Computational Approach to the Geometry of Compact Riemann Surfaces

THÈSE N^{O} 5719 (2013)

PRÉSENTÉE LE 15 MAI 2013
À LA FACULTÉ DES SCIENCES DE BASE
CHAIRE DE GÉOMÉTRIE
PROGRAMME DOCTORAL EN MATHÉMATIQUES

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Manuel RACLE

acceptée sur proposition du jury:

Prof. T. Mountford, président du jury
Prof. J. Buser, directeur de thèse
Prof. R. Silhol, rapporteur
Prof. A. Strohmaier, rapporteur
Prof. M. Troyanov, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2013

La géometrie n'est pas vraie. Elle est avantageuse.
— Henri Poincaré

To my parents Sylvie and Denis
To my brother Julien
To Claudia the love of my life

Acknowledgments

This research project would not have been possible without the help and support of many people.

The author wishes especially to express his infinite gratitude to his supervisor Prof. Peter Buser. The tremendous support and help, his patience and his huge kindness have been the key to the success of this thesis.

Our special thanks also go to the other members of the Mathematics section, notably to Dr. Klaus-Dieter Semmler for the very interesting mathematical discussions we had, and especially to Prof. Marc Troyanov for accepting to be a member of the jury of my thesis.

We also thank the two secretaries, Maroussia Schaffner-Portillo and Anna Dietler for their availability and their kindness.

Big thanks also go to my graduate colleagues, Grégoire Aubry, Björn Mützel, Marc Maintrot, Anthony Arnold, Rachel Bendjama, Christophe and Simona Hebeisen, Madeleine Jotz and Hugo Parlier for all the fun we had working together.

We also want to express our gratefulness to the EPFL as well as to the Swiss National Science Found for having allowed this research to be done through the amazing facilities and research environment they provided us as well as the fundings they granted.

Finally, last but not least, the author wishes to express his love and gratitude to his beloved family, his parents Denis and Sylvie as well as his brother Julien.

We kept our last thanks for my beloved Claudia, the star of my heart, for her support her patience and her love.

Lausanne, 11 February 2013 M. R.

Abstract

The goal of this document is to provide a general method for the computational approach to the topology and geometry of compact Riemann surfaces. The approach is inspired by the paradigms of *object oriented programming*.
Our methods allow us in particular to model, for numerical and computational purposes, a compact Riemann surface given by Fenchel-Nielsen parameters with respect to an arbitrary underlying graph, this in a uniform and robust manner.
With this programming model established we proceed by proposing an algorithm that produces explicit compact fundamental domains of compact Riemann surfaces as well as generators of the corresponding Fuchsian groups. In particular, we shall explain how one may obtain convex geodesic canonical fundamental polygons.
In a second part we explain in what manner *simple closed geodesics* are represented in our model. This will lead us to an algorithm that enumerates all these geodesics up to a given prescribed length.
Finally, we shall briefly overview a number of possible applications of our method, such as finding the *systoles* of a Riemann surface, or drawing its Birman-Series set in a fundamental domain.

Keywords Riemann surface, Fenchel-Nielsen parameters, enumeration algorithms, simple closed geodesic, fundamental domain, Fuchsian group, Möbius transformation, algorithmic hyperbolic geometry.

Résumé

Le but de ce document est d'apporter une manière de décrire la structure topologique et géometrique des surfaces de Riemann en s'inspirant des paradigmes de la *programmation orientée objet*.

Ceci nous permet notamment de modéliser une surface de Riemann compacte donnée à partir de n'importe quel type de graphe de Fenchel-Nielsen d'une manière uniforme et fiable.

Partant de cela, nous proposons ensuite un algorithme permettant d'obtenir explicitement des *domaines fondamentaux* pour les surfaces, ainsi que des générateurs des *groupes Fuchsiens* associés. Nous expliquons en particulier de quelle manière on peut obtenir un domaine fondamental *convexe et canonique*.

Ensuite, nous décrivons de quelle manière il est possible de représenter les *géodésiques fermées simples* d'une surface. Puis nous proposons un algorithme permettant de les énumérer.

Finalement, nous introduisons brièvement quelques exemples d'application possibles à ces algorithmes, comme par exemple, de trouver l'ensemble des *systoles* d'une surface, ou de dessiner l'ensemble de Birman-Series dans un domaine fondamental de la surface.

Mots Clés Surface de Riemann, paramètres de Fenchel-Nielsen, algorithme d'énumération, géodésique fermée simple, domaine fondamental, groupe Fuchsien, transformation de Möbius, géométrie hyperbolique algorithmique.

Contents

Acknowledgments v

Abstract (English/Français) vii

Contents xiii

Introduction 1

I Geometry of Riemann Surfaces 3

1 Riemann Surfaces 7
- 1.1 Examples 8

2 Uniformization theorem 11

3 Hyperbolic geometry 15
- 3.1 The models $\mathbb{H}$ and $\mathbb{D}$ 16
  - 3.1.1 The hyperbolic disk $\mathbb{D}$ 17
  - 3.1.2 The hyperbolic half-plane $\mathbb{H}$ 19
- 3.2 $M\mathbb{H}$, $M\mathbb{D}$ models 21
- 3.3 Hyperbolic trigonometry 23
  - 3.3.1 Triangles 23
  - 3.3.2 Right-angled triangles 23
  - 3.3.3 Trirectangles 23
  - 3.3.4 Right-angled pentagons 24
  - 3.3.5 Right-angled hexagons 24
  - 3.3.6 Crossed right-angled hexagons 24
- 3.4 Trace bounding 25

II Computational approach 41

4 Surface modeling 47
- 4.1 Fenchel-Nielsen parameters modeling 48
- 4.2 Surface sub-objects 50

4.3 Winding index 54
4.4 The `SurfaceTopology` module 56
  4.4.1 The `Surface` class 57
  4.4.2 The `Collar` class 59
  4.4.3 The `Border` class 60
  4.4.4 The `YPce` class 61
  4.4.5 The `Road` class 62
  4.4.6 The `Slot` class 64

5 Fundamental domains **67**
5.1 Frames in $\mathbb{H}$ 69
  5.1.1 The `HFrame` class 70
5.2 Hyperbolic polygons 72
5.3 Domain representation 77
5.4 The `HyperbolicDomain` module 79
  5.4.1 The `DomVertex` class 80
  5.4.2 The `DomEdge` class 84
  5.4.3 The `DomIdentification` class 85
  5.4.4 The `Domain` class 85
  5.4.5 The `YDom` class 86
  5.4.6 The `HexDom` class 89
  5.4.7 The `CollarDom` class 89
  5.4.8 The `BasicDomain` class 90
  5.4.9 The `CanonicalDomain` class 90
5.5 Basic Domain construction 92
5.6 Canonical Domain construction 96
5.7 Convexity of the polygon 107

6 Exploration algorithms **113**
6.1 Encoding of simple closed geodesics 116
6.2 Simple Path 122
6.3 Path geometry 130
6.4 Path exploration 133
  6.4.1 Codeword 133
  6.4.2 Algorithms 135
6.5 The `SurfacePath` module 137
  6.5.1 The `SurfaceLayer` class 138
  6.5.2 The `RoadLayer` class 140
  6.5.3 The `CollarLayer` class 141
  6.5.4 The `SlotLayer` class 142
  6.5.5 The `Path` class 144
  6.5.6 The `PathPoint` class 146
  6.5.7 The `PathElem` class 149

6.6 The `PathAlgorithms` module 150
  6.6.1 The `PathExplorer` class 150
  6.6.2 The `PathDecoder` class 152
  6.6.3 The `SimpleGeodesicFinder` class 153

7 Applications 157
7.1 Finding a base of the fuchsian group 157
7.2 Finding closed geodesics 157
7.3 Testing intersection between two simple closed geodesics 158
7.4 *Bers* decompositions 158
7.5 *Birman-Series* set 159

List of figures 164

Bibliography 166

Introduction

This document aims to serve as a *handbook* that explains the working manner of a new program library that we developed during this thesis. In addition, we shall give mathematical proofs that what we propose indeed works.

Our first goal in the research for the thesis was to use the *Object Oriented Programming* paradigms in order to be able to produce uniform programming models of compact hyperbolic Riemann surfaces given by Fenchel-Nielsen parameters based on an arbitrary *Fenchel-Nielsen graph.* Then, we present in considerable detail new algorithms designed to initially obtain (non-canonical) *fundamental domains* for a given surface. We also explain how to transform, algorithmically, any (non-canonical) *fundamental domain* into a new one that is both canonical and convex.

In a second part we turn our interest towards the *simple closed geodesics* of a compact Riemann surface. We firstly describe a way to represent these curves in a manner that enables us to enumerate them. We also explain how the geometric properties of the simple closed geodesics (like their length) can be deduced from their model.

We then present an efficient enumeration algorithm for the *simple closed geodesics* that yields all these geodesics up to a prescribed length. From this general enumeration process we can deduce some examples of algorithms that help solving geometric problems such as finding all the *systoles* of a surface, or finding a *Bers decomposition* of it.

This document is organized *as* follows.

It consists of two Parts. The first one, *Geometry of Riemann surfaces* is meant to be an introduction to the mathematics that are involved in this thesis and contains nothing about algorithms or programming. Here we also prove a new geometric inequality.

Chapters 1 and 2 recapitulate some basic notions about the Riemann surfaces, such as their definition.

This can be useful to help a reader who is not familiar to the subject to understand the context of what we are trying to do.

Chapter 3 contains informations and formulas about *hyperbolic geometry*. The Subsections

3.1,3.2 and 3.3 contain some known material and formulas that are used within our algorithms and they are reproduced within this document for the comfort of the reader. The Subsection 3.4 contain some original results that are used to justify the algorithms exposed in the last chapters of this thesis.

The second part of this document, "Computational approach", puts the emphasis on the computations and algorithms.

It contains four chapters.

Chapter 4 details how, starting from some Fenchel-Nielsen graph, we can produce a detailed model of the corresponding surface in terms of *classes*.

Chapter 5 details the algorithms used to construct some convex canonical fundamental domains of a given surface, as well as the (canonical) generators of the corresponding Fuchsian group.

Chapter 6 explains how to model and how to enumerate the simple closed geodesics on a surface.

And, finally, Chapter 7 serves as a kind of conclusion in the sense that we describe without entering details some other examples of algorithms that our library implements.

Geometry of Riemann Surfaces Part I

In this part, we recapitulate, the useful basic mathematical knowledge about Riemann surfaces and hyperbolic geometry, that we use in this document.

It will help to clarify the notations we choose, as well as it will help the reader to get familiar with this subject even if it is not his own field.

In Chapter 1, we provide the accurate definition of a *riemann surface*.

Then, in Chapter 2, we provide the very important *uniformization theorem* that shows that every *riemann surface* can be categorized in one of the 3 *types*:

Parabolic surfaces Parabolic surfaces posses a metric of constant Gauss curvature $+1$.

Elliptic surfaces Elliptic surfaces posses a metric of constant Gauss curvature 0.

Hyperbolic surfaces Hyperbolic surfaces posses a metric of constant Gauss curvature -1.

We will also see that the *parabolic* and *elliptic* cases leads to very few possible surfaces and requires no more investigations. That is why we restrict the rest of our study to the *hyperbolic* case.

In Chapter 3 we then describe the geometric properties of the *hyperbolic plane* $\mathbb{H}$ seen as a *metric space* of constant Gauss curvature -1. This space is of prime importance as we use it as the *universal covering* of our *hyperbolic Riemann surfaces*. This chapter also contains description of other equivalent models of hyperbolic space, called *matrix models*, that are very handy to make practical computations related to hyperbolic geometry. Finally some original results are presented in the final section 3.4 of this chapter that will be used in the algorithms of Chapter 6.

Chapter 2 explains how to describe an hyperbolic surface $S = \mathbb{H}/\Gamma$ as the quotient of the hyperbolic space $\mathbb{H}$ by a discrete group of Möbius transformations $\Gamma \subset \text{Möb}(\mathbb{H})$. Not any discrete subgroup of Möb($\mathbb{H}$) gives rise to a valid surface. That is why, in Definition 7, we express the conditions that Γ must respect such that $\mathbb{H}/\Gamma$ is valid surface. A group verifying such conditions is called a *fuchsian group*.

1 Riemann Surfaces

For the benefit of the reader who comes from a different field we review in the first two, very short, chapters some background concepts collected from textbooks. Only few definitions are given, the goal being only to indicate where the subject of this thesis sits. The basic reference for the two chapters is [7].

Let us begin by giving a definition of a *Riemann surface*[1].

Definition 1. *A Riemann surface S is a complex manifold of dimension one.*

In other words, S is a *separated topological space* such that there exists a covering $\Omega_i \subset S, i \in I$ of open subsets of S such that $\bigcup_{i\in I} \Omega_i = S$.

Furthermore, for every Ω_i, there must exist a *homeomorphism* $f_i : \Omega_i \to D_i \subset \mathbb{C}$. The pair (Ω_i, f_i) is called a *chart* of S. The set of the *charts* covering S is called an *atlas* of S.

Finally, we want that if two charts overlap, $\Omega_i \cap \Omega_j = \Sigma \neq \emptyset$, the *transition map*

$$f_j \circ f_i^{-1}\big|_{f_i(\Sigma)} : f_i(\Sigma) \to f_j(\Sigma)$$

is requested to be an *holomorphic function*.

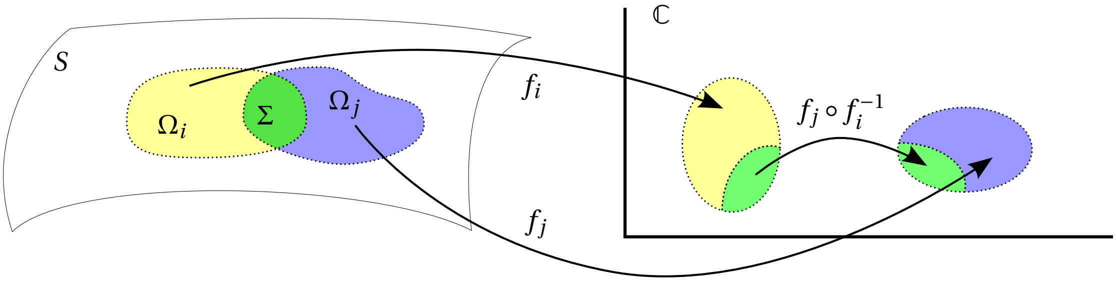


Figure 1.1: Transition map between two charts of a Riemann surface.

---

[1] Note that several equivalent definitions exist.

Definition 2. *Let S and S' be* Riemann surfaces. *If an homeomorphism $g : S \to S'$ is such that for each pair of charts (Ω, f) and (Ω', f') of S, respectively S', the mapping*

$$f' \circ g \circ f^{-1} : f(\Omega \cap g^{-1}(\Omega')) \subset \mathbb{C} \to f'(\Omega') \subset \mathbb{C}$$

is a conformal *mapping, then g is said to be a* conformal mapping *between S and S'.*

Definition 3. *Let S and S' be* Riemann surfaces. *If there exists a* conformal mapping $g : S \to S'$ *between them, with an inverse $g^{-1} : S' \to S$ that is also conformal, then the surfaces S and S' are said to be* conformally equivalent*, or sometimes simply* equivalent.

It will later turn out that *Riemann surfaces* can be seen as *metric spaces* and that conformal mappings between surfaces are *isometries* between them.

1.1 Examples

Obviously, any open subset of $\mathbb{C}$ (or $\mathbb{R}^2$) is a *Riemann surface.* For the following part of this document, we will focus on *closed* surfaces, that are compact surfaces without border.

Sphere A "classroom" example is the *two dimensional sphere*

$$\mathbb{S}^2 = \{(x, y, z) \in \mathbb{R}^3 \quad | \quad x^2 + y^2 + z^2 = 1\}.$$

To see it as a *Riemann surface*, we can provide the atlas $A = \{(\Omega_1, \pi_1), (\Omega_2, \pi_2)\}$, where

$$\begin{aligned} \Omega_1 =& \mathbb{S}^2 \setminus \{(0,0,1)\} \\ \Omega_2 =& \mathbb{S}^2 \setminus \{(0,0,-1)\} \end{aligned}$$

and

$$\begin{array}{rccc} \pi_1 : & \Omega_1 & \to & \mathbb{C} \\ & (x,y,z) & \mapsto & \frac{1}{1-z}(x + iy) \end{array}$$

$$\begin{array}{rccc} \pi_2 : & \Omega_2 & \to & \mathbb{C} \\ & (x,y,z) & \mapsto & \frac{1}{1+z}(x - iy) \end{array}$$

The function π_1 is called the *stereographic projection*, while π_2 is a modified version of π_1, the purpose of the modification being that the transition map between the two *charts* preserves orientation.

One can check by a direct calculation that the inverses of these functions are as follows, where $w = \Re(w) + i\Im(w)$ designates a complex number and $\bar{w} = \Re(w) - i\Im(w)$ is the complex czonjugate.

$$\begin{array}{rccc} \pi_1^{-1}: & \mathbb{C} & \rightarrow & \Omega_1 \\ & w & \mapsto & \left(\dfrac{2\Re(w)}{1+w\bar{w}}, \dfrac{2\Im(w)}{1+w\bar{w}}, \dfrac{-1+w\bar{w}}{1+w\bar{w}}\right) \end{array}$$

$$\begin{array}{rccc} \pi_2^{-1}: & \mathbb{C} & \rightarrow & \Omega_2 \\ & w & \mapsto & \left(\dfrac{2\Re(w)}{1+w\bar{w}}, \dfrac{-2\Im(w)}{1+w\bar{w}}, \dfrac{1-w\bar{w}}{1+w\bar{w}}\right) \end{array}$$

Defining $\Sigma = \Omega_1 \cap \Omega_2 = \mathbb{S}^2 \setminus \{(0,0,1),(0,0,-1)\}$ a direct calculation yields

$$\begin{array}{rcccc} \pi_2 \circ \pi_1^{-1}\big|_{\pi_1(\Sigma)} & : & \pi_1(\Sigma) = \mathbb{C}^* & \rightarrow & \pi_2(\Sigma) = \mathbb{C}^* \\ & & w & \mapsto & 1/w \end{array}$$

which is an *holomorphic function.*

Riemann sphere Let us now define the following set $\hat{\mathbb{C}} = \mathbb{C} \cup \{\infty\}$, where ∞ is an additional point often called *infinity.*

We can endow $\hat{\mathbb{C}}$ with the *charts* $(\mathbb{C}, \mathrm{Id}_{\mathbb{C}})$ and

$$\left(\hat{\mathbb{C}} \setminus \{0\}, \begin{cases} z & \mapsto & 1/z \\ \infty & \mapsto & 0 \end{cases}\right).$$

This surface is *conformally equivalent* to the one obtained in the previous example, $\mathbb{S}^2$, by taking $\mathrm{Id}_{\mathbb{C}}$ as the *conformal mapping* represented in the *charts.* This is why the set $\hat{\mathbb{C}}$ is called the *Riemann sphere.*

Differentiable real 2-manifold The following theorem gives us a clue on how general the concept of *Riemann surface* is.

Theorem 1. *Let S be an* orientable smooth manifold *of dimension 2, given with a* maximal differentiable atlas A^{diff}. *Then,* A^{diff} *possesses a* sub-atlas $A^{conf} \subset A^{diff}$ *such that all transition maps are holomorphic.*

According to this theorem, e.g. all smooth orientable surfaces S in $\mathbb{R}^3$ may be seen as *Riemann surfaces.* There is actually a stronger theorem stating that for any such surface an atlas exists such that for all charts (Ω_i, f_i) the mapping $f_i : \Omega_i \rightarrow D_i \subset \mathbb{R}^2 = \mathbb{C}$ is angle preserving, like in the example of the sphere. This is the theorem on *isothermal coordinates.* A proof may be found in [12, chapter 9, addendum 1].

2 Uniformization theorem

In this chapter, we give, without proof, a theorem called *Uniformization theorem*, conjectured by Felix Klein and Henry Poincaré around 1882 and proven by H.Poincaré and Paul Koebe in 1907.

This very deep theorem serves as the foundation for a *classification* of all *closed Riemann surfaces* [1].

Theorem 2 (Uniformization)**.** *Every simply connected Riemann surface is conformally equivalent to one of the following Riemann surfaces*

- *the* Riemann sphere $\hat{\mathbb{C}} = \mathbb{C} \cup \{\infty\}$,
- *the* complex plane $\mathbb{C}$,
- *the* unit disk $\mathbb{D} = \{z \in \mathbb{C} \mid |z| < 1\}$.

Before we look at what this implies for closed surfaces we review some definitions which will be frequently used in this document.

Definition 4. *Let S be a topological space (e.g. an open subset of $\mathbb{R}^n$ or a Riemann surface). Two parametrized curves (or* arcs*) $\gamma, \delta : [a,b] \to S$ with the same endpoints: $\gamma(a) = \delta(a)$, $\gamma(b) = \delta(b)$, are called* homotopic *if γ can be continuously deformed into δ with both endpoints kept fixed.*

Formally, this means that there exists a continuous mapping (the "homotopy") $H : [0,1] \times [a,b] \to S$, with the following properties

$$H(0,t) = \gamma(t), \quad H(1,t) = \delta(t), \quad \forall t \in [a,b],$$
$$H(s,a) = \gamma(a), \quad H(s,b) = \gamma(b), \quad \forall s \in [a,b].$$

---

[1] To be accurate, there are a number of different theorems in the literature for which the name *Uniformization theorem* is used.
According to some of them, the theorem as we present it here would be presented as a *corollary.*

Definition 5. *A topological space S is* simply connected *if for any couple of points $p, q \in S$ the following holds.*

(1) There exists an arc $\gamma : [a,b] \to S$ such that $\gamma(a) = p$, $\gamma(b) = q$.

