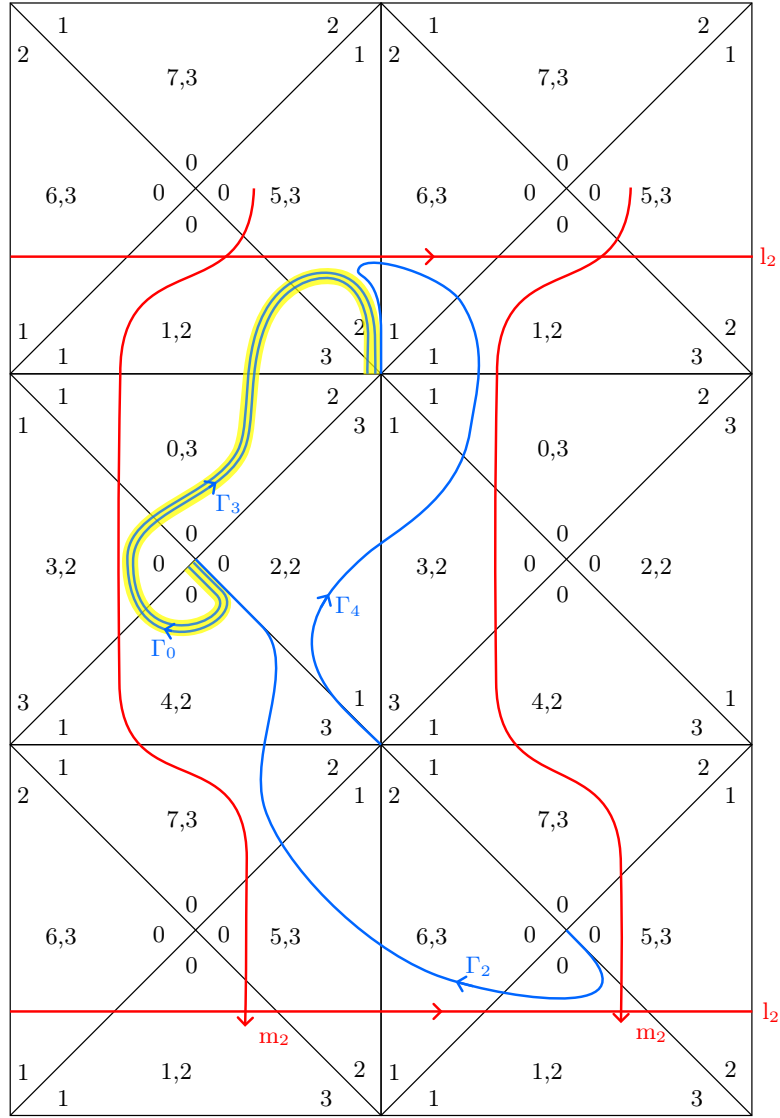


Figure 46: Cusp 2 of L6a4 with oscillating curves, shown in blue, and train line with sidings, shown in yellow

## 8 Conclusion

In conclusion, we have developed and implemented an algorithm which constructs a symplectic basis for knot complements. We have made progress towards constructing oscillating curves on link complements. The implementation for knot complements has been tested on the `HTLinkExteriors` database which contains 313,230 knots. This has been merged into the SnapPy source code [Cul+23], with the algorithm contained in the C file `kernel/kernel_code/symplectic_basis.c`. The train lines are partially implemented in the `symplectic-basis` python package <https://pypi.org/project/symplectic-basis/>, with the source code available at <https://github.com/jchilds0/symplectic-basis>. Further research is needed to understand how oscillating curves on train lines with sidings interact with the symplectic form  $\omega$ , along with proofs of the conjectures made earlier in this paper. SnapPy currently contains a gui to view links and the cusp triangulations of each cusp, this could be extended to show the oscillating curves and train line constructed. Currently, the oscillating curve information is kept long enough to construct the symplectic matrix. Instead, this information could be stored as part of the triangulation, to be used elsewhere as new algorithms involving oscillating curves are developed.

## 9 Bibliography

- [Bin59] R. H. Bing. “An Alternative Proof that 3-Manifolds Can be Triangulated”. In: *Annals of Mathematics* 69.1 (1959), pages 37–65.
- [Cul+23] Marc Culler et al. *SnapPy, a computer program for studying the geometry and topology of 3-manifolds*. Available at <http://snappy.computop.org>. 2023.
- [HMP21] A. Joshua Howie, Daniel V. Mathews, and Jessica S. Purcell. *A Polynomials, Ptolemy Equations and Dehn Filling*. 2021. URL: <https://doi.org/10.48550/arXiv.2002.10356>.
- [Kau83] Louis H. Kauffman. *Formal Knot Theory*. Princeton University, 1983.
- [Mat23] Daniel V. Mathews. *Spinors and Horospheres*. 2023. URL: <https://doi.org/10.48550/arXiv.2308.09233>.
- [MP22] Daniel V. Mathews and Jessica S. Purcell. *A Symplectic Basis for 3-Manifold Triangulations*. 2022. URL: <https://dx.doi.org/10.48550/arxiv.2208.06969>.
- [Moi52] Edwin E. Moise. “Affine Structures in 3-manifolds: The Triangulation Theorem and Hauptvermutung”. In: *Annals of Mathematics* 56.1 (1952), pages 96–114.
- [Mos86] GD Mostow. “Quasi-conformal mappings in  $n$ -space and the rigidity of hyperbolic space forms”. In: *Publications Mathématiques de l’IHÉS* 34 (1986), pages 53–104.
- [Neu92] Walter D Neumann. “Combinatorics of Triangulations and the Chern-Simons Invariant for Hyperbolic 3-Manifolds”. In: *Topology* 90 (1992), pages 243–271.
- [NZ85] Walter D. Neumann and Don Zagier. “Volumes of Hyperbolic Three-Manifold”. In: *Topology* 24.3 (1985), pages 307–332.
- [Pen10] Robert C. Penner. *Decorated Teichmüller Theory*. European Mathematical Society, 2010.
- [Pra73] G Prasad. “Strong Rigidity of  $\mathbb{Q}$ -rank 1 Lattices”. In: *Invent Math* 21 (1973), pages 255–286.
- [Pur20] Jessica S. Purcell. *Hyperbolic Knot Theory*. Volume 209. American Mathematical Society, 2020.
- [Thu80] William P. Thurston. *The Geometry and Topology of Three-Manifolds*. Princeton University, 1980.
- [Thu82] William P. Thurston. “Three Dimensional Manifolds, Kleinian Groups and Hyperbolic Geometry”. In: *Bulletin of the American Mathematical Society* 6.3 (1982), pages 357–382.
- [Wee03] Jeffery R. Weeks. *Computation of Hyperbolic Structures in Knot Theory*. 2003. URL: <https://dx.doi.org/10.48550/arxiv.math/0309407>.
- [Wee85] Jeffrey R. Weeks. *Hyperbolic Structures on Three-Manifolds*. 1985.