(2) Two arcs $\gamma, \delta : [a,b] \to S$ with $\gamma(a) = \delta(a) = p$ and $\gamma(b) = \delta(b) = q$, are always homotopic.

For example, the sphere $\mathbb{S}^2$, the complex plane $\mathbb{C}$ and the unit disk $\mathbb{D}$ are simply connected.

An important topological fact is that any Riemann surface S has a *universal covering*.

Definition 6. *A continuous mapping $\Pi : S' \to S$ from a topological space S' to a topological space S is a* covering *(or a* covering map*) if for any point $p \in S$ an open neighbourhood $U_p \subset S$ can be found that has the following "good" property.*

The inverse image $\Pi^{-1}(U_p)$ *in S' is the union*

$$\Pi^{-1}(U_p) = \bigcup_i U_{p_i}$$

of pairwise disjoint open neighbourhoods $U_{p_i} \subset S'$ (finitely or infinitely many) such that for each i the restriction *of Π to U_{p_i}*

$$\Pi|_{U_{p_i}} : U_{p_i} \to U_p$$

is a homeomorphism.

In the case where S' is simply connected the space S' is called a universal covering *of S and Π is called a* universal covering map.

The points p_i in this definition are called the *lifts* of the point p.

Using arguments from *algebraic topology* one proves that any Riemann surface S has a universal covering $\Pi : \tilde{S} \to S$. Then, with the "good" neighbourhoods and the conformal atlas of S it is possible to define a conformal atlas for $\tilde{S}$ such that $\Pi : \tilde{S} \to S$ is a conformal mapping. In other words:

Any Riemann surface has a universal Riemann covering surface.

Together with the covering surface comes the group of *covering transformations*. For this we provide a final definition.

Definition 7. *Let Σ be a simply connected Riemann surface. A group Γ of mappings from Σ to Σ is called a* Fuchsian group [2] *if it satisfies the following "axioms".*

---

[2] To be accurate, we should us the term "fixed point free Fuchsian group" or "Fuchsian group with free action".

1. *Any $\phi \in \Gamma$ is a* conformal transformation $\phi : \Sigma \to \Sigma$, *meaning that ϕ is a bijection and both ϕ and ϕ^{-1} are conformal mappings.*
2. *For any point $q \in \Sigma$ there exists a "good" open neighbourhood $U_q \subset \Sigma$ such that the images $\phi(U_q)$, $\phi \in \Gamma$ are pairwise disjoint.*

Translations An example is the group $\mathbb{Z} \oplus \tau\mathbb{Z}$, where $\tau \in \mathbb{C} \backslash \mathbb{R}$. Its members are the translations

$$\phi(z) = z + m + \tau n, \quad z \in \mathbb{C},$$

where $m, n \in \mathbb{Z}$.

If now a universal covering $\Pi : \tilde{S} \to S$ of a Riemann surface S is given, then there exists a unique *Fuchsian group* Γ with the following properties concerning lifts of points.

- Whenever $p', p'' \in \tilde{S}$ are two lifts of the same point $p \in S$, then we can find $\phi \in \Gamma$ such that $\phi(p') = p''$. (By property 2. of a Fuchsian group ϕ is unique.)
- For any $\psi \in \Gamma$: if p' is a lift of p then $\psi(p')$ is also a lift of p.

This particular Fuchsian group is called the *group of covering transformations* corresponding to the universal covering $\Pi : \tilde{S} \to S$.

In [7, Chapter1] it is explained how, conversely, for any Fuchsian group Γ acting on a simply connected Riemann surface Σ one can define a Riemann surface S such that the universal Riemann covering surface is Σ and the corresponding group of covering transformations is Γ. This surface is unique up to conformal equivalence. It is called the *quotient surface* and is written

$$S = \Sigma/\Gamma. \tag{2.0.1}$$

Finally, a result of complex function theory says that the only fixed point free conformal transformation of $\hat{\mathbb{C}} = \mathbb{C} \cup \infty$ is the identity, and the only fixed point free conformal transformations of $\mathbb{C}$ are the translations $\phi(z) = z + a$, $z \in \mathbb{C}$, where $a \in \mathbb{C}$.

From these facts we can deduce the following classification.

Corollary 1 (Classification of *closed* Riemann surfaces)**.** *Every closed* Riemann surface *is conformally equivalent to one of the following surfaces.*

- *The* Riemann sphere $\hat{\mathbb{C}} = \mathbb{C} \cup \infty$.
- *A* flat torus $\mathbb{C}/(\mathbb{Z} \oplus \tau\mathbb{Z})$ *where* $\tau \in \mathbb{C} \setminus \mathbb{R}$.
- *The quotient* $\mathbb{D}/\Gamma$ *of the* unit disk $\mathbb{D}$ *by the free action of a Fuchsian group on* $\mathbb{D}$.

We will now discuss how to treat the *geometrical* aspect of these models.

Riemann sphere The Riemann sphere $\hat{\mathbb{C}}$ is an object of *Euler characteristic* $\chi = 2$, thus using the *Gauss-Bonnet* theorem, if we are able to find a *Riemannian metric* of constant *Gauss curvature* on it, it must be positive.

The solution consists of taking the following Riemannian metric at point $z = u + iv \in \mathbb{C}$.

$$ds^2 = \frac{4}{(1+u^2+v^2)^2}(du^2+dv^2)$$

It is a metric that has everywhere Gauss curvature 1.

This space is called *a parabolic surface.*

Flat torus The flat tori simply receive the euclidean flat metric $ds^2 = du^2 + dv^2$ of $\mathbb{C}$ seen as a real 2-manifold.

These spaces are called *elliptic surfaces.*

Hyperbolic surfaces The last case, $\mathbb{D}/\Gamma$ is by far the richest. The surfaces corresponding to that case are called *hyperbolic surfaces.*

The rest of this document is devoted to their study.

We describe in chapter 3 how we can naturally endow them with a *Riemannian metric* of constant curvature -1.

3 Hyperbolic geometry

As we have pointed out in Chapter 2, the richest type of *Riemann surfaces* are the hyperbolic ones, i.e. surfaces that can be written as $S = \mathbb{D}/\Gamma$.

In this chapter, we show how we can endow the surface S with a *metric space* structure that is compatible with it's complex structure.

In order to do that, in Section 3.1, we present how to present $\mathbb{D}$ as a metric space and we define a second space $\mathbb{H}$ which is isometric to $\mathbb{D}$ but is sometimes more suitable for certain illustrations as well as for computation.

Then, in Section 3.2, we present two equivalent models, $\mathrm{M}\mathbb{D}$ and $\mathrm{M}\mathbb{H}$ where points and geodesics can both be represented by certain *Möbius* transformations. These models are far more useful than the previous ones to solve computational geometrical questions.

3.1 The models $\mathbb{H}$ and $\mathbb{D}$

In Chapter 2 we defined $\mathbb{D}$ to be the set $\{z \in \mathbb{C} \mid |z| < 1\}$.

The goal is to provide a *metric* on $\mathbb{D}$ that is invariant under the *conformal transformations* of $\mathbb{D}$.

Möbius transformations Before doing that, we present a special case of complex functions called *Möbius transformations.*

Definition 8 (Möbius transformation)**.** *Let* $a, b, c, d \in \mathbb{C}$ *such that* $ad - bc \neq 0$ *. We define the following complex function* $f : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$.

$$f(z) = \begin{cases} \dfrac{az+b}{cz+d} & \text{if} \quad cz+d \neq 0 \\ \infty & \text{if} \quad cz+d = 0 \\ \frac{a}{c} & \text{if} \quad z = \infty \ \text{and} \ c \neq 0 \\ \infty & \text{if} \quad z = \infty \ \text{and} \ c = 0 \end{cases} \tag{3.1.1}$$

Such a function is called a Möbius transformation of $\hat{\mathbb{C}}$ *(see Section 1.1 for the definition of the* Riemann sphere $\hat{\mathbb{C}}$*).*

The set of all the Möbius transformations of $\hat{\mathbb{C}}$ *is denoted by* $Möb(\hat{\mathbb{C}})$.

Remark 1. *Since* Möbius transformations *are* rational functions*, they are* holomorphic.

Proposition 1. *The Mapping*

$$\begin{array}{cccc} \Psi : & GL_2(\mathbb{C}) & \to & Möb(\hat{\mathbb{C}}) \\ & \begin{pmatrix} a & b \\ c & d \end{pmatrix} & \mapsto & f \end{array}$$

where f *is the function defined in equation* (3.1.1) *is a* group homomorphism *between* $(GL_2(\mathbb{C}), \cdot)$*, the group of* invertible 2 by 2 complex matrices *with standard matrix multiplication and* $(Möb(\hat{\mathbb{C}}), \circ)$ *the group* $Möb(\hat{\mathbb{C}})$ *with function composition.*

Proof. It can be shown by a direct calculation that

$$\begin{array}{rcl} \forall A, B \in GL_2(\mathbb{C}), \forall z \in \hat{\mathbb{C}}, \quad \Psi(A) \circ \Psi(B)(z) & = & \Psi(A \cdot B)(z) \\ \Psi(\mathrm{Id}_2)(z) & = & z. \end{array}$$

□

Let us now take the matrix $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in GL_2(\mathbb{C})$. It *corresponds* to a uniquely defined *Möbius* *transformation* acting on $\hat{\mathbb{C}}$. We use the following notation to express it.

$$M[z] := \Psi(M)(z)$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}[z] = \frac{az+b}{cz+d}.$$

Since $\forall \lambda \in \mathbb{C}^*, M \in \mathrm{GL}_2(\mathbb{C}), z \in \hat{\mathbb{C}}$,

$$(\lambda M)[z] = \frac{\lambda az + \lambda b}{\lambda cz + \lambda d} = M[z],$$

seeing $\mathrm{GL}_2(\mathbb{C})$ as a $\mathbb{C}$-*vector space*, Ψ is invariant by scalar multiplication.

That is why we work with $\mathrm{PGL}_2(\mathbb{C}) = \mathrm{GL}_2(\mathbb{C})/(\mathbb{C}^*\mathrm{Id}_2)$, the *projective linear group*.

By using the *first isomorphism theorem*, one can show that $\mathrm{PGL}_2(\mathbb{C}) \simeq \mathrm{Möb}(\hat{\mathbb{C}})$. The function induced by Ψ is an *isomorphism of groups* between $\mathrm{PGL}_2(\mathbb{C})$ and $\mathrm{Möb}(\hat{\mathbb{C}})$.

3.1.1 The hyperbolic disk $\mathbb{D}$

In Chapter 2, we have pointed out that $\mathbb{D}$ is the *universal covering* [1] of the surfaces we are going to study.

Möbius transformations of $\mathbb{D}$

Definition 9. *We define Möb*($\mathbb{D}$) *to be the subgroup of Möb*($\hat{\mathbb{C}}$) *leaving the set* $\mathbb{D}$ *invariant, i.e.*

$$\mathit{Möb}(\mathbb{D}) = \{f \in \mathit{Möb}(\hat{\mathbb{C}}) \quad | \quad f(\mathbb{D}) = \mathbb{D}\}$$

This definition leads to the very important theorem.

Theorem 3. *The* biholomorphic *functions* $f : \mathbb{D} \to \mathbb{D}$ *are exactly the* Möbius transformations *of the form*

$$f(z) = e^{i\phi}\frac{z-c}{1-\bar{c}z}$$

with $\phi \in \mathbb{R}$ *and* $c \in \mathbb{C}$.

Proof. You can find a proof of this in [11]. □

---

[1] At least up to *conformal* transformation.

Corollary 2. *Let*

$$M\mathbb{D} = \left\{ \begin{pmatrix} a & b \\ \bar{b} & \bar{a} \end{pmatrix} \in \mathbb{GL}_2(\mathbb{C}) \quad | \quad a, b \in \mathbb{C} \right\},$$

be the subgroup of $\mathbb{GL}_2(\mathbb{C})$ *composed of the matrices of the form* $\begin{pmatrix} a & b \\ \bar{b} & \bar{a} \end{pmatrix}$ *then,*

$$M\ddot{o}b(\mathbb{D}) = \Psi(M\mathbb{D}).$$

Proof. Taking $e^{i\phi} = \frac{a}{\bar{a}}$ and $c = -\frac{b}{a}$, we obtain

$$\begin{pmatrix} a & b \\ \bar{b} & \bar{a} \end{pmatrix}[z] = \frac{az+b}{\bar{b}z+\bar{a}} = \frac{a}{\bar{a}} \cdot \frac{z+\frac{b}{a}}{1+\frac{\bar{b}}{\bar{a}}z} = e^{i\phi}\frac{z-c}{1-\bar{c}z}.$$

□

Metric on $\mathbb{D}$ The last point of this section is to provide $\mathbb{D}$ with a *metric* $d_{\mathbb{D}} : \mathbb{D} \times \mathbb{D} \to \mathbb{R}_+$ invariant under the transformations of M$\mathbb{D}$.

Definition 10. *We first define the* Riemannian metric $\rho_{\mathbb{D}}$ *on* $\mathbb{D}$ *given by the following* metric tensor

$$\rho_{\mathbb{D}}(z) = \frac{4}{(1-z\bar{z})^2} Id_2$$

Definition 11. *Let us define* $d_{\mathbb{D}}$ *to be*

$$\begin{array}{rccc} d_{\mathbb{D}}: & \mathbb{D} \times \mathbb{D} & \to & \mathbb{R}_+ \\ & (z_1, z_2) & \mapsto & \cosh^{-1}(1+\delta(z1, z2)) \end{array}$$

where

$$\begin{array}{rccc} \delta_{\mathbb{D}}: & \mathbb{D} \times \mathbb{D} & \to & \mathbb{R}_+ \\ & (z_1, z_2) & \mapsto & 2\dfrac{|z_2 - z_1|^2}{(1-|z_1|^2)(1-|z_2|^2)}. \end{array}$$

The function $d_{\mathbb{D}}$ *is called the* hyperbolic distance *of* $\mathbb{D}$*.*

It can be shown that the distance defined by the Riemannian metric tensor $\rho_{\mathbb{D}}$ coincides with $d_{\mathbb{D}}$.

This metric has a constant *Gauss curvature* -1 and is *unique* in the sense of the following theorem.

Theorem 4. *If $d : \mathbb{D} \times \mathbb{D} \to \mathbb{R}_+$ is a* continuous metric *invariant under the transformations of $M\mathbb{D}$, then $d = \lambda d_{\mathbb{D}}$ for some $\lambda \in \mathbb{R}_+^*$. Furthermore, if the* Gauss curvature *of d (which is constant) is -1, then $d = d_{\mathbb{D}}$.*

Proof. You can find a proof of this in [14]. □

3.1.2 The hyperbolic half-plane $\mathbb{H}$

Another very important model of the hyperbolic geometry is

$$\mathbb{H} = \{z \in \mathbb{C} \quad | \quad \Im(z) > 0\}.$$

this set is called the *upper half-plane.*

We provide the following definitions and propositions to show that the $\mathbb{H}$ model is *conformally equivalent* to the *hyperbolic disk model* of the previous section.

Definition 12. *Let us define the matrices*

$$\Pi_{\mathbb{D}}^{\mathbb{H}} = \begin{pmatrix} i & i \\ -1 & 1 \end{pmatrix} \in GL_2(\mathbb{C}) \tag{3.1.2}$$

$$\Pi_{\mathbb{H}}^{\mathbb{D}} = \begin{pmatrix} 1 & -i \\ 1 & i \end{pmatrix} \in GL_2(\mathbb{C}) \tag{3.1.3}$$

These matrices have the following property.

Proposition 2. *The mapping $\pi_{\mathbb{D}}^{\mathbb{H}} := \Psi(\Pi_{\mathbb{D}}^{\mathbb{H}})$ is a* conformal bijection *from $\mathbb{D}$ to $\mathbb{H}$ whose inverse is $\pi_{\mathbb{H}}^{\mathbb{D}} := \Psi(\Pi_{\mathbb{H}}^{\mathbb{D}})$.*

Proof. A direct computation shows us that $\pi_{\mathbb{D}}^{\mathbb{H}}(\mathbb{D}) \subset \mathbb{H}$ and that $\pi_{\mathbb{H}}^{\mathbb{D}}(\mathbb{H}) \subset \mathbb{D}$. Finally, using Proposition 1, we see that

$$\pi_{\mathbb{D}}^{\mathbb{H}} \circ \pi_{\mathbb{H}}^{\mathbb{D}} = \Psi(\Pi_{\mathbb{D}}^{\mathbb{H}}) \circ \Psi(\Pi_{\mathbb{H}}^{\mathbb{D}}) = \Psi(\Pi_{\mathbb{D}}^{\mathbb{H}} \cdot \Pi_{\mathbb{H}}^{\mathbb{D}}) = \Psi(2i\mathrm{Id}_2) = \mathrm{Id}_{\hat{\mathbb{C}}}.$$

□

Möbius transformations of $\mathbb{H}$ By analogy to Definition 9, we set

Definition 13. *$Möb(\mathbb{H})$ is the subgroup of $Möb(\hat{\mathbb{C}})$ leaving the set $\mathbb{H}$ invariant, i.e.*

$$Möb(\mathbb{H}) = \{f \in Möb(\hat{\mathbb{C}}) \quad | \quad f(\mathbb{H}) = \mathbb{H}\}$$

Proposition 3. *Let* $M\mathbb{H} = \mathbb{GL}_2(\mathbb{R}) \subset \mathbb{GL}_2(\mathbb{C})$ *be the subgroup of* $\mathbb{GL}_2(\mathbb{C})$ *consisting of the* real 2×2 *invertible matrices. Then,*

$$M\ddot{o}b(\mathbb{H}) = \Psi(M\mathbb{H}).$$

Proof. Using Propositions 1 and 2 and the *conjugation principle*, it is possible to show through a calculation that

$$\mathrm{M}\mathbb{H} = \Pi_{\mathbb{D}}^{\mathbb{H}} \mathrm{M}\mathbb{D} \Pi_{\mathbb{H}}^{\mathbb{D}} = \mathbb{GL}_2(\mathbb{R}).$$

□

Metric on $\mathbb{H}$ The *metric* $d_{\mathbb{D}}$ of $\mathbb{D}$ induces the *metric* $d_{\mathbb{H}} : \mathbb{H} \times \mathbb{H} \to \mathbb{R}_+$ where

$$d_{\mathbb{H}}(z_1, z_2) = d_{\mathbb{D}}(\pi_{\mathbb{H}}^{\mathbb{D}}(z_1), \pi_{\mathbb{H}}^{\mathbb{D}}(z_2)) = \cosh^{-1}(1 + \frac{|z_2 - z_1|^2}{2\Im(z_1)\Im(z_2)}). \tag{3.1.4}$$

By construction, $\pi_{\mathbb{D}}^{\mathbb{H}}$ is an isometry between the *metric spaces* $(\mathbb{D}, d_{\mathbb{D}})$ and $(\mathbb{H}, d_{\mathbb{H}})$.

3.2 $\mathbb{MH}$, $\mathbb{MD}$ models

In the implementations of our algorithms the numerical evaluations are carried out by matrices that represent Möbius transformations. More precisely, we use the tools that have been developed in [1, Lecture 4]. We call this calculus the $\mathbb{MH}$ model when we work with Möbius transformation that operate in $\mathbb{H}$ and the $\mathbb{MD}$ model when the Möbius transformations operate in $\mathbb{D}$.

For convenience we collect here a few definitions and formulas. For a full description of the calculus we refer the reader to the lectures in [1].

As in [1] we restrict ourselves to the $\mathbb{MH}$ model.

The first point is to look at rotations with rotational angle π, the so-called *half turns*. If $z = r + is \in \mathbb{H}$ is any point ($r \in \mathbb{R}$, $s > 0$), then the rotation in $\mathbb{H}$ of angle π and center z is given by the Möbius transformation whose matrix is

$$p = \frac{1}{s}\begin{pmatrix} -r & r^2+s^2 \\ -1 & r \end{pmatrix} \tag{3.2.1}$$

In the $\mathbb{MH}$ model the idea is that p *is* a point. Points are thus matrices $\in \mathrm{GL}_2(\mathbb{R})$ of determinant 1 and $\mathrm{Trace}(p) = 0$.

In a similar way, symmetries with respect to geodesics are represented by matrices of the form

$$A = \frac{1}{\sigma}\begin{pmatrix} \rho & -\rho^2+\sigma^2 \\ 1 & -\rho \end{pmatrix}, \quad (\sigma > 0, \rho \in \mathbb{R})$$

$$A = \begin{pmatrix} -1 & 2a \\ 0 & 1 \end{pmatrix}, \quad (a \in \mathbb{R}) \tag{3.2.2}$$

and in the $\mathbb{MH}$ model A *is* the geodesic. In [1] instead of “geodesic” the term “line” is used. Lines are matrices $\in \mathrm{GL}_2(\mathbb{R})$ of derminant -1 and $\mathrm{Trace}(A) = 0$.

The advantage of this point of view is that everything can be formulated in terms of matrices. For instance, the following formulas hold, in which $\cdot$ is the matrix product and tr is the *half trace*: $\mathrm{tr}(A) := \frac{1}{2}\mathrm{Trace}(A)$:

If p, q are points then their distance is given by

$$\cosh d(p,q) = \frac{|\mathrm{tr}(p\cdot q)|}{\sqrt{\mathrm{tr}(p^2)\mathrm{tr}(q^2)}} \tag{3.2.3}$$

If A and B are non-intersecting lines then their distance is given by

$$\cosh d(A,B) = \frac{|\operatorname{tr}(A\cdot B)|}{\sqrt{\operatorname{tr}(A^2)\operatorname{tr}(B^2)}} \tag{3.2.4}$$

If A and B are lines that intersect each other under the angle $\sphericalangle$ then this angle is given by

$$\cos\sphericalangle(A,B) = \frac{\operatorname{tr}(A\cdot B)}{\sqrt{\operatorname{tr}(A^2)\operatorname{tr}(B^2)}} \tag{3.2.5}$$

If p is a point and A a line then the *oriented distance* is given by

$$\sinh d(A,p) = \frac{\operatorname{tr}(A\cdot p)}{\sqrt{-\operatorname{tr}(A^2)\operatorname{tr}(p^2)}} \tag{3.2.6}$$

To “do geometry” the *wedge product* is very useful. For matrices X, Y it is defined as

$$X \wedge Y := \frac{1}{2}(X\cdot Y - Y\cdot X). \tag{3.2.7}$$

We then have the following geometric facts.

Proposition 4. *Let p, q be points and A, B lines. Then the following hold.*

(i) *If p, q are distinct points, then $p \wedge q$ is the line through p and q.*

(ii) *If p is a point and A a line, then $p \wedge A$ is the line through p perpendicular to A.*

(iii) *If A, B are lines and* $\det(A\wedge B) > 0$, *then $A \wedge B$ is the point incident with A and B.*

(iv) *If A, B are lines and* $\det(A\wedge B) < 0$, *then $A \wedge B$ is the line perpendicular to A and B.*

Finally we mention that the same can be carried out with matrices that represent half turns and geodesic symmetries in $\mathbb{D}$. The corresponding matrices are obtained from those of the $\mathrm{M}\mathbb{H}$ model by conjugating them with the matrices

$$\Pi_{\mathbb{D}}^{\mathbb{H}} = \begin{pmatrix} i & i \\ -1 & 1 \end{pmatrix}$$
$$\Pi_{\mathbb{H}}^{\mathbb{D}} = \begin{pmatrix} 1 & -i \\ 1 & i \end{pmatrix}$$

from the preceding Section.

3.3 Hyperbolic trigonometry

In this Section, we just reproduce a list of very useful *trigonometric* relations that have been taken "as they are" from the book [4].

These formulas are used later to calculate the geometrical properties of our surfaces.

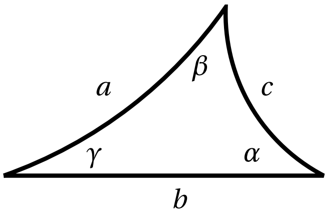
We give them without proof, they can be proven by an almost direct calculation, using the results of the $\mathbb{MH}$ model in Section 3.2.

3.3.1 Triangles

(i) $\cosh c = -\sinh a \sinh b \cos\gamma + \cosh a \cosh b$

(ii) $\cos\gamma = \sin\alpha \sin\beta \cosh c - \cos\alpha \cos\beta$

(iii) $\dfrac{\sinh a}{\sin\alpha} = \dfrac{\sinh b}{\sin\beta} = \dfrac{\sinh c}{\sin\gamma}$



3.3.2 Right-angled triangles

(i) $\cosh c = \cosh a \cosh b$

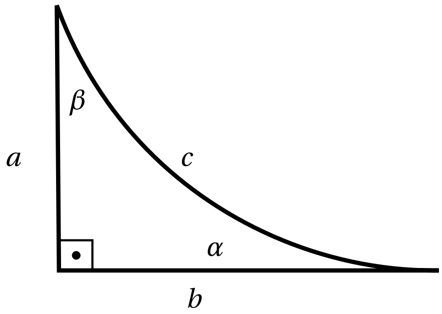
(ii) $\cosh c = \cot\alpha \cot\beta$

(iii) $\sinh a = \sin\alpha \sinh c$

(iv) $\sinh a = \cot\beta \tanh b$

(v) $\cos\alpha = \cosh a \sin\beta$

(vi) $\cos\alpha = \tanh b \coth c$



3.3.3 Trirectangles

(i) $\cos\phi = \sinh a \sinh b$

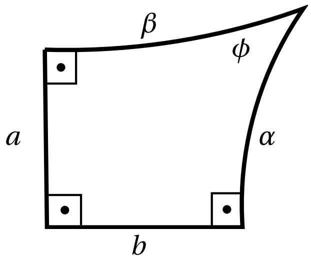
(ii) $\cos\phi = \tanh\alpha \tanh\beta$

(iii) $\cosh a = \cosh\alpha \sin\phi$

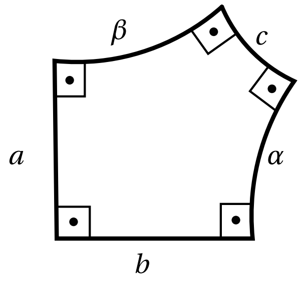
(iv) $\cosh a = \tanh\beta \coth b$

(v) $\sinh\alpha = \sinh a \cosh\beta$

(vi) $\sinh\alpha = \coth b \cot\phi$



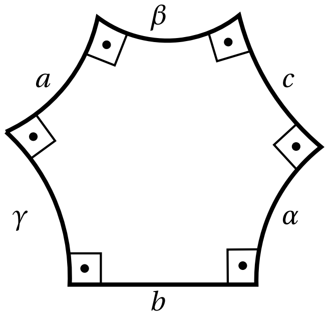
3.3.4 Right-angled pentagons



(i) $\cosh c = \sinh a \sinh b$

(ii) $\cosh c = \coth \alpha \coth \beta$

3.3.5 Right-angled hexagons

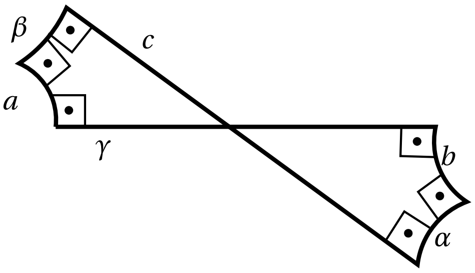


(i) $\cosh c = \sinh a \sinh b \cosh \gamma - \cosh a \cosh b$

(ii) $\dfrac{\sinh a}{\sinh \alpha} = \dfrac{\sinh b}{\sinh \beta} = \dfrac{\sinh c}{\sinh \gamma}$

(iii) $\coth \alpha \sinh \gamma = \cosh \gamma \cosh b - \coth a \sinh b$

3.3.6 Crossed right-angled hexagons



(i) $\cosh c = \sinh a \sinh b \cosh \gamma + \cosh a \cosh b$

3.4 Trace bounding

In this section we present a theorem giving a lower bound on the *trace* of an hyperbolic Möbius transformation.

Definition 14. *Let the functions* $V, H : \mathbb{R} \to M\mathbb{H}$ *and the Möbius transformation* $O \in M\mathbb{H}$ *be*

$$V : t \mapsto V^t := \begin{pmatrix} e^{t/2} & 0 \\ 0 & e^{-t/2} \end{pmatrix}$$

$$H : t \mapsto H^t := \begin{pmatrix} \cosh(t/2) & \sinh(t/2) \\ \sinh(t/2) & \cosh(t/2) \end{pmatrix}$$

$$O := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$$

V^t *and* H^t *are respectively called the* vertical *and the* horizontal *Möbius transformation of distance* t*.*

O *is called the* rotation *around* i *of angle* $\frac{\pi}{2}$ *with positive orientation.*

Remark 2. *One sees that for all* $a, b, t \in \mathbb{R}$*:*

- $\det(V^t) = \det(H^t) = \det(O) = 1$
- $tr(V^t) = tr(H^t) = \cosh(\frac{t}{2})$
  $tr(O) = 1/\sqrt{2}$
- $V^0 = H^0 = Id_{M\mathbb{H}}$
- $V^a \cdot V^b = V^{(a+b)}$
  $H^a \cdot H^b = H^{(a+b)}$
- $V^t = OH^tO^{-1}$
  $H^t = OV^{-t}O^{-1}$

Definition 15. *Let* $V^*, H^* \in M\mathbb{H}$ *be the following matrices*

$$V^* = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$H^* = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

and let Let $\tilde{V}, \tilde{H} \subset \mathbb{H}$ *be the following sets*

$$\tilde{V} = \{z \in \mathbb{H} \mid \Re(z) = 0\}$$

$$\tilde{H} = \{z \in \mathbb{H} |\ |z| = 1\}$$

V^* and H^* represent symmetries of axes $\tilde{V}$ and $\tilde{H}$.

$\tilde{V}$ and $\tilde{H}$ are the axes of V^t, respectively $H^t, \forall t \in \mathbb{R}$ and are called vertical *and* horizontal axes.

Definition 16. *Let $\tilde{V}^-, \tilde{V}^+, \tilde{H}^-, \tilde{H}^+ \subset \mathbb{H}$ be the following sets.*

$$\tilde{V}^- = \{z \in \mathbb{H} | \Re(z) < 0\}$$

$$\tilde{V}^+ = \{z \in \mathbb{H} | \Re(z) > 0\}$$

$$\tilde{H}^- = \{z \in \mathbb{H} | |z| < 1\}$$

$$\tilde{H}^+ = \{z \in \mathbb{H} | |z| > 1\}$$

They are the half-planes of $\mathbb{H}$ *of border $\tilde{V}$, respectively $\tilde{H}$.*

Theorem 5 (Minimal distance)**.** *Let $M \in M\mathbb{H}$ be a transformation of the form*

$$M = \prod_{i=1}^{n} X_i$$

where the X_i are either of the form $X_i = H^{h_j}$ or of the form $X_i = V^{v_k}$ for some $h_j, v_k \in \mathbb{R}$.

Let $\{h_j\}_{j=1}^{p} \subset \mathbb{R}$ and $\{v_k\}_{k=1}^{q} \subset \mathbb{R}$ be respectively the powers *of the H and the V that appear in the product M.*

If $v_k \geq 0, \quad \forall k = 1, 2, ..., q$

then, the following inequalities

- $trM \geq tr\prod_{j=1}^{p} H^{h_j} = trH^{\sum_j h_j} \geq 1$
- $trM \geq tr\prod_{k=1}^{q} V^{v_k} = trV^{\sum_k v_k} \geq 1$

are true.

These inequalities are strict unless $v_k = 0, \forall k = 1, 2, ..., q$ respectively $h_j = 0, \forall j = 1, 2, ..., p$.

Before giving the proof of this theorem, we will present some other properties that will be useful for the proof.

Remark 3. *Let $a, b, \alpha, \beta \in \mathbb{R}$ such that $a \geq |\alpha|$ and $b \geq |\beta|$. Then*

$$ab + \alpha\beta \geq |a\beta + \alpha b|$$

and

$$ab - \alpha\beta \geq |a\beta - \alpha b|$$

with equalities iif $a = |\alpha|$ *or* $b = |\beta|$.

Proof. (Remark 3) Firstly, $ab \pm \alpha\beta \geq ab - |\alpha||\beta| \geq a(b - |\beta|) \geq 0$.

And we also have

$$(ab+\alpha\beta)^2 - (a\beta+\alpha b)^2 = (ab-\alpha\beta)^2 - (a\beta-\alpha b)^2$$

$$= a^2b^2 + \alpha^2\beta^2 - a^2\beta^2 - \alpha^2b^2 = \underbrace{(a^2-\alpha^2)}_{\geq 0}\underbrace{(b^2-\beta^2)}_{\geq 0} \geq 0.$$

with equality iif $(a^2 = \alpha^2)$ or $(b^2 = \beta^2)$. □

Remark 4. *Let* $M = \prod_{i=1}^{n} X_i$, $\{h_j\}_{j=1}^{p} \subset \mathbb{R}$ *and* $\{v_k\}_{k=1}^{q} \subset \mathbb{R}$ *be as in the assumptions of Theorem 5.*

Then, there are isometries $S, T \in M\mathbb{H}$ *that can be written as*

$$S = \prod_{i=1}^{n'} H^{h_j^S} \cdot V^{v_j^S} = H^{h_1^S} \cdot V^{v_1^S} \cdot H^{h_2^S} \cdot V^{v_2^S} \cdot \ldots \cdot H^{h_{n'}^S} \cdot V^{v_{n'}^S}$$

$$T = \prod_{i=1}^{n'} V^{v_j^T} \cdot H^{h_j^T} = V^{v_1^T} \cdot H^{h_1^T} \cdot V^{v_2^T} \cdot H^{h_2^T} \cdot \ldots \cdot V^{v_{n'}^T} \cdot H^{h_{n'}^T}$$

for some $n' \in \mathbb{N}$ *and for some* $h_j^S, v_j^S, h_j^T, v_j^T \in \mathbb{R}$ *such that*

$$\begin{array}{rcl} trM & = & trS = trT \\ \sum_{i=1}^{p} h_i & = & \sum_{j=1}^{n'} h_j^S = \sum_{j=1}^{n'} h_j^T \\ \sum_{i=1}^{q} v_i & = & \sum_{j=1}^{n'} v_j^S = \sum_{j=1}^{n'} v_j^T \\ v_i \geq 0 \quad \forall i = 1, \ldots, n & \Rightarrow & v_j^S \geq 0 \quad \forall j = 1, \ldots, n' \\ & \Rightarrow & v_j^T \geq 0 \quad \forall j = 1, \ldots, n'. \end{array}$$

Proof. (Remark 4) This fact comes from the properties of Remark 2 and the fact that the trace is invariant under conjugation. □

Lemma 1. *Let* $M \in SL_2^+(\mathbb{R})$ *be such that* M *can be written as a product of a finite number of elements of the set*

$$\{V^a \in M\mathbb{H} \quad | \quad a \in \mathbb{R}_+^*\} \cup \{H^b \in M\mathbb{H} \quad | \quad b \in \mathbb{R}\}.$$

Then,

$$M \in \tilde{G} := \left\{ \begin{pmatrix} m_1 & m_2 \\ m_3 & m_4 \end{pmatrix} \in SL_2^+(\mathbb{R}) \mid m_1 \pm m_i \geq |m_j \pm m_k|, \forall (i,j,k) \in Perm(2,3,4) \right\}.$$

(The inequality must be read as "$m_1 + m_i \geq |m_j + m_k|$ and $m_1 - m_i \geq |m_j - m_k|$".)

Proof. Lemma 1
First, by using the fact that $e^a > |e^{-a}|$ and $\cosh(b) > |\sinh(b)|$ $\forall a \in \mathbb{R}_+, b \in \mathbb{R}$, we can check that $V^a, H^b \in \tilde{G}$, $\forall a \in \mathbb{R}_+^*, b \in \mathbb{R}$.

Now let $M, N \in \tilde{G}$ and $R = M \cdot N$. We want to show that $R \in \tilde{G}$.

By using Remark 3 a computation gives us

$$(r_1 + r_2) + (r_3 + r_4) = \overbrace{(m_1 + m_3)}^{a \geq |\alpha|} \overbrace{(n_1 + n_2)}^{b \geq |\beta|} + \overbrace{(m_2 + m_4)}^{\alpha} \overbrace{(n_3 + n_4)}^{\beta} \geq |...| \geq 0$$

$$(r_1 + r_2) - (r_3 + r_4) = \overbrace{(m_1 - m_3)}^{a \geq |\alpha|} \overbrace{(n_1 + n_2)}^{b \geq |\beta|} + \overbrace{(m_2 - m_4)}^{\alpha} \overbrace{(n_3 + n_4)}^{\beta} \geq |...| \geq 0$$

from which $r_1 + r_2 \geq |r_3 + r_4|$.

The other cases $r_1 \pm r_i \geq |r_j \pm r_k|$ can be shown by exactly the same kind of argument.

□

Proposition 5 (Words caracterisation)**.**
Let $M = \begin{pmatrix} m_1 & m_2 \\ m_3 & m_4 \end{pmatrix} \in SL_2^+(\mathbb{R})$ *be an hyperbolic transformation. Then the following statements are equivalent.*

(a) *M can be written as a product of a finite number of elements of the set* $\{V^a \in M\mathbb{H} \mid a \in \mathbb{R}_+^*\} \cup \{H^b \in M\mathbb{H} \mid b \in \mathbb{R}\}$, *with at least one element of the set* $\{V^a \in M\mathbb{H} \mid a \in \mathbb{R}_+^*\}$.

(b)

$$M \in G := \left\{ \begin{pmatrix} m_1 & m_2 \\ m_3 & m_4 \end{pmatrix} \in SL_2^+(\mathbb{R}) \mid \quad m_1 \pm m_4 > |m_2 \pm m_3| \right\}$$

which also implies that $m_1 > |m_i|, \forall i = 2,3,4$.

(c) $\exists! \alpha, \beta, \gamma \in \mathbb{R}$ *with* $\beta > 0$ *such that* $M = H^\alpha \cdot V^\beta \cdot H^\gamma$.

(d) $M \in SL_2^+(\mathbb{R})$ *represents an* hyperbolic *transformation whose axis cuts the axis of* H^1 *and* $\tilde{H}^- \subsetneq M(\tilde{H}^-)$

Proof. (Proposition 5) First, let us show that $M \in G$ indeed implies that $m_1 > |m_i|, \forall i = 2,3,4$.

We have $m_1 + m_4 > |...| \geq 0$ and $m_1 - m_4 > |...| \geq 0$ and then $m_1 > |m_4|$.
Since $m_1 + m_4 > m_2 + m_3$ and $m_1 - m_4 > m_2 - m_3$, $2m_1 > 2m_2$ from where $m_1 > m_2$. And $m_1 + m_4 > -m_2 - m_3$ and $m_1 - m_4 > -m_2 + m_3$, $2m_1 > -2m_2$ from where $m_1 > -m_2$ and $m_1 > |m_2|$. The case $m_1 > |m_3|$ is identical.

To finish the proof, we will show that (b)⇒(c)⇒(a)⇒(b). And then we will finally show that (a,b,c)⇒(d) and (d)⇒(c).

(b)⇒(c)

A rather long computation shows that $\forall \alpha, \beta, \gamma \in \mathbb{R}$, we have

$$R = H^\alpha V^\beta H^\gamma \Rightarrow \tag{3.4.1}$$

$$r_1 + r_4 = 2\cosh(\beta/2)\cosh((\alpha+\gamma)/2)$$

$$r_3 + r_2 = 2\cosh(\beta/2)\sinh((\alpha+\gamma)/2)$$

$$r_1 - r_4 = 2\sinh(\beta/2)\cosh((\alpha-\gamma)/2) \tag{3.4.2}$$

$$r_3 - r_2 = 2\sinh(\beta/2)\sinh((\alpha-\gamma)/2)$$

Existence of α, β, γ:

Now let be $N \in G$. Since, by assumption, we have $n_1 \pm n_4 > |n_3 \pm n_2|$, the following inequality $-1 < \frac{n_3 \pm n_2}{n_1 \pm n_4} < 1$ holds. That allow us to write

$$\alpha = \tanh^{-1}\left(\frac{n_3 + n_2}{n_1 + n_4}\right) + \tanh^{-1}\left(\frac{n_3 - n_2}{n_1 - n_4}\right) \tag{3.4.3}$$

$$\gamma = \tanh^{-1}\left(\frac{n_3 + n_2}{n_1 + n_4}\right) - \tanh^{-1}\left(\frac{n_3 - n_2}{n_1 - n_4}\right) \tag{3.4.4}$$

and

$$\beta = 2\tanh^{-1}\left(\sqrt{\frac{(n_1 - n_4)^2 - (n_3 - n_2)^2}{(n_1 + n_4)^2 - (n_3 + n_2)^2}}\right) > 0. \tag{3.4.5}$$

In the definition of β we use that $n_1 n_4 - n_2 n_3 = \det N > 0$ in order to show that the numerator is smaller than the denominator.

Now let be $R = H^\alpha V^\beta H^\gamma$ with the α, β, γ defined just above. We want to prove that $R = N$.

Even if it is perhaps possible, a direct calculation would be very complex to achieve, thus we will prove this fact in a different manner.

We see that

$$\frac{n_3+n_2}{n_1+n_4} = \tanh((\alpha+\gamma)/2) = \frac{2\cosh(\beta/2)\sinh((\alpha+\gamma)/2)}{2\cosh(\beta/2)\cosh((\alpha+\gamma)/2)} = \frac{r_3+r_2}{r_1+r_4}$$

From which we deduce that $\exists\lambda \in \mathbb{R}^*$ such that

$$r_1 + r_4 = \lambda(n_1+n_4) \tag{3.4.6}$$

$$r_3 + r_2 = \lambda(n_3+n_2)$$

By Lemma 1, we know that $r_1 + r_4 \geq 0$ and by hypothesis $N \in G$, we have $n_1 + n_4 > 0$ and then we know that $\lambda > 0$.

In the same way

$$\frac{n_3-n_2}{n_1-n_4} = \tanh((\alpha-\gamma)/2) = \frac{2\sinh(\beta/2)\sinh((\alpha-\gamma)/2)}{2\sinh(\beta/2)\cosh((\alpha-\gamma)/2)} = \frac{r_3-r_2}{r_1-r_4}$$

$\Rightarrow \exists\mu \in \mathbb{R}^*$ such that

$$r_1 - r_4 = \mu(n_1-n_4) \tag{3.4.7}$$

$$r_3 - r_2 = \mu(n_3-n_2)$$

$$\mu > 0$$

.

Finally, we calculate that

$$0 < \frac{(n_1-n_4)^2-(n_3-n_2)^2}{(n_1+n_4)^2-(n_3+n_2)^2} = \tanh^2(\beta/2)$$

$$= \frac{4\sinh^2(\beta/2)(\cosh^2((\alpha-\gamma)/2)-\sinh^2((\alpha-\gamma)/2))}{4\cosh^2(\beta/2)(\cosh^2((\alpha+\gamma)/2)-\sinh^2((\alpha+\gamma)/2))}$$

$$= \frac{(r_1-r_4)^2-(r_3-r_2)^2}{(r_1+r_4)^2-(r_3+r_2)^2}$$

$$= \frac{\mu^2}{\lambda^2}\frac{(n_1-n_4)^2-(n_3-n_2)^2}{(n_1+n_4)^2-(n_3+n_2)^2}.$$

Since we know that $\lambda, \mu > 0$, we must have $\lambda = \mu$.

By adding and subtracting the equations (3.4.6) and (3.4.7), we see that $R = \lambda N$.

Finally, since we know that by assumption $N \in \mathrm{SL}_2^+(\mathbb{R})$, we have

$$1 = \det(H^\alpha V^\beta H^\gamma) = \det R = \lambda^2 \det N = \lambda^2$$

and, since $\lambda > 0$, we have $\lambda = 1$ which finally proves that $N = R = H^\alpha V^\beta H^\gamma$ for the α, β, γ defined in (3.4.3)-(3.4.5) which concludes the existence part.

Uniqueness of α, β, γ:
Let us assume we have $\alpha, \beta, \gamma, \alpha', \beta', \gamma' \in \mathbb{R}$, $\beta, \beta' > 0$ such that
$R := H^\alpha V^\beta H^\gamma = H^{\alpha'} V^{\beta'} H^{\gamma'}$.

We have to show that $\alpha = \alpha'$, $\beta = \beta'$ and $\gamma = \gamma'$.

Starting with the equations (3.4.1), the same computation as before gives us that

$$\frac{n_3 + n_2}{n_1 + n_4} = \tanh((\alpha + \gamma)/2) = \tanh((\alpha' + \gamma')/2) \Rightarrow \alpha + \gamma = \alpha' + \gamma'$$

$$\frac{n_3 - n_2}{n_1 - n_4} = \tanh((\alpha - \gamma)/2) = \tanh((\alpha' - \gamma')/2) \Rightarrow \alpha - \gamma = \alpha' - \gamma'$$

$$\Rightarrow \alpha = \alpha' \text{ and } \gamma = \gamma'$$

and

$$\frac{(n_1 - n_4)^2 - (n_3 - n_2)^2}{(n_1 + n_4)^2 - (n_3 + n_2)^2} = \tanh^2(\beta/2) = \tanh^2(\beta'/2)$$

Since $\beta, \beta' > 0$, this implies that $\beta = \beta'$ which ends the proof of (b)⇒(c).

(c)⇒(a) Trivial.

(a)⇒(b)
To prove that (a)⇒(b), we will show the following statements, $\forall A \in G, \forall a \in \mathbb{R}_+^*, \forall b \in \mathbb{R}$

1) $V^a \in G$
2) $A \cdot V^a \in G$
3) $V^a \cdot A \in G$
4) $A \cdot H^b \in G$
5) $H^b \cdot A \in G$.

Since, by assumption, $A \in G$, the point (b)⇒(c) and Lemma 1 imply that $A \in \tilde{G}$. (So we have $G \subset \tilde{G}$, but we need this fact only here.)

Now we give the proof item per item using Lemma 1.

1) $V^a \in G, \forall a \in \mathbb{R}_+^*$:

$$(V^a)_1 \pm (V^a)_4 = e^{a/2} \pm e^{-a/2} > 0 = |(V^a)_2 \pm (V^a)_3|$$

2) $R := A \cdot V^a \in G, \forall a \in \mathbb{R}^*_+, A \in G$: Using that $a_1 > |a_i|$, $i = 2, 3, 4$ (see the beginning of the proof) we get

$$(r_1 + r_4) + (r_2 + r_3) = e^{a/2}\overbrace{(a_1 + a_3)}^{\geq |a_2+a_4|} + e^{-a/2}(a_2 + a_4) \geq \overbrace{(a_1 + a_3)}^{>0}\overbrace{(e^{a/2} - e^{-a/2})}^{>0} > 0$$

$$(r_1 + r_4) - (r_2 + r_3) = e^{a/2}\overbrace{(a_1 - a_3)}^{\geq |a_2-a_4|} - e^{-a/2}(a_2 - a_4) \geq \overbrace{(a_1 - a_3)}^{>0}\overbrace{(e^{a/2} - e^{-a/2})}^{>0} > 0$$

which implies that $r_1 + r_4 > |r_2 + r_3|$. The case $r_1 - r_4 > |r_2 - r_3|$ is similar.

3) $R := A \cdot H^b \in G, \forall b \in \mathbb{R}, A \in G$:

$$(r_1 + r_4) + (r_2 + r_3) = \overbrace{e^{b/2}}^{>0}\overbrace{((a_1 + a_4) + (a_2 + a_3))}^{>0} > 0$$

$$(r_1 + r_4) - (r_2 + r_3) = e^{-b/2}((a_1 + a_4) - (a_2 + a_3)) > 0$$

which implies that $r_1 + r_4 > |r_2 + r_3|$. The case $r_1 - r_4 > |r_2 - r_3|$ is similar.

4) $R := V^a \cdot A \in G, \forall a \in \mathbb{R}^*_+, A \in G$: Similar to case 2).

5) $R := H^b \cdot A \in G, \forall b \in \mathbb{R}, A \in G$: Similar to case 3).

This concludes the proof that (a), (b) and (c) are equivalent. Now let us prove that

(a,b,c)⇒(d)

Let $M \in G$ be a Möbius transformation.

Firstly we have to show that M is *hyperbolic* which is equivalent to check that $\mathrm{tr}M > 1$. By point (c), $M = H^\alpha V^\beta H^\gamma$ for some $\alpha, \beta, \gamma \in \mathbb{R}^*$, $\beta > 0$. Then, by equation (3.4.1), we see that

$$\mathrm{tr}M = \frac{m_1 + m_4}{2} = \overbrace{\cosh(\beta/2)}^{>1}\overbrace{\cosh((\alpha+\gamma)/2)}^{\geq 1} > 1.$$

Now we have to show that the axis of M cuts the axis $\tilde{H}$. By using the theorem 4.7 of [1] page 48, we can conclude that this is the case if and only if $\det(M \wedge H^*) = \frac{1}{4}\det(MH^* - H^*M) > 0$. A computation tells us that, after simplification

$$\det(M \wedge H^*) = \overbrace{((m1 - m4) + (m2 - m3))}^{>0}\overbrace{((m1 - m4) - (m2 - m3))}^{>0} > 0.$$

Finally, we have to prove that $\tilde{H}^- \subsetneq M(\tilde{H}^-)$.

By point (c), let $\alpha, \beta, \gamma \in \mathbb{R}$ with $\beta > 0$ be such that $M = H^\alpha V^\beta H^\gamma$.

Since $\tilde{H}$ is the axis of H^α, $H^\alpha(\tilde{H}) = \tilde{H}$ and then $H^\alpha(\tilde{H}^-)$ is an half plane of border $\tilde{H}$ it follows that $H^\alpha(\tilde{H}^-) \in \{\tilde{H}^-, \tilde{H}^+\}$. We also have that $0 \in \overline{\tilde{H}^-}$ and $H^\alpha(0) = \frac{\sinh(\alpha)}{\cosh(\alpha)}$ and then $|H^\alpha(0)| < 1$ from where $H^\alpha(0) \in \overline{\tilde{H}^-}$ and then $H^\alpha(\tilde{H}^-) = \tilde{H}^-$. The same argument yields $H^\gamma(\tilde{H}^-) = \tilde{H}^-$.

Next, $\forall z, |z| < 1 \Leftrightarrow |V^{\beta}(z)| = e^{\beta}|z| < e^{\beta}$ and then, since $\beta > 0$, we have

$$\{z \in \mathbb{H} \mid |z| < 1\} = \tilde{H}^{-} \subsetneq V^{\beta}(\tilde{H}^{-}) = \{z \in \mathbb{H} \mid |z| < e^{\beta}\}.$$

This allow us to write the following.

$$\tilde{H}^{-} \subsetneq V^{\beta}(\tilde{H}^{-}) = V^{\beta}(H^{\gamma}(\tilde{H}^{-}))$$

$$\Rightarrow H^{\alpha}(\tilde{H}^{-}) \subsetneq H^{\alpha}(V^{\beta}(H^{\gamma}(\tilde{H}^{-})))$$

$$\Rightarrow \tilde{H}^{-} \subsetneq M(\tilde{H}^{-}).$$

The last step to do on the proof of the proposition is to show that

(d)⇒(c)

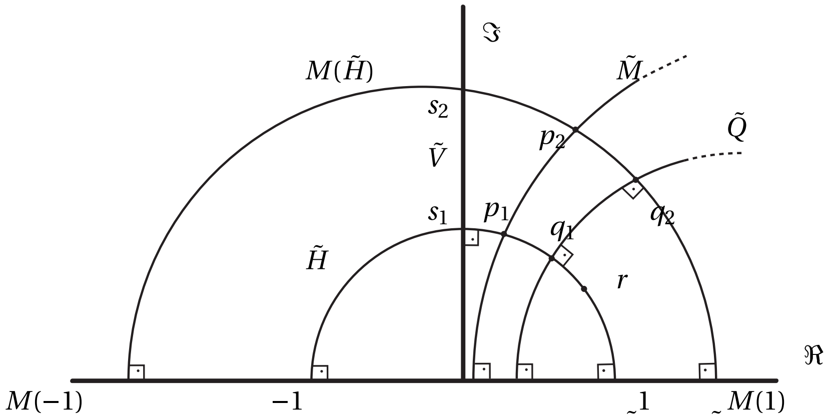


Figure 3.1: The hyperbolic plane with $\tilde{H}$ and $\tilde{M}$.

Let $p_1 \in \mathbb{H}$ be the intersection point of $\tilde{H}$ and $\tilde{M}$, the axis of M;
let $p_2 = M(p_1)$ be be the intersection point of $M(\tilde{H})$ and $\tilde{M}$. Since $\tilde{H}^{-} \subsetneq M(\tilde{H}^{-})$, $\tilde{H} \cap M(\tilde{H}) = \emptyset$. And since $\tilde{M}$ cuts $\tilde{H}$, the endpoints $-1, 1 \in \overline{\mathbb{H}}$ of $\tilde{H}$ are not in $\tilde{M}$ and then -1 and 1 are not endpoints of $M(\tilde{H})$. From that, we deduce that $\exists \tilde{D} \subset \mathbb{H}, \exists q_1, q_2 \in \tilde{D}$ a geodesic and two points such that $\tilde{D} \perp_{q_1} \tilde{H}$ and $\tilde{D} \perp_{q_2} M(\tilde{H})$.

Let $Q \in \mathrm{SL}_2^{+}(\mathbb{R})$ be the unique *hyperbolic transformation* with axis $\tilde{D}$ such that $Q(q_1) = q_2$.

Since $\tilde{H} \perp_{q_1} \tilde{D}$, $Q(\tilde{H}) \perp_{Q(q_1)} \overbrace{Q(\tilde{D})}^{=\tilde{D}}$ and then
$Q(\tilde{H}), M(\tilde{H}) \perp_{q_2} \tilde{D}$ from where $Q(\tilde{H}) = M(\tilde{H})$.

Let $s_1 = i \in \tilde{H} \cap \tilde{V}$ and let $S \in \mathrm{SL}_2^{+}(\mathbb{R})$ be the unique *hyperbolic transformation* with axis $\tilde{H}$ such that $S(q_1) = s_1$. Let $\alpha \in \mathbb{R}$ be the unique real number such that $S = H^{\alpha}$. Let $s_2 = S(q_2)$.

Since $\tilde{D} \perp_{q_1} \tilde{H}$ from where $S(\tilde{D}) \perp_{s_1} S(\tilde{H}) = \tilde{H}$ and $\tilde{V} \perp_{s_1} S(\tilde{H})$ we obtain that $S(\tilde{D}) = \tilde{V}$ and that $s_2 \in \tilde{V}$.

Then we can take $\beta \in \mathbb{R}$ to be the unique real number such that $V^{\beta}(s_1) = (s_2)$.

Since $q_2 \in M(\tilde{H})$, $q_2 \in \tilde{H}^+$ and then $s_2 = S(q_2) \in S(\tilde{H}^+) = \tilde{H}^+$. From where $|s_2| = |V^\beta(s1)| = |ie^\beta| > 1 \Rightarrow \beta > 0$.

We will now prove that $Q = H^{-\alpha}V^\beta H^\alpha$. Since both of these transformations are *hyperbolic*, to do that we will show that $q_2 = Q(q_1) = H^{-\alpha}V^\beta H^\alpha(q_1)$ and that $\tilde{D} = \text{axis}(Q) = \text{axis}(H^{-\alpha}V^\beta H^\alpha)$. We have

$$H^{-\alpha}V^\beta H^\alpha(q_1) = H^{-\alpha}V^\beta(s_1) = H^{-\alpha}(s_2) = (H^\alpha)^{-1}(s_2) = q_2 = Q(q_1)$$

and, using the *conjugation principle* we have that

$$\text{axis}(H^{-\alpha}V^\beta H^\alpha) = H^{-\alpha}(\text{axis}(V^\beta)) = H^{-\alpha}(\tilde{V}) = \tilde{D} = \text{axis}(Q).$$

This finishes the proof that $Q = H^{-\alpha}V^\beta H^\alpha$.

Now let $r = M^{-1}(q_2)$. Since $q_2 \in M(\tilde{H})$, $r \in \tilde{H}$. Let then be $\delta \in \mathbb{R}$ the unique real number such that $H^\delta(r) = q_1$.

We want to prove that $M = QH^\delta$. Since both of these transformations are *hyperbolic* this follows from

$$QH^\delta(r) = Q(q_1) = q_2 = M(r)$$

and

$$QH^\delta(\tilde{H}) = Q(\tilde{H}) = M(\tilde{H}).$$

Hence, $M = H^{-\alpha}V^\beta H^{\alpha+\delta}$ with $\beta > 0$ which proves (c).

This concludes the proof of the proposition. □

We can now give the proof of the Theorem 5.

Proof. (Theorem 5) Let M be a product of some H^{h_i} and some V^{v_j} like in the assumptions of Theorem 5. We will now separate the proof into three different parts.

Part 0

If $v_k = 0$, $\forall k = 1, ..., q$, then, by Remark 2, $M = H^{\sum_j h_j}$ and then

$$\text{tr}M = \text{tr}\prod_{j=1}^{p} H^{h_j} = \text{tr}H^{\sum_j h_j}.$$

If $h_j = 0$, $\forall j = 1, ..., p$, then, by Remark 2, $M = V^{\sum_j v_k}$ and then

$$\text{tr}M = \text{tr}\prod_{k=1}^{q} V^{v_k} = \text{tr}V^{\sum_k v_k}.$$

For the next part, using Remark 4, we may assume, without loss of generality that M is of the form

$$M = V^{v_1} \cdot H^{h_1} \cdot V^{v_2} \cdot H^{h_2} \cdot \ldots \cdot V^{v_n} \cdot H^{h_n} \tag{3.4.8}$$

with $v_i > 0, h_i \neq 0, \quad \forall i = 1, \ldots, n$ for a given *integer* $n \geq 1$.

Part 1

In this part, assuming that M is of the form (3.4.8), we want to show that

$$\mathrm{tr} M > \mathrm{tr} H^{\sum_i h_i}. \tag{3.4.9}$$

Let $R = \prod_{i=1}^{n-1} V^{v_i} \cdot H^{h_i}$ such that $M = R \cdot V^{v_n} H^{h_n}$.

Since the *trace* is invariant under conjugaison,

$$\mathrm{tr} M = \mathrm{tr}(R \cdot V^{v_n} \cdot H^{h_n}) = \mathrm{tr}(\underbrace{H^{h_n} \cdot R}_{:=S} \cdot V^{v_n}) = \mathrm{tr}(S \cdot V^{v_n})$$

We are going to prove the inequality (3.4.9) by induction over n.

Initial step $n = 1$:

then

$$\mathrm{tr} M = \mathrm{tr}(V^{v_1} \cdot H^{h_1}) = \cosh(\frac{v_1}{2}) \cosh(\frac{h_1}{2}) > \cosh(\frac{h_1}{2}) = \mathrm{tr} H^{h_1}.$$

Induction:

Suppose that the inequality (3.4.9) holds for all M' of the form $M' = \prod_{i=1}^{n-1} V^{v'_i} \cdot H^{h'_i}$ with $v'_i > 0$. We want to show that the inequality (3.4.9) holds for M. Using the lemma 1 on S, we see that $s_1 \geq |s_4|$ and then

$$\mathrm{tr} M - \mathrm{tr} S = \mathrm{tr}(S \cdot V^{v_n}) - \mathrm{tr}(S) =$$

$$\frac{1}{2}(s_1 \underbrace{(e^{v_n/2} - 1)}_{>0} + s_4 \underbrace{(e^{-v_n/2} - 1)}_{<0}) \geq \frac{1}{2}(s_1 (e^{v_n/2} - 1) + \underbrace{s_1}_{\geq |s_4|} \underbrace{(e^{-v_n/2} - 1)}_{<0})$$

$$= s_1 (\cosh(v_{n+1}/2) - 1) \geq 0.$$

Finally, by induction hypothesis on the M' defined in the next equation, we have that

$$\mathrm{tr} S = \mathrm{tr}(H^{h_n} \cdot R) = \mathrm{tr}(R \cdot H^{h_n}) = \underbrace{V^{v_1} \cdot H^{h_1} \cdots V^{v_{n-1}} \cdot H^{(h_{n-1} + h_n)}}_{=:M'} > \mathrm{tr} H^{\sum_{i=1}^{n} h_i}$$

and then

$$\mathrm{tr} M \geq \mathrm{tr} S > \mathrm{tr} H^{\sum_i h_i}.$$

Part 2

In this last part, assuming that M is of the form $M = H^{h_1} \cdot V^{v_1} \cdot H^{h_2} \cdot V^{v_2} \cdot \ldots \cdot H^{h_n} \cdot V^{v_n}$, we want to show that

$$\mathrm{tr} M > \mathrm{tr} V^{\sum_i v_i}. \tag{3.4.10}$$

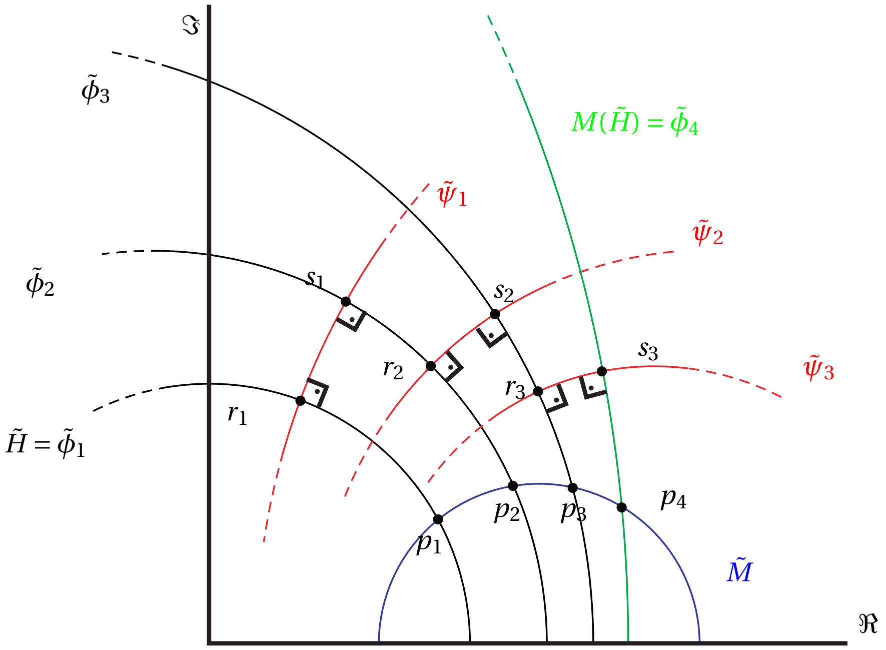


Figure 3.2: The hyperbolic plane with $\tilde{\phi}_i$ and $\tilde{\psi}_i$ in a case where $n = 3$. For better visibility the axis of $\tilde{M}$ has been drawn lower than its actual position.

Let $S_0 = \mathrm{Id}_2$, we now define recursively the following *Möbius transformations.*

$$\forall i = 1, \ldots, n:$$

$$\phi_i := S_{i-1} \cdot H^{h_i} \cdot S_{i-1}^{-1} \tag{3.4.11}$$

$$R_i := \phi_i \cdot S_{i-1} = S_{i-1} \cdot H^{h_i} \tag{3.4.12}$$

$$\psi_i := R_i \cdot V^{v_i} \cdot R_i^{-1} \tag{3.4.13}$$

$$S_i := \psi_i \cdot R_i = R_i \cdot V^{v_i} = S_{i-1} \cdot H^{h_i} \cdot V^{v_i} \tag{3.4.14}$$

By using the Proposition 5, we see that $\tilde{M}$, the *axis* of M, cuts $\tilde{H}$ and we define $p_1 \in \mathbb{H}$ to be the intersection point.

By (3.4.14), we can see that $S_n = M$ by a simple recursion argument.

Let us define $p_{n+1} = M(p_1) \in M(\tilde{H}) =: \tilde{\phi}_{n+1}$.

Let $s_0 = i \in \mathbb{H}$, we now define recursively the following *points* of $\mathbb{H}$.

$$\begin{aligned} &\forall i = 1,\dots,n: \\ &\qquad r_i := \phi_i(s_{i-1}) \in \mathbb{H} \\ &\qquad s_i := \psi_i(r_i) \in \mathbb{H} \end{aligned}$$

We can show by recursion that $r_i = R_i(s_0)$ and that $s_i = S_i(s_0)$.

Finally, since ϕ_i and ψ_i are conjugates of *hyperbolic transformations,* they are themselves *hyperbolic transformations.* We then define $\tilde{\phi}_i \subset \mathbb{H}$ and $\tilde{\psi}_i \subset \mathbb{H}$ to be the *axes* of ϕ_i and respectively ψ_i.

Using equation (3.4.11) and the *conjugation principle* $(\widetilde{YXY^{-1}}) = Y(\tilde{X})$, we see that $\forall i = 1,\dots n$,

$$\tilde{\phi}_i = S_{i-1}(\tilde{H}).$$

and by (3.4.12),

$$R_i(\tilde{H}) = S_{i-1} \cdot H^{h_i}(\tilde{H}) = S_{i-1}(\tilde{H}) = \tilde{\phi}_i.$$

Similarly, by using (3.4.13) and (3.4.14), we find that

$$\tilde{\psi}_i = R_i(\tilde{V})$$

and

$$S_i(\tilde{V}) = R_i \cdot V^{v_i}(\tilde{V}) = R_i(\tilde{V}) = \tilde{\psi}_i.$$

Then, since $\forall i = 1,\dots,n$, R_i, and S_i are isometries, we have

$$\tilde{V} \perp_{s_0} \tilde{H} \Rightarrow R_i(\tilde{V}) \perp_{R_i(s_0)} R_i(\tilde{H}) \Rightarrow \tilde{\psi}_i \perp_{r_i} \tilde{\phi}_i$$

$$\tilde{V} \perp_{s_0} \tilde{H} \Rightarrow S_i(\tilde{V}) \perp_{S_i(s_0)} S_i(\tilde{H}) \Rightarrow \tilde{\psi}_i \perp_{s_i} \tilde{\phi}_{i+1}.$$

For $i = 1,\dots,n$, let us define the following half planes of border $\tilde{\phi}_i$

$$\tilde{\phi}_i^- = S_{i-1}(\tilde{H}^-) \subset \mathbb{H}$$

$$\tilde{\phi}_i^+ = S_{i-1}(\tilde{H}^+) \subset \mathbb{H}.$$

We will prove by induction that for $i = 1, \dots, n$

$$\tilde{\phi}_i^- \underset{\neq}{\subsetneq} \tilde{\phi}_{i+1}^-$$

and

$$\tilde{\phi}_{i+1}^+ \underset{\neq}{\subsetneq} \tilde{\phi}_i^+.$$

By using Proposition 5, we see that $\tilde{\phi}_i$ and $\tilde{\phi}_{i+1}$ are *strongly parallel* and

$$\tilde{H}^- \underset{\neq}{\subsetneq} H^{h_i} V^{v_i}(\tilde{H}^-).$$

Then by (3.4.14),

$$\tilde{\phi}_i^- = S_{i-1}(\tilde{H}^-) \underset{\neq}{\subsetneq} S_{i-1} H^{h_i} V^{v_i}(\tilde{H}^-) = S_i(\tilde{H}^-) = \tilde{\phi}_{i+1}^-$$

and

$$\overline{\tilde{\phi}_i^+} = \mathbb{H} \setminus \tilde{\phi}_i^- \underset{\neq}{\supsetneq} \mathbb{H} \setminus \tilde{\phi}_{i+1} = \overline{\tilde{\phi}_{i+1}^+}$$

and since $\tilde{\phi}_{i+1} \subset \tilde{\phi}_i$,

$$\tilde{\phi}_{i+1}^+ \underset{\neq}{\subsetneq} \tilde{\phi}_i^+.$$

Now, $p_1 \in \tilde{H}$, and so for $i = 2, \dots, n+1, \quad p_1 \in \tilde{\phi}_i^-$. And since $p_{n+1} \in M(\tilde{H}) = S_n(\tilde{H}) = \tilde{\phi}_{n+1}$, we have for $i = 1, \dots, n, \quad p_{n+1} \in \tilde{\phi}_i^+$.

Since $p_0, p_{n+1} \in \tilde{M}$, by the definition of *half-planes,*$\forall i = 2, \dots, n$, the geodesic $\tilde{M}$ cuts $\tilde{\phi}_i$. We define the intersection points to be $p_i := \tilde{M} \cap \tilde{\phi}_i$.

Since for $i = 2, \dots, n$, the *open geodesic segment* $]p_{i-1}, p_i[\subset \tilde{\phi}_i^-$ whereas the *open geodesic segment* $]p_i, p_{i+1}[\subset \tilde{\phi}_i^+$, then all the segments are disjoint

$$\forall i, j \in \{1, \dots, n\}, i \neq j \quad]p_i, p_{i+1}[\cap]p_j, p_{j+1}[= \emptyset.$$

And since the p_i are aligned and $]p_1, p_{n+1}[= \coprod_{i=1}^n]p_1, p_{i+1}[$, we have that

$$\operatorname{dist}(p_1, p_{n+1}) = \sum_{i=1}^n \operatorname{dist}(p_i, p_{i+1}).$$

But, since $p_i \in \tilde{\phi}_i, p_{i+1} \in \tilde{\phi}_{i+1}$ and since $\tilde{\psi}_i \perp_{r_i} \tilde{\phi}_i$ and $\tilde{\psi}_i \perp_{s_i} \tilde{\phi}_{i+1}$, we have, by theorem (2.24) of [1], that $\operatorname{dist}(p_1, p_{i+1}) \geq \operatorname{dist}(r_i, s_i)$ and then

$$\operatorname{dist}(p_1, p_{n+1}) \geq \sum_{i=1}^{n} \operatorname{dist}(r_i, s_i).$$

Now we show that the inequality is strict.
$\operatorname{dist}(p_1, p_{n+1}) = \sum_{i=1}^{n} \operatorname{dist}(r_i, s_i) \Leftrightarrow p_i = r_i = s_{i-1}, \forall i = 1, \ldots, n+1$, but since by assumption, $h_i \neq 0$, $\operatorname{dist}(p_1, p_{n+1}) > \sum_{i=1}^{n} \operatorname{dist}(r_i, s_i)$.

Finally, since

$$\operatorname{dist}(r_i, s_i) = \operatorname{dist}(R_i(s_0), \psi_i R_i(s_0)) = \operatorname{dist}(s_0, R_i^{-1} \psi_i R_i(s_0)) = \operatorname{dist}(s_0, V^{v_i}(s_0)),$$

we may write, using theorem (2.24) of [1],

$$2\cosh^{-1}(\underbrace{\operatorname{tr} M}_{>1}) = \operatorname{dist}(p_1, p_{n+1}) > \sum_{i=1}^{n} \operatorname{dist}(r_i, s_i) = \sum_{i=1}^{n} \operatorname{dist}(s_0, V^{v_i}(s_0))$$

$$= \operatorname{dist}(s_0, V^{\sum_i v_i}(s_0)) = 2\cosh^{-1}(\underbrace{\operatorname{tr} V^{\sum_i v_i}}_{>1})$$

and then

$$\operatorname{tr} M > \operatorname{tr} V^{\sum_i V_i} > 1.$$

This concludes the proof of the theorem.

□

Computational approach Part II

In this part, we will find a description of Riemann surfaces that is well suited for computation. Therefore, we will be able to describe certain properties of surfaces, such as the *systole*, fundamental domain, etc. that can be calculated *algorithmically*.

Choice of the language. Since the nature of these problems is to a large extent combinatorial, with sometimes some subtle stop conditions, we have to use a suitable language to express it.

It appears that the most suitable way to express these concepts is the so called *Oriented Object Programming*[2] (*OOP*) programming paradigm. Most of the the time, in this document, we will try to be quite language-independent by using *UML class diagrams* [3].

Instead of using pure *pseudocode*, we will use the `Python` language to give explicit implementations of certain algorithms.

We have chosen `Python` as the language of reference of this section, rather than a faster but lower-level language like `C++` to keep the focus on the didactic aspect rather than implementation details.

Objects and class diagrams. It is not the point here to recapitulate the whole theory of *objects*. The reader is supposed to be familiar with the concepts of *class*, *inheritance*, *method*, etc.

If this is not the case, the reader will find more explanations about these concepts in the books [5], [6].

What we want to keep in mind is that we will have to define a *class* to describe each kind of concept (mathematical, geometrical, purely abstract) that we want to model.

A *class* possesses two main kinds of informations: it's *attributes* and it's *methods*. The *attributes* are the data necessary to describe a given object of this *class* and the *methods* are some functions that can act on these data.

For the sake of clarity we use the following typeset to speak about classes: **ClassExample**, and the following typeset for variables and methods : `variableExample`.

---

[2] For a description of the concept see [5].

[3] *Unified Modeling Language*: is a standard to represent *object oriented programing* concepts in an unified implementation-independent way.

We present in Figure 3.3 an example of *class diagram* to clarify our notations.

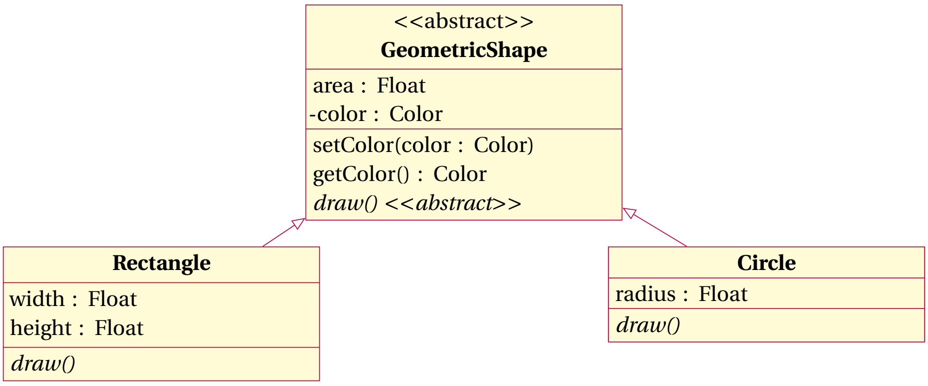


Figure 3.3: An example of *class diagram*.

The example diagram contains 3 classes: **GeometricShape**, **Rectangle** and **Circle**.

Here **GeometricShape** is the *base class* of the classes **Rectangle** and **Circle**.

GeometricShape possesses:

- An *attribute* called `area` which is of type **Float**.
- A *method* called `setColor` which takes one argument `color` of type **Color**, modifies it'internal state and returs nothing.
- A *method* called `getColor` which takes no argument and returns an object of type **Color**.
- A *virtual method* (written in italic) called `draw` which takes no argument and returns nothing. Furthermore this method is marked as *abstract*. It means that this method has no implementation and is meant to be overloaded. A class that contains *abstract* methods is an *abstract class*.

A `Python` code implementing these three classes would look like this.

```python
class GeometricShape:
  def __init__(self):#Constructor
    self.area = 0.0
    self.__color = Color("black")
  def setColor(self, color):
    self.__color = color
  def getColor(self):
    return self.__color
  def draw(self): abstractMethod

class Rectangle(GeometricShape):
  def __init__(self, width, height):#Constructor
    GeometricShape.__init__(self)#Calls the parent's constructor.
    self.width = width
    self.height = height
  def draw(self):
    outputDevice.drawRectangle(0,0,width,height)# pseudo implementation

class Rectangle(GeometricShape):
  def __init__(self, radius):#Constructor
    GeometricShape.__init__(self)#Calls the parent's constructor.
    self.radius = radius
  def draw(self):
    outputDevice.drawCircle(0,0,radius)# pseudo implementation
```

4 Surface modeling

In this chapter we will introduce a way to implement a model for the concept of a Riemann surface that will be suitable to explain algorithms to solve geometrical questions about the surface.

According to the notations of part I, let $S = \mathbb{H}/\Gamma$, with $\Gamma \subset \text{Möb}(\mathbb{H})$ it's Fuchsian group, be a *Riemann surface*.

Our goal being to have a geometric description of the surface, it appears that the better starting point is to suppose that we have a *marked surface*. In other words, that a Fenchel-Nielsen parametrization of S is already known (see chap 4.1 on page 48).

We will see in section 7.1 how to deduce a *set of generators* of the Fuchsian group Γ of the surface from these informations.

4.1 Fenchel-Nielsen parameters modeling

We consider that it is known that any *hyperbolic compact surface* of *genus g* can be described [1] by a *graph of order* 3 with

- $2(g-1)$ nodes corresponding to *Y -pieces*.
- $3(g-1)$ edges corresponding to *pants gluing*. Each edge of the graph is associated with two real numbers (l,τ) that are called the *length* and the *twist* of the *gluing*.

A quite *natural* way to model this programmatically is to associate a *name* to each *Y -piece* in the form of a **String** that will later serve as an *identifier* of this *Y -piece*.

The *edges* of the graph are then fully described by a **tuple** of length 5 of the form

```
("Y1name","Y2name", L, t, "edgeName"="")
```

where `L` and `t` are represented by (**float** *floating point real numbers*), the last argument being optional.

`L` is called the *length parameter* and is strictly positive `L>0.0`.

`t` is called the *twist parameter* and, in our approach can be any *floating point number*, positive or negative.

Note that it is also possible to overcome the issues arising from usage of *floating point numbers* in the case of *algebraic surfaces*, by using exact numbers representations, but it is outside the scope of this document.

The *Fenchel-Nielsen parameters* of the surface we are modeling can simply be given as a list of $3(g-1)$ tuples as described above.

Examples Here are three examples of valid *Fenchel-Nielsen graphs* and their `Python` representation.

---

[1] This description is not unique.

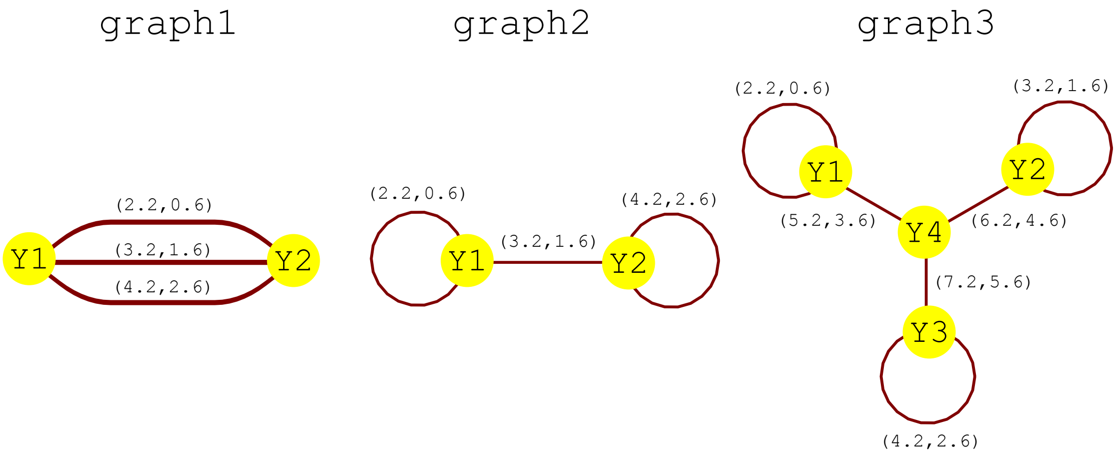


Figure 4.1: Examples of *Fenchel-Nielsen* graphs.

In `Python` language, these graphs are implemented in the following way[2]

```python
#The graph of a surface of genus 2.
graph1 = [("Y1", "Y2", 2.2, 0.6), ("Y1", "Y2", 3.2, 1.6), ("Y1", "Y2", 4.2, 2.6)]

#The graph of a surface of genus 2.
graph2 = [("Y1", "Y1", 2.2, 0.6), ("Y2", "Y2", 4.2, 2.6), ("Y1", "Y2", 3.2, 1.6)]

#The graph of a surface of genus 3.
graph3 = [("Y1", "Y1", 2.2, 0.6), ("Y2", "Y2", 3.2, 1.6), ("Y3", "Y3", 4.2, 2.6),
    ("Y1", "Y4", 5.2, 3.6), ("Y2", "Y4", 6.2, 4.6), ("Y3", "Y4", 7.2, 5.6)]
```

Validity conditions A `graph` given in this form is a valid *Fenchel-Nielsen parametrization* of a surface if and only if it gives rise to a connected graph of order 3.

That means that every `"YName"` **String** identifier must appear **exactly** 3 times in the `graph`.

As a counter-example, the graph `invalidGraph` shown below is not valid since the *Y-piece* named `"Ylow"` appears only twice and the *Y-piece* named `"Yhigh"` appears 4 times.

```python
invalidGraph = [("Ylow", "Yhigh", 1.0, 0.0), ("Ylow", "Yhigh", 1.0, 0.0),
    ("Yhigh", "Yhigh", 1.0, 0.0)]
```

---

[2] In this example, we chose not to give *names* to the edges.

4.2 Surface sub-objects

The point of this chapter is to define a new *class* called **Surface**.

Any *object*[3] `surface` of the *type* **Surface** is aimed to be a valid representations of a given *marked surface* $S = \mathbb{H}/\Gamma$.

To ensure our model catches the useful geometrical properties of S that we will need in the algorithms of Chapter 6, we shall define five important *types* of subsets of S. These are, the Y*-pieces* (who are the subsets of S that are bounded by three simple closed geodesic, i.e. elements of $\tau_{0,3}$), the *collars*, the *borders*, the *roads* and the *slots*[4] of S.

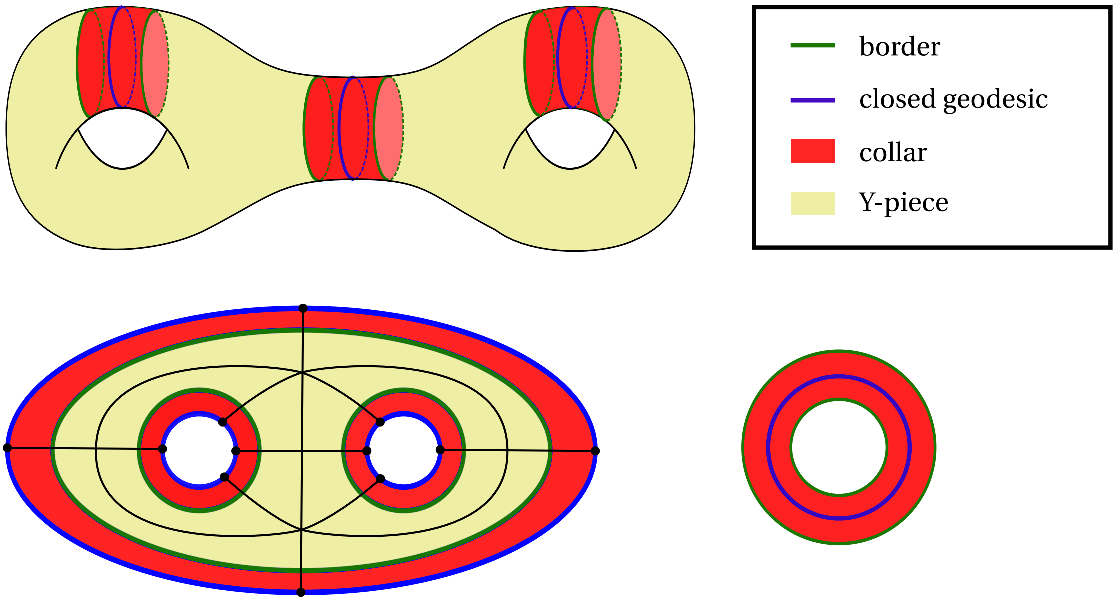


Figure 4.2: A surface of *genus* 2 with 3 collars and 2 Y*-pieces*, together with a topological representation of a Y*-piece* and a *collar*.

Collars Before giving more details about its implementation we provide a more accurate description of how we understand the *gluing* term of the previous section 4.1.

Definition 17. *Let* $S = \mathbb{H}/\Gamma$ *be an* hyperbolic surface, $\epsilon \in \mathbb{R}_+^*$ *and* $\gamma \subset S$ *a* closed geodesic.

We define the following subset of S

$$Coll_\gamma^\epsilon(S) = \left\{ s \in S \quad | \quad d_S(\gamma, s) \leq \epsilon \right\}$$

where d_S *is the* hyperbolic metric *of* S *and* $d_S(\gamma, s) := \inf_{p \in \gamma} d_S(p, s)$.

---

[3] Also called *instance of the* **Surface** *class.*

[4] Note that the terms *border, road* and *slot* are not *official*, they are just invented for practical purpose and clarity within this document.

The set $Coll_{\gamma}^{\epsilon}(S)$ *is called* ϵ-collar around γ *in* S.

This definition leads us to the very well known and important Lemma due to L.Keen.

Lemma 2 (L.Keen)**.** *Let* $Coll_{\gamma}^{\epsilon}(S)$ *be an* ϵ-collar *around a simple closed geodesic* $\gamma \in S$.

If

$$\sinh(\epsilon)\sinh(\frac{1}{2}l(\gamma)) \leq 1,$$

then $Coll_{\gamma}^{\epsilon}(S)$ *is* diffeomorphic *to an* annulus $A \subset \mathbb{C}$ *i.e. a set of the form*

$$A = \{z \in \mathbb{C} \quad | \quad |z| \in [1,2]\} \tag{4.2.1}$$

Due to this Lemma's hypothesis, we will always assume that we take sufficiently small ϵ and directly speak of *collar* if the ϵ is not relevant in the particular context. So, *collars* will always be diffeomorphic to annuli.

Borders

Definition 18. *Let* $\gamma \in S$ *be a simple closed geodesic,* $C \subset S$ *a* collar *around* γ, $A \subset \mathbb{C}$ *an* annulus *as in* (4.2.1) *and* $f : C \to A$ *a diffeomorphism.*

Since the topological boundary ∂A *of* A *is the disjoint union of the two circles* $c_1 = \{|z| = 1\} \subset \mathbb{C}$ *and* $c_2 = \{|z| = 2\} \subset \mathbb{C}$, *the* topological boundary ∂C *of* C *is composed of* exactly *two disjoint closed curves* $b_1 = f^{-1}(c_1)$ *and* $b_2 = f^{-1}(c_2)$.

The curves b_1 *and* b_2 *will now be called the* borders *of the* collar C.

Remark 5. *One can easily show with the above-mentioned collar lemma that if* $Y_1, Y_2 \subset S$ *are two* Y-pieces *sharing as common* boundary *the closed geodesic* γ, *then one of the* borders *of the* collar C *around* γ *lies in* Y_1 *and the other one lies in* Y_2.

The *collars* and their *borders* on a *surface* will play an important role in the construction of the algorithms we will provide. That is why we define a new *class* **Collar** and a new *class* **Border** to symbolically (and numerically) represent these mathematical objects.

The relevant *properties* and *methods* with respect to the geometry that these *classes* have to represent will be explained in Sections 4.4.2 and 4.4.3.

Y-Piece As we have seen that any *closed surface* S of *genus* g can be divided into $2g-2$ non-overlapping open subset, called Y*-pieces*, separated by $3g-3$ non intersecting simple closed geodesics.

The Y-*pieces* will be represented in the following by the class **YPce**, which is detailed in chapter 4.4.4.

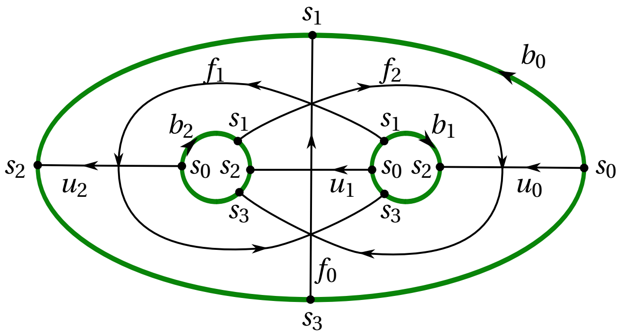


Figure 4.3: The **Road**s and the **Slot**s of a **YPce**.

Roads Let $Y \subset S$ be a Y-*piece* with $\gamma_1, \gamma_2, \gamma_3$ the 3 *simple closed geodesics* composing it's boundary.

The roads are the 6 different simple geodesic arcs that intersect orthogonally two borders of an **YPce**.

Let b and b' be two **Border**s of an Y-*piece* y (the case $b = b'$ is also possible).

Let $f, g : [0,1] \to y$ be two continuous curves on y, such that $f(0), g(0) \in b$ and $f(1), g(1) \in b'$.

If there exists a continuous mapping $h : [0,1] \times [0,1]$ such that $\forall s \in [0,1]$:

$$
\begin{aligned}
&h(s,0) = f(s)\\
&h(s,1) = g(s)\\
&h(0,s) \in b\\
&h(1,s) \in b',
\end{aligned}
$$

then f and g are said to be *locally homotopic curves across* y.

We can also extend this definition to an *orientation-free* equivalence relation if, in addition, we consider the curves $x \mapsto f(x)$, $x \mapsto f(1-x)$, $x \in [0,1]$ to be equivalent.

We can now state the following lemma which plays a fundamental role in the sequel.

Lemma 3. *Let e_i be a geodesic arc of a simple closed geodesic γ, as above, crossing a Y-*piece y_i *and cutting the border(s) b_i and b'_i of y_i.*

Then, in the set of the curves starting on b_i and ending on b'_i, there is exactly one road *of the* **YPce** *that is* locally homotopic *(without orientation) to e_i.*

Proof. You may find a proof of this lemma in the book [4, Chapter 3] ☐

Slot As we have seen, each *road* possess exactly two endpoints. Each of these endpoints lies on a *main geodesic* of the surface.

These two points will for now on be called *slot*. In Section 5.1, on page 69 we will clarify the notion of *frame*. A *frame* will turn out to be the data given by a point and a "direction".

The *slots* of a surface, will then be interpreted not only as points, but as "natural" *frames* (the incoming *road* will define the *axis* of the *frame* associated to the *slot*).

The class that will be used to model the *slots* of a surface will be called **Slot** and is detailed in the `SurfaceTopology` *module*, on page 64.

4.3 Winding index

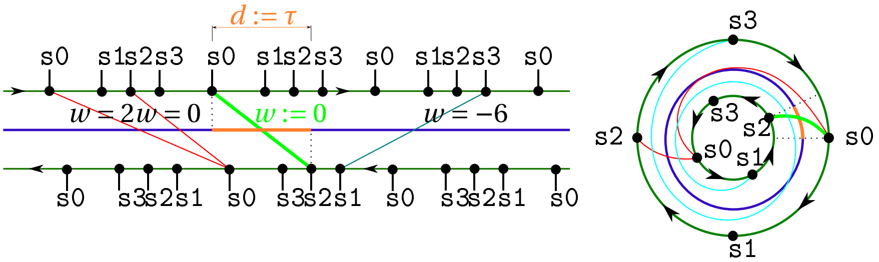

$d := \tau$
s0 s1s2s3 s0 s1s2s3 s0 s1s2s3 s0
$w = 2$
$w = 0$
$w := 0$
$w = -6$
s0 s3s2s1 s0 s3s2s1 s0 s3s2s1 s0
s3
s2
s0
s1
s3
s2
s0
s1

Figure 4.4: The lift of a collar with lifts of **Slot**s facing each other.
For ease of comprehension the *slots* have been drawn on their associated *borders*. Though they are in fact on the *main geodesic*.
Here τ is the *Fenchel-Nielsen twist* oriented distance. w is the *winding index* related to some examples "connected" slots.

Let c be a **Collar** around a *main geodesic* α and b, b' be the **Border**s of c. Let $s0, s1, s2, s3 \in \alpha$ designate the slots whose roads lead towards b and $s0', s1', s2', s3' \in \alpha$ the slots whose roads lead towards b'.

Let $\tilde{\alpha} \subset \mathbb{H}$ be a *lift* of α in the universal covering $\mathbb{H}$, let $S \subset \tilde{\alpha}$ be the set of all the lifts of $s0, s1, s2$ and $s3$ lying on $\tilde{\alpha}$ and let $S' \subset \tilde{\alpha}$ be the set of all the lifts of $s0', s1', s2'$ and $s3'$ lying on $\tilde{\alpha}$.

We define a function $\tau : S \times S' \to \mathbb{Z}$, called the *winding index* that somehow encapsulates the "distance between two lifts of slots".

τ is defined recurcively through the following properties, $\forall \tilde{s} \in S, \tilde{s}' \in S'$:

- If $\tilde{s}$ is a lift of $s0$, and if t is the *twist distance* Fenchel-Nielsen parameter associated to this **Collar**, then by the definition of t, there is exactly one $\tilde{s}' \in S'$ such that the oriented distance from $\tilde{s}$ to $\tilde{s}'$ with respect to the orientation of b is t. Furthermore, according to this definition, $\tilde{s}'$ will be a lift of $s2'$.

  In that circumstance, as starting point, we define that $\tau(\tilde{s}, \tilde{s}') = 0$.

- If we take `prevSlot`, `nextSlot` : $S' \to S'$ to be respectively the previous slot and the next slot along the current border we set

$$\tau(\tilde{s}, \tilde{s}') + 1 = \tau(\tilde{s}, \tilde{s}'.\texttt{nextSlot}())$$

$$\tau(\tilde{s}, \tilde{s}') - 1 = \tau(\tilde{s}, \tilde{s}'.\texttt{prevSlot}())$$

- Finally, we want τ to be symmetric in the sense that if we exchange the roles of the two **Border**s we obtain the same result. Thus, we complete the definition by requiring that

$$\tau(\tilde{s}, \tilde{s}') = \tau(\tilde{s}', \tilde{s})$$

These three conditions obviously allows us to deduce the value of τ for any combination of two lifts of **Slot** facing each other on the same **Collar**.

A resulting property of τ is that if one of its argument is fixed, the restriction of τ to the set of the possible values of the second one is a bijection between S' and $\mathbb{Z}$.

Since in our approach there is no need to implement a class representing a *lift* of a **Slot** there is no *method* to compute this *winding index* in the implementation. However, there are two *methods* of the **Slot** class that are related to the winding index. Assuming that `s1` and `s2` are two **Slot**s lying on the two *facing borders* of the same **Collar**, and assuming that $\tilde{s}1$ and $\tilde{s}2$ are lifts of these slots such that $\tau(\tilde{s}1, \tilde{s}2)) = n$ and that the oriented distance from $\tilde{s}1$ to $\tilde{s}2$ is D, these two methods are the following

`s1.collarFacingSlot(n)`: who returns `s2`.

`s1.collarFacingDist(n)`: who returns D.

See also subsection 4.4.6 on page 64.

4.4 The `SurfaceTopology` module

According to the definitions of the previous sections we define a *module* [5] named `SurfaceTopology`.

This package contains the six following *classes*.

Surface: The class representing a marked *surface*.

YPce: The class representing a Y*-piece* $\in \mathcal{T}_{0,3}$, bounded by 3 of the *main* [6] simple closed geodesics of the surface.

Collar: The class representing an ϵ*-collar* around a *main* geodesic of the surface, with $\epsilon > 0$ an arbitrary small real number. (The value of ϵ does not enter any computation.)

Border: The class representing one of the two *boundaries* of a **Collar**.

Road: The class representing an orthogonal geodesic arc between two boundaries of a Y*-piece* .

Slot: The class representing the intersection point of a **Road** and a **Border**.

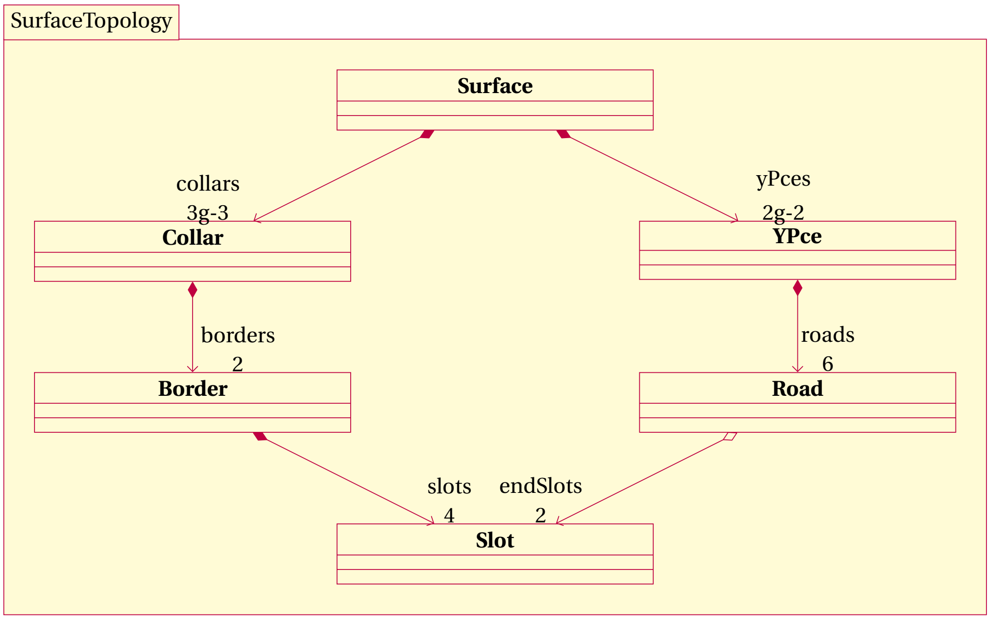


Figure 4.5: Aggregation diagram of the classes of module `SurfaceTopology`.

Figure 4.5 shows the different classes defined in the `SurfaceTopology` module and the *aggregation / composition* relations between them.

---

[5] In `Python` , a *module* is an autonomous file possessing the extension `.py` containing classes and variables that can be used in other *modules* using the `import` statement. Here, the file is then called `SurfaceTopology.py`.

[6] We defined the *main* closed geodesics of a *marked surface* to be the ones that have been chosen for the pants decomposition.

An arrow starting from a class **A** with a plain diamond pointing to a class **B** has to be read as *composition* and means that the class **A** contains a list of objects of type **B**. As an example, the class **Border** contains a list of four **Slot**s called `slots`.

On the other hand the arrows starting with an empty diamond (between **Road** and **Slot**) represent an *aggregation* relation. The **Road** class contains a list of two *references* (or pointers depending on the language and implementation) on **Slot**s called `endSlots`.

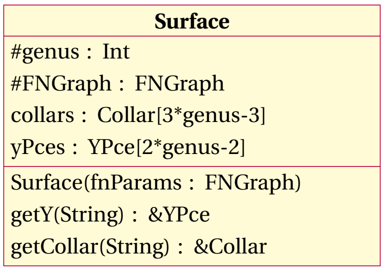
The difference between *composition* and *aggregation* lies in the *ownership* of the objects:
A *composition* relation implies that the contained objects *strongly* belong to the owner, and are destroyed whenever the owner is destroyed.
An *aggregation* relation implies that the container contains just a list of references to pre-existing objects that may survive the destruction of the container.

Note that this difference has an impact on the coding in most of the languages, like `C++`, where the developer has to declare the exact *type* of the variables and has to explicitly implement the *destructors* of the classes.

But in `Python`, the language we choose to illustrate our algorithms, *every* variable is a *reference* and no destructor is mandatory, since the *Python interpreter* can automatically decide to destroy the objects that are not referenced anymore, by a process called *garbage collecting*.

This peculiarity of `Python` is both it's strength, since less code is needed and since *memory leaks* errors are impossible (or at least uncommon), and it's weakness since the *garbage collecting* is a very inefficient process.

4.4.1 The `Surface` class



Surface

#genus : Int
#FNGraph : FNGraph
collars : Collar[3*genus-3]
yPces : YPce[2*genus-2]

Surface(fnParams : FNGraph)
getY(String) : &YPce
getCollar(String) : &Collar

Figure 4.6: **Surface** class diagram.

We define a *class*, **Surface**, that is supposed to contain the data (in finite amount) necessary to represent the geometrical and topological properties of a given surface.

The two main roles of this class are

- To be the container of the **YPce**s and the **Collar**s composing it.

- To act as the *factory* of the five other classes of the `SurfaceTopology` module. I.e. the *constructor* of the **Surface** class is *public* [7] and takes a valid `FNGraph` (see Section 4.1) as argument, whereas the **Collar**s, **YPce**s, **Border**s, **Road**s and **Slot**s have *private* constructors and are only supposed to be *constructed* and filled with adequate data by the **Surface** *constructor*.

Attributes and properties As we have seen in Section 4.2, a **Surface** of *genus* g is composed of a list of $3g-3$ **Collar**s and a list of $2g-2$ **YPce** s. That is why the **Surface** class will have the following attributes.

`property`
Int `genus`**:** A *property* [8] representing the *genus* of the surface. Its value is deduced on the fly through the length of the `yPces` list.

`property`
FNGraph `fnGraph`**:** A *property* giving back the **FNGraph** that has been used to construct the surface. It can be deduced through the content of the `collars` list and can be useful to obtain a *copy* of a **Surface**.

Collar[3g-3] `collars`**:** A list of the $3g-3$ **Collar**s of the **Surface**.

YPce[2g-2] `yPces`**:** A list of the $2g-2$ **YPce**s of the **Surface**.

Methods

Surface (FNGraph): The *constructor* of the class.

As discussed in the previous Section 4.1, a *Fenchel-Nielsen graph* is one of the inputs needed to *construct* **Surface** objects.

This method is also in charge of the initialization of the **Collar**, **YPce**, **Border**, **Road** and **Slot** classes and their content.

&YPce `getY(` **String):** We have seen that, in a **FNGraph** each Y-*piece* is identified by a **String**. This method allows one to recover the **YPce** object that has the given **String** as its `name`.

An exception is raised if the passed argument does not correspond to the *name* of one of the Y-*pieces*.

---

[7] A *public* method is supposed to be used from outside, in contrast to *private* and *protected* methods that are supposed to be used only internally.

[8] A *property* is a special kind of attribute, it can be called like a normal variable but it is in fact a *method* taking no argument. The advantage of using this concept is to provide the user of the class with an interface that is independent of the underlying implementation. Furthermore, since the properties are just functions, they have no impact on the *size* of the object they belong to. In the context of this document, we will signal the *properties* with a # in the *class diagrams* and consider them as *read-only*.

&Collar `getCollar(` **String):** We have seen that, in an **FNGraph** each *edge* (represented by a **Collar**) can optionally be identified by a **String**. This method allows one to recover the **Collar** object corresponding to the passed argument.

An exception is raised if the passed argument does not correspond to the *name* of one of the **Collar**s.

4.4.2 The `Collar` class

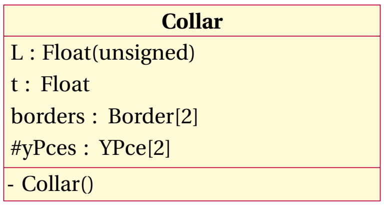


Figure 4.7: **Collar** class diagram.

The class **Collar** will at the same time represent an *edge* of the *Fenchel-Nielsen graph* of a surface.

Geometrically, this class also represents an ϵ-*collar* around a *main* geodesic of the surface. The ϵ in question is supposed to be an arbitrary small number, it aims just to give a sense to the fact that the **Collar** is topologically bounded by two **Border**s. One has to think of ϵ as being chosen so small that the $3g-3$ **Collar**s are pairwise disjoint.

Collar contains following attributes

Float `L`**:** A positive number representing the *length* of the *main geodesic* of this **Collar**.

Float `t`**:** The twist associated to the *gluing* of the adjacent **YPce**s (see Section 4.1).

Border[2] `borders`**:** The two **Border**s of this **Collar**.

`property`
&YPce[2] `yPces`**:** A *property* returning the two **YPce**s of this **Collar**. It can be deduced by the content of `borders`.

Finally, the *constructor* of the **Collar** class is marked as *private* (the - sign in the class diagram) which means that they have not to be created from outside. The only *method* authorized to construct new **Collar**s is the *constructor* of the **Surface** class.

4.4.3 The `Border` class

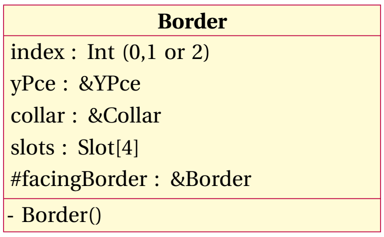


Figure 4.8: **Border** class diagram.

The class **Border** is designed to model a *border*, one of the two *boundaries* of a *collar* around a main geodesic γ.

In Section 4.2, we defined the *roads* of an Y*-piece*. Each *border* intersects precisely four *roads*. These intersections are points are called *slots* and are represented by the class **Slot**.

The **Border** class has the following attributes.

Int `index`: As will be described below, each **YPce** contains a list of 3 **Border**s, this integer allows the **Border** to know it's position in it's adjacent **YPce**'s **Border** list. This attribute can take the values 0,1 or 2.

&YPce `yPce`: The **YPce** adjacent to this **Border**.

&Collar `collar`: The **Collar** adjacent to this **Border**.

Slot[4] `slots`: A list of the 4 **Slot**s lying on this **Border**. According to the notations of Section 4.2, this list is of the form $[s0, s1, s2, s3]$, where $s0$ and $s2$ are endpoints of the two *u roads* connected to γ, while $s1$ and $s3$ are the two endpoints of the f road connecting γ to itself.

`property`
& Border `facingBorder`: The **Border** that lies at the other side of the adjacent **Collar**.

Finally, the *constructor* of the **Border** class is marked as *private* (the - sign in the class diagram) which means that they cannot be created from outside . The only *method* authorized to construct new **Border**s is the *constructor* of the **Surface** class.

4.4.4 The `YPce` class

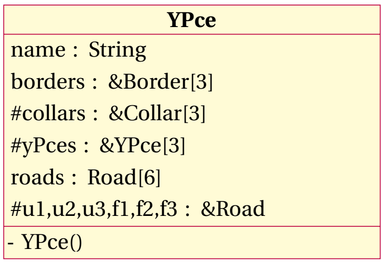


Figure 4.9: **YPce** class diagram.

A (marked) surface of *genus* g is divided into $2g-2$ Y-*pieces* as described in Section 4.2.

Each **YPce** is identified by a **String**. It is bounded by 3 **Collar**s and contains 6 distinguished geodesic arcs represented by the **Road** class.

These **Road**s are the perpendicular geodesic arcs u_1, u_2, u_3, f_1, f_2 and f_3 of the Y-*piece* as drawn in Figure 4.3 and are represented by a list $[u_1, u_2, u_3, f_1, f_2, f_3]$, given in this order.

This class then possesses the following attributes.

String `name`**:** Each **YPce** is identified by an unique **String**, inherited from the **FNGraph**. The `name` attribute stores this identifier.

&Border[3] `borders`**:** The **YPce** is adjacent to 3 **Collar**s, (it may occur that two among these **Collar**s coincide). Each boundary of the **YPce** is one of the two **Border**s of the adjacent **Collar**. This list contains a reference to these borders.

`property`
&Collar[3] `collars`**:** The 3 **Collar**s adjacent to this **YPce**, deduced from the `borders` list.

`property`
&YPce[3] `yPces`**:** The 3 **YPce**s adjacent to this **YPce**, deduced from the `borders` list.

Road[6] `roads`**:** The 6 **Road**s of the **YPce**, as shown in Figure 4.2 , given in the order $[u_0, u_1, u_2, f_0, f_1, f_2]$.

`property`
&Road `u0,u1,u2,f0,f1,f2`**:** These properties allow one to reach the roads by their *name*.

```
#Example
ypce.f1 is ypce.roads[4]
>>True
```

Finally, the *constructor* of the **YPce** class is marked as *private* (the - sign in the class diagram) which means that they cannot be created from outside. The only *method* authorized to construct new **YPce**s is the *constructor* of the **Surface** class.

4.4.5 The `Road` class

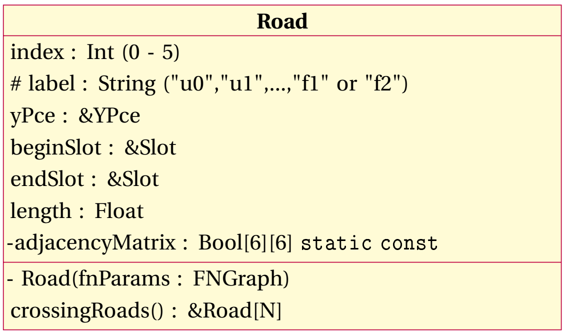

Road
index : Int (0 - 5)
# label : String ("u0","u1",...,"f1" or "f2")
yPce : &YPce
beginSlot : &Slot
endSlot : &Slot
length : Float
-adjacencyMatrix : Bool[6][6] static const
- Road(fnParams : FNGraph)
crossingRoads() : &Road[N]

Figure 4.10: **Road** class diagram.

The **Road** class will be our model to represent the 6 oriented geodesic arcs of a particular **YPce**, as described in Section 4.2, on page 52.

Both of the endpoints of these arcs lie on some of the *main closed geodesics* bounding the **YPce**. We identify these endpoints with two **Slot**s that we call `beginSlot` and `endSlot` according to the orientation we have chosen for the roads. To be precise, the slots lie on the **Border** of a **Collar** and not on the closed geodesic; so the endpoints are at distance ϵ away from the slots. We shall suppress this fact by assuming that ϵ is “infinitesimally small”.

Road attributes

Int `index`: This integer stores the position of this **Road** in the **YPce**.`roads` list.

`property`
String `label`: The label `"u0"`, `"u1"`, `"u2"`, `"f0"`, `"f1"` or `"f2"` as a **String**, depending on the value of `index`.

&YPce `yPce`: The **YPce** on which the **Road** lies.

&Slot `beginSlot`: The **Slot** from which the road starts.

&Slot `endSlot`: The **Slot** on which the road ends.

Float `length`: The geometrical distance between the two endpoints of the road. This parameter only depends on the length of the **Border**s of the **YPce** an can be calculated at the construction of the **YPce**, using the formulas given in Section 3.3.

`static const`
Bool[6][6] `adjacencyMatrix`**:** This attribute corresponds to the following 6×6 boolean matrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(a possible example). It is marked as `static, const` and `private` and has therefore to be considered as a *global* constant whose scope is restricted to the methods of the **Road** class.

This matrix contains a 1 (true) at the position given by the indexes i, j if and only if the road with `index` i intersects the road with `index` j.

This matrix is used by the method `crossingRoads()`.

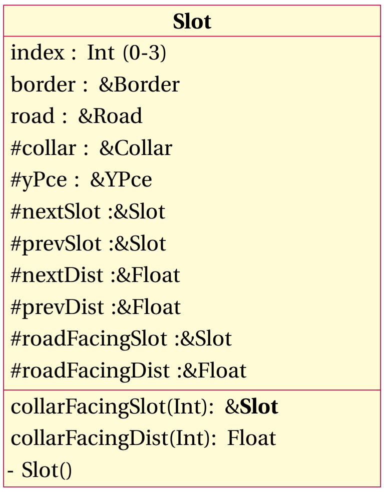
Road methods

&Road[N] `crossingRoads`**():** Returns a list of the **Road**s crossing this one (using the `adjacencyMatrix` content).

The value of N is 1 for the u roads and 3 for the f roads.

Road ( ): The constructor is marked *private* and is only supposed to be called by the **Surface** constructor.

4.4.6 The `Slot` class



Slot

index : Int (0-3)
border : &Border
road : &Road
#collar : &Collar
#yPce : &YPce
#nextSlot :&Slot
#prevSlot :&Slot
#nextDist :&Float
#prevDist :&Float
#roadFacingSlot :&Slot
#roadFacingDist :&Float

collarFacingSlot(Int): &**Slot**
collarFacingDist(Int): Float
- Slot()

Figure 4.11: **Slot** class diagram.

Finally, **Slot** is the last class of the `SurfaceTopology` module.

It represents the intersection between a **Border** and a **Road**. This class plays a particular role in the context of the search of (simple) closed geodesics of the surface. They will be used as the *vertices* of certain *geodesic polygons* to be constructed on the surface. These *polygons* will themselves turn out to be in 1 to 1 correspondence with the simple closed geodesics of the surface and will be represented by the class **Path**, see Chapter 6.

Properties and methods

Int `index`**:** Represents the position of this **Slot** as an element of the **Border**'s **Slot** list. It can take the value 0 for $s0$ to 3 for $s3$.

& Border: `border` The border on which the **Slot** lies.

& Road: `road` The road of which the **Slot** is an endpoint.

`property`
& Collar: `collar` The collar adjacent to the **Slot**.

`property`
& YPce: `yPce` The **YPce** adjacent to the **Slot**.

`property`
& Slot: `nextSlot` The *next* **Slot** on the same **Border** going in the positive direction according to the **Border**'s orientation.

`property`

& Slot: `prevSlot` The *previous* **Slot** on the same **Border** going in the opposite direction to the **Border**'s orientation.

`property`

& Float: `nextDist` The positive hyperbolic distance between `self` and `self.nextSlot`.

`property`

& Float: `prevDist` The positive hyperbolic distance between `self` and `self.prevSlot`.

`property`

& Slot: `roadFacingSlot` The **Slot** at the other endpoint of the **Road**.

`property`

& Float: `roadFacingDist` The distance between `self` and the `roadFacingSlot`.

& Slot: `roadFacingSlot` **(Int)** The slot we reach by crossing the collar with a given *windingIndex*.

& Float: `roadFacingDist` **(Int)** Returns the distance between `self` and the `collarFacingSlot(twistParam)`, for a given *twistParam*.

5 Fundamental domains

Let $S = \mathbb{H}/\Gamma$ be an *hyperbolic Riemann surface.* We provide the definition of a *fundamental domain* of S before studying how we can computationally provide representations of such sets. Then, we also present some algorithms that we can apply to these models of *domains* to obtain, for example *canonical domains.*

Definition 19 (Fundamental Domain)**.** *Let* $S = \mathbb{H}/\Gamma$ *be a* Riemann surface. *A subset* $F \subset \mathbb{H}$ *is called a* Fundamental Domain *if and only if it respects the following conditions:*

- $\bigcup_{g\in\Gamma} g(F) = \mathbb{H}$
- $\forall x, y \in int(F), \qquad [x]_\Gamma = [y]_\Gamma \Leftrightarrow x = y.$
- ∂F *the boundary of* F *has Lebesgue measure* 0.

Here $[x]_\Gamma = \{\gamma(x) \mid \gamma \in \Gamma\}$ is the *orbit* of a point $x \in \mathbb{H}$ under the action of the group Γ.

Remark 6. *In this document we consider only* domains *with* piecewise geodesic boundaries*, the* Lebesgue measure zero *condition will then always be satisfied.*

Note that there are infinitely many valid *fundamental domains* for a given surface. In this chapter, we will present how to model two different kinds of *fundamental domains* of a given surface (represented by a **Surface** object).

The first kind of fundamental domain will be called *basic domain* and will be represented computationally by the class **BasicDomain**, and mainly represents a *patchwork* of Y*-pieces* domains.

The second type of domains will be called *canonical domain* and represented by the class **CanonicalDomain**. The *canonical domains* are *convex* $4g$*-polygons.*

Both of these classes are subclasses of the **Domain** class, which will be the base of every hyperbolic domain we deal with.

The details about these classes can be found in the `HyperbolicDomain` module on page 79.

5.1 Frames in $\mathbb{H}$

In order to describe the geometry of the fundamental domains and the algorithms involved in their *construction*, we provide some handy description of a *frame* in $\mathbb{H}$.

Definition 20. *Let $a \in \mathbb{H}$ be a point and let $\lambda_a \subset \mathbb{H}$ be an* half-geodesic *with endpoint*[1] *a.*

We call the pair (a, λ_a), also denoted $\vec{a}$, the oriented point a in direction of λ_a.

In Definition 24, it will be pointed out how the position of any oriented point $\vec{b}$ can be described relatively to the position of an oriented point $\vec{a}$. This is why the oriented points will be referred to as frames.

The point a is referred as the origin *of the frame $\vec{a}$ and λ_a as the* vertical axis *of the frame $\vec{a}$.*

On the purely geometrical point of view, every oriented point of $\mathbb{H}$ plays exactly the same role, but it is still necessary to pick one up to play the role of the *absolute* or *canonical frame* the following choice looks the least arbitrary.

Definition 21. *The frame $\vec{0} := (i, V_i^+)$ with $i \in \mathbb{H}$ associated to the purely imaginary half-geodesic $V_i^+ = \{\lambda i \in \mathbb{H} | \quad \lambda \in \mathbb{R}, \lambda > 1\}$, will be referred as the* absolute *or* canonical frame.

Now we can introduce the definition of some Möbius transformations that will encapsulate the relations between the positions of these frames relatively to each other.

Proposition 6. *Let $\vec{a} = (a, \lambda_a)$ be a* frame.

Then, there exists a unique[2] Möbius transformation *$\phi^a \in Möb(\mathbb{H})$ such that $\phi^a(i) = a$ and $\phi^a(V_i^+) = \lambda_a$.*

The transformation ϕ^a will be referred as the absolute position *of the frame $\vec{a}$.*

Proof. Given without proof, since well known. $\square$

Remark 7. *We can note that $\phi^0 = Id_{\mathbb{H}}$.*

Definition 22. *Let $\vec{a}, \vec{b}$ be* frames *of* $\mathbb{H}$.

The transformation $\phi^{ab} := \phi^b(\phi^a)^{-1}$ will be called the absolute transformation from a to b.

Remark 8. *One can see that $\phi^{ab}(a) = b$ and $\phi^{ab}(\lambda_a) = \lambda_b$.*

Remark 9. *One can also note that $\phi^{0b} = \phi^b$.*

---

[1] boundary

[2] Here, note that Möb($\mathbb{H}$) stands for the *direct* transformations only.

Definition 23. *Let $\vec{a}, \vec{b}, \vec{c}$ be* frames *of* $\mathbb{H}$.

The transformation $\phi_a^{bc} := (\phi^a)^{-1}\phi^{bc}\phi^a = (\phi^a)^{-1}\phi^c(\phi^b)^{-1}\phi^a$ *will be called the* relative transformation from $\vec{b}$ to $\vec{c}$ with respect to the frame $\vec{a}$.

Remark 10. *We can note that* $\phi_0^{ab} = \phi^{ab}$.

Definition 24. *Finally, $\vec{a}$ and $\vec{b}$ being* frames *of* $\mathbb{H}$, *we define the transformation* $\phi_a^b := \phi_a^{ab} = (\phi^a)^{-1}\phi^b$ *to be the* relative position of the frame $\vec{b}$ with respect to the frame $\vec{a}$.

Remark 11. *Note that* $\phi_0^a = \phi^a$.

We can now justify these definitions (and the chosen notations) by the following propositions about the compositions of the transformations.

Proposition 7. *Let $\vec{a}, \vec{b}, \vec{c}$ and $\vec{d}$ be any* frames *in* $\mathbb{H}$, *then the following equations hold.*

$$\phi_a^{cd}\phi_a^{bc} = \phi_a^{bd}$$

$$\phi_a^b\phi_b^c = \phi_a^c.$$

Proof. By a direct computation using Definitions 23 and 24. ☐

Remark 12. *The preceding proposition can also be applied recursively to obtain a composition of any number of transformations.*

5.1.1 The `HFrame` class

Computationally speaking, the 2 by 2 *real matrices* that represent the ***Möbius Transformations** of* $\mathbb{H}$ will from now on be represented by elements of the class **TMH**[3].

In this section, we describe how to conceive a class **HFrame** that represents a *frame in* $\mathbb{H}$ as described above.

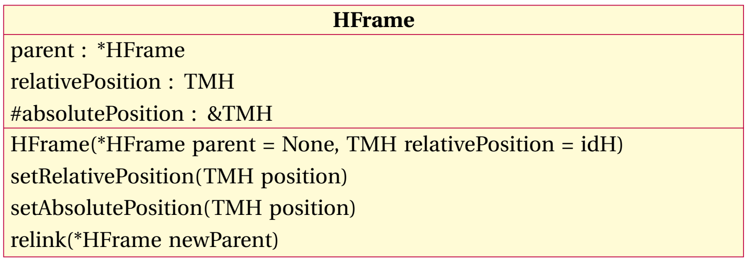

HFrame
parent : *HFrame
relativePosition : TMH
#absolutePosition : &TMH
HFrame(*HFrame parent = None, TMH relativePosition = idH)
setRelativePosition(TMH position)
setAbsolutePosition(TMH position)
relink(*HFrame newParent)

Figure 5.1: **HFrame** class diagram.

---

[3] We do not give more details about the content of this **TMH** class, but one can assume that the *objects* of this *class* can be added, multiplied, inverted, etc...

Each **HFrame** `a` is the representation of a frame $(a, \lambda a)$. The constructor of the **HFrame** class takes two informations:

- A *pointer* [4], called `parent` that either points to an other **HFrame** or points to `None`.
- A *Möbius transformation of* $\mathbb{H}$, `relativePosition`, represented by the class **TMH**.

  This `relativePosition` variable contains the *relative position* of the current frame with respect to the `parent` frame, like in Definition 24.

  If the variable `parent` is `None`, it is assumed that the parent is the *absolute frame* $\vec{0}$.

The class **HFrame** also contains a *property* called `absolutePosition` that returns the *absolute position* of the current frame, in the form of a **TMH**, as defined in Proposition 6.

If the `parent` is `None`, according to the Remark 11, the *absolute* and the relative position coincide and the property `absolutePosition` simply returns a copy of `relativePosition`.

If the `parent` is pointing to a valid **HFrame**, according to Proposition 7,
the `absolutePosition` will return
`self.parent.absolutePosition*self.relativePosition`.

Finally, the class possesses three methods:

`setRelativePosition`**(TMH newRelPos)** Replace the current `relativePosition` by the value of `newRelPos`. It also affects the value of `absolutePosition`.

`setAbsolutePosition`**(TMH newAbsPos)** Modify the value of the current `relativePosition` in order to make the new value of `absolutePosition` coincide with the value of `newAbsPos`.

`relink`**(HFrame * newParent = None)** Change the current `parent` and modify the value of the current `relativePosition` in order to let value of `absolutePosition` be unchanged.

Note that both classes **TMH** and **HFrame** are parts of the `HyperbolicGeometry` module, which gather all the basic classes to represent the *points* and the *transformations* in different models ($\mathbb{H}$, $\mathbb{D}$, $\mathbb{MH}$, $\mathbb{MD}$, etc.) as well as tools for translating objects from a model to an other.

---

[4] In our philosophy, the pointers are *nullable*, whereas the *references* **must** refer to a valid object.

5.2 Hyperbolic polygons

Let us now describe more precisely how piecewise geodesic hyperbolic polygons can play the role of the *fundamental domain* of a surface, and conversely, why we can guarantee the existence of such domains for any hyperbolic (closed) surface.

The answer to this question which we will provide in this chapter is modeled on the proof of the classical *surface classification theorem*, that is valid for any connected closed surface. The proof of this theorem may be found in many textbooks on topology, e.g. [9, Chapter 1, §8]. It serves as a guideline for the case of our Riemann surfaces.

Let us now begin with some definitions.

Definition 25. *A compact connected subset $F \subset \mathbb{H}$ is an* hyperbolic polygon *with n* edges*, or simply an n-gon if and only if it's* boundary *∂F respects the following conditions.*

- *There exists n geodesic arcs denoted by $e_1 := \overrightarrow{v_1 v_2}, e_2 := \overrightarrow{v_2 v_3}, \ldots, e_{n-1} := \overrightarrow{v_{n-1} v_n}, e_n := \overrightarrow{v_n v_1}$,*
  where $\overrightarrow{v_i v_j}$ stands for the oriented geodesic arc *from v_i to v_j,*
  such that $\partial F = \bigcup_{i=1}^n e_i$;
- *$e_1 \cap e_2 = \{v_2\}$, $e_2 \cap e_3 = \{v_3\}, \ldots, e_{n-1} \cap e_n = \{v_n\}$, $e_1 \cap e_n = \{v_1\}$;*
- *$e_i \cap e_j = \emptyset$, for all other $i < j$.*

By convention, we will assume that the interior of F has a positive orientation *i.e. that we turn counter-clockwise around the polygon when we follow the successive oriented edges, like in figure 5.2.*

Note that ∂F is connected and of Lebesgue measure 0. The n points $v_1, \ldots, v_n \in \partial F$, are called the *vertices* of the polygon.

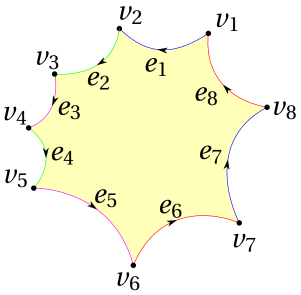


Figure 5.2: An 8-gon up to homeomorphism.

Rather than thinking about polygons in terms of sets, we can also think in terms of *marked hyperbolic polygons.*

Definition 26. *A* marked hyperbolic n-polygon *P is the data given by an ordered list $P = [v_1, v_2, \ldots, v_n]$, where $v_i \in \mathbb{H}, \forall i \in \{1, \ldots, n\}$, of points that we will call* vertices.

Each of these vertices is supposed to be joined to the next one by an oriented geodesic arc $e_i := \overrightarrow{v_i v_{i+1}}$, called edge*; whereas the last vertex v_n is connect to the first one by the edge $e_n := \overrightarrow{v_n v_1}$ so that we get a closed curve. All arcs are supposed to have positive lengths.*

This definition is less strict than the Definition 25 on an *hyperbolic polygon*, in the sense that it allows features like *self-intersections* or *cusps*, like in Figures 5.5 and 5.3. Furthermore, it is not always the full boundary of a domain.

However, this degree of freedom will be helpful in the design of the algorithms.

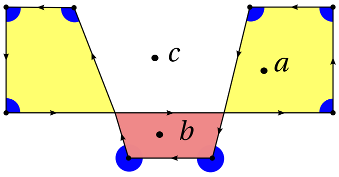


Figure 5.3: The *marked polygon* is degenerated.
The *winding angle* is such that $\alpha_P(a) = 2\pi, \alpha_P(b) = 0, \alpha_P(c) = -2\pi$

The following remark connects the two definitions.

Remark 13. *The classical* Jordan curve theorem *(see e.g. [13, Section 0.3] tells us that a marked polygon in the sense of Definition 26 that satisfies the items as in Definition 25 is always the boundary of a compact domain $F \subset \mathbb{H}$ that is homeomorphic to a closed disk.*

If a given *marked hyperbolic polygon* is the boundary of a domain F as in the above remark, then it is said to be *non-degenerated.* In the other cases it will be called degenerated.

Since a *marked polygon* is an oriented curve, it is also still possible to define a notion of "interior angle" at each *vertex* even though the "interior" is sometimes not well defined. Assume that the two edges at vertex v have only the point v in common. Then the *interior angle at v* is defined to be the positive angle in the interval $]0, 2\pi[$ that you have “on your left” if you are following the orientation of the edges. It is also possible to extend this definition to certain limiting cases where the two edges at v have the smaller edge in common. Two such cases are considered in Definition 28.

The *interior angle* of each vertex has been drawn in blue in Figures 5.5 and 5.3.

Next we come to a number of limit situations of marked polygons that may arise in various cases.

Definition 27 (Flat point)**.** *Consider a (marked) n-gon P. It may happen that three* consecutive

[5] *vertices* v_{j-1}, v_j *and* v_{j+1} *are aligned, i.e.* $v_j \in \overrightarrow{v_{j-1}v_{j+1}}$ *for a certain* j [6].

A vertex v_j *with this property is said to be a* flat vertex *of* P.

Note that the value interior angle *at a* flat vertex *is* π.

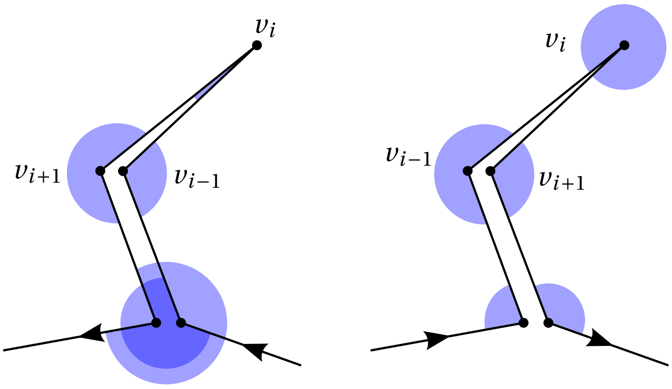


Figure 5.4: The angles in an "exagerated" view of a cusp.

Definition 28 (Peaks and cusps)**.** *Let us take a marked* n*-gon* P *and a sequence of vertices of* P *that, up to cyclic permutation, can be written*

$$v_{i-k}, \ldots, v_{i-1}, v_i, v_{i+1}, v_{i+k},$$

where $v_{i-k} = v_{i+k}, \ldots, v_{i-1} = v_{i+1}$*, and where the two interior angles* α *at* v_{i-k} *and* β *at* v_{i+k} *satisfy* $\alpha + \beta \neq 2\pi$*. A vertex* v_i *for which this holds is called*

- *a* cusp *point of* P *if* $\alpha + \beta < 2\pi$,
- *a* peak *point of* P *if* $\alpha + \beta > 2\pi$.

The interior angle *at a* peak point *is defined to be* 0 *whereas it is defined to be* 2π *for* cusps.

Look at the Figure 5.5 for illustration.5

A marked n*-gon with no* flat vertex, *and no* peak/cusp *will be said to be* minimal.

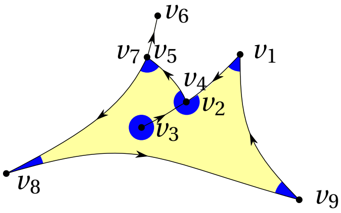


Figure 5.5: A marked 9-gon up to homeomorphism, with v_3 a *cusp* vertex and v_6 a *peak* vertex.

---

[5] In the sens that $\overrightarrow{v_{j-1}, v_j}$ and $\overrightarrow{v_j, v_{j+1}}$ are consecutive *edges*.

[6] The cases v_n, v_0, v_1 and $v_{n-1}v_nv_1$ have to be defined separately.

Later, we will consider polygons that are *fundamental domains* of a surface. It will then be handy to dispose of a definition for "being a fundamental domain" that also includes the *degenerated* hyperbolic polygons since we have not been able to avoid dealing with this kind of objects in our algorithms.

To do that we introduce the concept of *winding angle around a point with respect to a marked polygon*, which is also sometimes referred to as "Umlaufszahl".

Definition 29. *Let* $P = [v_1, \dots, v_n]$ *be a marked polygon in* $\mathbb{H}$ *and* $p \in \mathbb{H}$ *be a point that does not lie on the polygon itself (seen as a curve).*

Then, for each edge $e_i = \overrightarrow{v_i v_{i+1}}$, *we take* α_i *to be the oriented angle from the geodesic arc* $\overrightarrow{pv_i}$ *to the geodesic arc* $\overrightarrow{pv_{i+1}}$ *at point* p. *By "oriented angle" we mean the angle of rotation in the interval* $]-\pi, \pi]$ *of the rotation around* p *that sends* $\overrightarrow{pv_i}$ *towards* $\overrightarrow{pv_{i+1}}$.

Given that, we define

$$\alpha_P(p) := \sum_{i=1}^{n} \alpha_i$$

to be the winding angle *around* p *with respect to* P. *As in Euclidean geometry it is an integer multiple of* 2π.

For example in Figure 5.3, the yellow points have a winding angle of 2π, the red ones have -2π and all the other ones have 0.

Note also that in a non-degenerated polygon, according to our orientation convention, the winding angle is 2π in the interior and 0 everywhere else.

The last definition we need before passing to the fundamental domains themselves is the following.

Definition 30. *Let* $v_1, v_2, w_1, w_2 \in \mathbb{H}$ *be points (or vertices of a polygon), such that* $d(v_1, v_2) = d(w_1, w_2)$. *Then, by Proposition 6, there is a unique Möbius transformation, denoted by*

$$\phi_{\overrightarrow{v_1 v_2}}^{\overrightarrow{w_1 w_2}} \in Möb(\mathbb{H})$$

such that,

$$\phi_{\overrightarrow{v_1 v_2}}^{\overrightarrow{w_1 w_2}}(v_1) = w_2$$

$$\phi_{\overrightarrow{v_1 v_2}}^{\overrightarrow{w_1 w_2}}(v_2) = w_1.$$

This transformation is here referred to as the direct identification *from the* edge $\overrightarrow{v_1 v_2}$ *to the* edge $\overrightarrow{w_1 w_2}$.

These two edges *are then said to be* pairwise identified *through* $\phi_{\overrightarrow{v_1 v_2}}^{\overrightarrow{w_1 w_2}}$.

Note that $\phi_{\overrightarrow{v_1 v_2}}^{\overrightarrow{w_1 w_2}} = \left(\phi_{\overrightarrow{w_1 w_2}}^{\overrightarrow{v_1 v_2}}\right)^{-1}$.

5.3 Domain representation

At the beginning of the Chapter, on page 67, we gave a standard definition of a *fundamental domain* for a given surface $S = \mathbb{H}/\Gamma$.

Let us begin this chapter by the definition of a *pairwise identified polygon*. We will then recall the *Surface classification Theorem* that for any surface $S = \mathbb{H}/\Gamma$, there exist some *pairwise identified polygon* which is at the same time a *fundamental domain* of S.

Definition 31 (pairwise identified polygon)**.** *Let P be an hyperbolic marked polygon with $2n$ vertices. Then P is said to be a* pairwise identified (hyperbolic) polygon *if and only if the $2n$ edges can be split into n pairs of edges of the same hyperbolic length.*

Let us assume that $(\vec{e}_i, \vec{e}_j)$ is one of these pairs, then, we define the Möbius transformations $\gamma_{e_i} = \phi_{\vec{e}_i}^{\vec{e}_j}$, to be the identification *associated to e_i, where $\phi_{\vec{e}_i}^{\vec{e}_j}$ is the* direct identification *from $\vec{e}_i = \overrightarrow{v_i v_{i+1}}$ to $\vec{e}_j = \overrightarrow{v_j v_{j+1}}$, in the sense of Definition 30 (v_i is sent to v_{j+1}, and v_{i+1} to v_j).*

Note that we have $\gamma_{e_j} = (\gamma_{e_i})^{-1}$, we then define the n oriented identifications *to be the ones for which $i < j$.*

On Figure 5.2, on page 72, a case is represented; four different colors have been used to highlight the different pairwise identified edges.

A *pairwise identified polygon* can also be characterized by a symbol called its *signature*, through this concept is classic, we give our definition to make things clear.

Definition 32 (signature)**.** *Let $\{a, b, c, ...\}$ be the ordered sequence of the n* oriented identifications *that appear in a pairwise identified polygon. Then, the* signature *of the identifications is defined as the "word" that can, for example be of the form*

$$abcc^{-1}da^{-1}\ldots$$

obtained by the following process:

1) *Start at any* edge *on the oriented polygon.*

2) *Write the* name *of the* oriented *identification (d for example) associated to this edge.*

3) *Add the* inversion *symbol $^{-1}$ to this name if the current edge happens to be the* end *of the oriented identification.*

4) *Take the next edge on the oriented boundary and redo steps 2 and 3 until every edge has been treated.*

All these definitions allow us to state the following classical theorem.

Theorem 6 (Surface classification Theorem)**.** *Let* $S = \mathbb{H}/\Gamma$ *be a closed Riemann surface of genus* g *with* Γ *it's Fuchsian group. Then, there exist a* fundamental domain $F \subset \mathbb{H}$ *of* S *respecting the following conditions.*

a) F *is a* pairwise identified convex hyperbolic polygon *with* $4g$ vertices $v_1, v_2, \ldots, v_{4g} \in \mathbb{H}$.

b) The $2g$ oriented identifications $\{\gamma_1, \ldots, \gamma_{2g}\}$ *associated to the* $2g$ *pairs of edges* [7] *form a minimal generating set of the Fuchsian group* Γ.

c) Each vertex lies in the same orbit of Γ*, i.e.* $v_i \sim_\Gamma v_j, \quad \forall i, j \in \{1, 2, \ldots, 4g\}$.

d) The signature *of* F *is* canonical *i.e. it is a pattern of the form* $\quad aba^{-1}b^{-1}cdc^{-1}d^{-1}\ldots$

e) F *is* non-degenerated.

Any fundamental domains *of* S *respecting a)- e) will be called a* canonical domain.

This Theorem as it is given here is in fact a corollary of the *official* and more general *Topological compact surface classification theorem*, restricted to the case of hyperbolic surfaces, that we know to be orientable. A proof of it can be found in [4, Chapter 6].

One of the goals of this document (as indicated by it's title) is to actually provide a formal explanation on how such a polygon can be found *computationally.* This is carried out under the hypothesis that the surfaces are described by Fenchel-Nielsen parameters.

The algorithms needed to perform this are explained in Sections 5.5 and 5.6.

Before that, in the next Section, 5.4, we will introduce a description of how to represent the preceding mathematical notions (domain,vertex,etc.) in terms of *classes.*

These classes are grouped in the `HyperbolicDomain` module.

Once the data structure needed has been clearly described, the Section 5.5 explains how to algorithmically build a **Domain** and show that it is a fundamental domain with certain particular properties. These domains will be called *basic domains.*

Finally, in Section 5.6, we describe how, given a *basic domain*, we can transform it step by step into a *canonical domain* in the sense of the Theorem 6.

This will in itself constitute a (formal) proof of Theorem 6.

---

[7] This avoids taking the inverse of these identifications with.

5.4 The `HyperbolicDomain` module

According to the definitions of the previous Section,
we define a *module* called `HyperbolicDomain` that will contain the following *classes* :

- **DomVertex**: That plays the role of the model of a vertex of a *polygonal domain.*

- **DomEdge**: That is a representation of an *edge* of a *polygonal domain.*

- **DomIdentification**: Represents an *(oriented) identification* from a **DomEdge** to an other **DomEdge**.

- **Domain**: That is the base class of the *pairwise identified polygonal domains.*

- **HexDom**,**YDom**,**CollarDom**: Are models of *sub-domains* as defined in Section 4.2 and serve as "tiles" in the construction of a **BasicDomain**. They are subclasses of the **Domain** class.

- **BasicDomain**, **CanonicalDomain**: Are models of a *basic domain* and of a *canonical domain* respectively. They are subclasses of the **Domain** class.

Figure 5.6 shows the inheritance relations as well as the aggregation/composition relations between these classes.

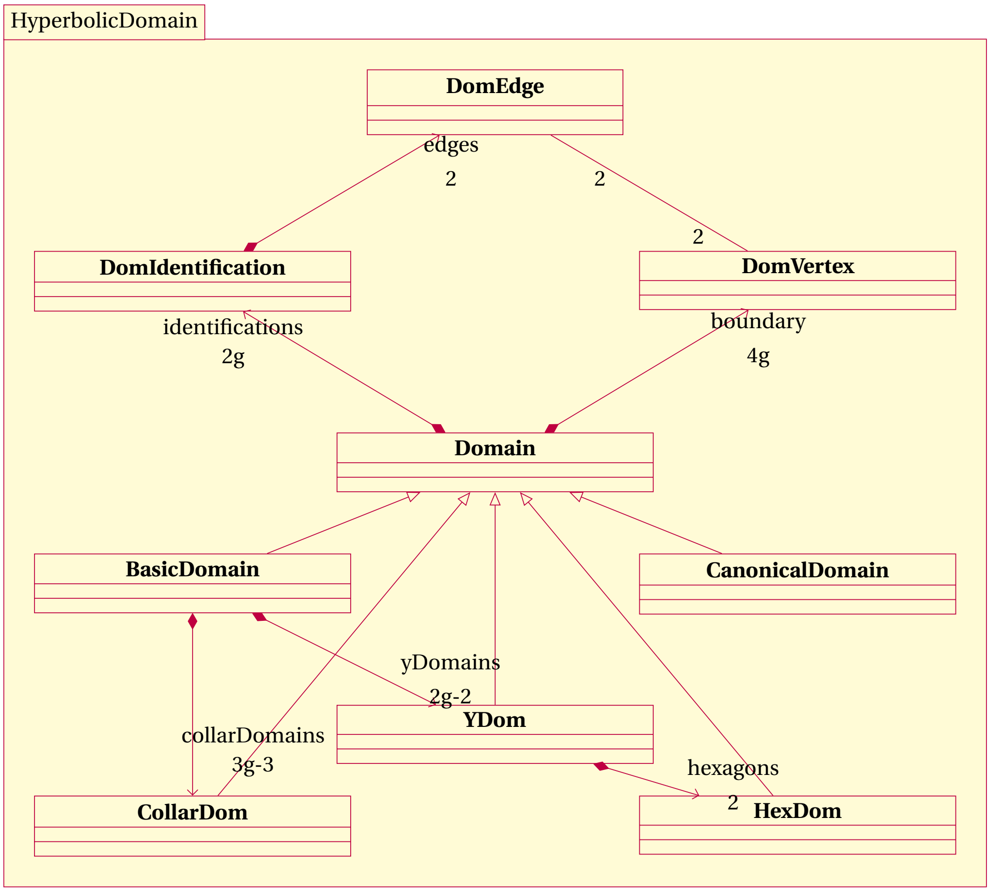


Figure 5.6: Classes of the `HyperbolicDomain` module.

5.4.1 The `DomVertex` class

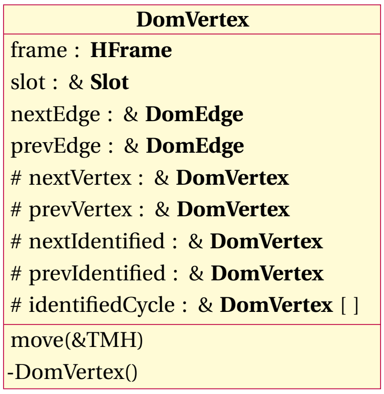

DomVertex
frame : HFrame
slot : & Slot
nextEdge : & DomEdge
prevEdge : & DomEdge
# nextVertex : & DomVertex
# prevVertex : & DomVertex
# nextIdentified : & DomVertex
# prevIdentified : & DomVertex
# identifiedCycle : & DomVertex [ ]
move(&TMH)
-DomVertex()

Figure 5.7: **DomVertex** class diagram.

The **DomVertex** class is meant to be a model of a *vertex* of a *polygonal domain*.

We now describe the actual data contained in a **DomVertex**. These data are in terms of attributes. We also define some other relevant data that can be deduced through the values of the attributes in terms of properties and methods.

First of all we stipulate that the **DomVertex** class shall not be designed to represent arbitrary points of $\mathbb{H}$ but only those points in $\mathbb{H}$ that are in the orbit (under the Fuchsian group Γ) of a **Slot**, i.e. of a vertex of the initially given *fundamental domain*.

This stipulation might at first glance be seen as a severe restriction, but it is actually quite mild and has the following advantages:

- The algorithm for the construction of *canonical domains* is exact. [8]
- The canonical polygons constructed by the algorithm are convex

Since each **DomVertex** $p \in \mathbb{H}$ is a covering of a **Slot** $s \in \mathbb{H}/\Gamma$ (under the natural universal covering map $\mathbb{H} \to \mathbb{H}/\Gamma$), there is a unique half-geodesic r_p ending at p that is a covering [9] of the **Road** adjacent to s.

The pair (p, r_p) will be considered as the *frame* associated to the **DomVertex** p.

That is why the *position* of a **DomVertex** will contain the following two members:

&Slot `slot`**:** A *reference* to the underlying **Slot** of the surface.

HFrame `frame`**:** The frame containing the (absolute) position of (p, r_p). The *origin* of this frame is $p \in \mathbb{H}$ and represents the *position* of the **DomVertex**.

The rest of the informations with respect to a **DomVertex** that one has to know are the adjacent **DomEdge**s, the **DomIdentification**s involving these **DomEdge**s and, finally, the **DomVertex**es that are identified with this one by these identifications. The following members will hold these informations (as references).

&DomEdge `nextEdge`**:** A *reference* to the *next* **DomEdge** on the (oriented) boundary. We must have that (expressed in python)

```
self.nextEdge.beginVertex is self #-> True
```

&DomEdge `prevEdge`**:** A *reference* to the *previous* **DomEdge** on the boundary,such that

---

[8] in the sense that it does not rely on numerical values

[9] at least locally

```
self.prevEdge.endVertex is self #-> True
```

`property`
DomVertex `nextVertex`**:** A *reference* to the *next* **DomVertex** on the (oriented) boundary. We must have that (expressed in python)

```
self.nextVertex is self.nextEdge.endVertex #-> True
```

`property`
DomVertex `prevVertex`**:** A *reference* to the *previous* **DomVertex** on the (oriented) boundary. We must have that

```
self.prevVertex is self.prevEdge.beginVertex #-> True
```

`property`
DomVertex `nextIdentified`**:** Let a denote the identification sending `nextEdge` to it's associated **DomEdge**. Then `nextIdentified` is the vertex that is identified to this one through a.

```
self.nextIdentified is self.nextEdge.facingEdge.endVertex #-> True
```

Since $a \in \Gamma$, these two points are identified. As a consequence we have

```
self.slot is self.nextIdentified.slot #-> True
```

`property`
DomVertex `prevIdentified`**:** The **DomVertex** identified to `self` through the `prevEdge`'s identification.

```
self.prevIdentified is self.prevEdge.facingEdge.beginVertex #-> True
```

`property`
DomVertex `identifiedCycle[]`**:** Let `self` be a **DomVertex**. This method (property) returns the list of vertices obtained by recursively appending the result of `currentVertex.nextIdentified`, starting with `currentVertex` as `self` until `currentVertex` is reached again. This will happen since the number of vertices is finite.

Note that this list is designed to contain `self` as the first element and the last one.

Here is an interesting property that arises in every **Domain**. For the notation we recall that `v.slot` means the slot on the surface $S = \mathbb{H}/\Gamma$ to which `v` is mapped under the natural covering $\mathbb{H} \to \mathbb{H}/\Gamma$.

Lemma 4. *Let* `vertex` *be a* **DomVertex** *of a* **Domain** `domain` *Then, for any other vertex* `v2` *of the domain,* `vertex.slot` *is* `v2.slot` *if and only if* `v2` $\in$ `vertex.identifiedCycle`.

Proof. Let $\phi \in \Gamma$ be the Möbius transformation obtained by taking the product of the `nextIdentification.transformation` associated to each of the elements of the `vertex.identifiedCycle` list for a certain `vertex` of a **Domain**.

Obviously, `vertex` is a fixed point of ϕ. Since Γ is a Fuchsian group, $\phi = \mathrm{Id}$.

From that, we can deduce that the sum of the *interior angles* associated to each of the vertices of a `vertex.identifiedCycle` list is 2π.

Since `domain` is a fundamental domain and $\mathbb{H} \to S = \mathbb{H}/\Gamma$ a covering map, 2π is also precisely the sum of the interior angles of the set of all vertices `v` for which `v.slot is vertex.slot`. This concludes the proof. ☐

Finally, the constructor of a **DomVertex** is declared as *private*. In our point of view, the *owner* of the **DomVertex**es is **Domain**, who creates them and deletes them while the algorithm is running.

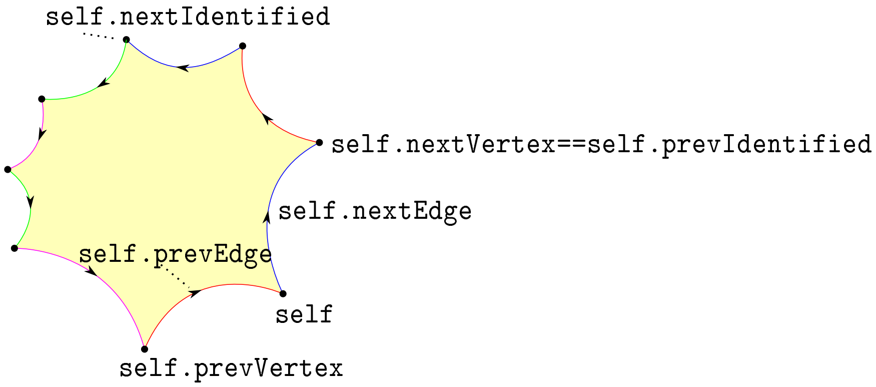


Figure 5.8: The members of **DomVertex** class, graphically.

5.4.2 The `DomEdge` class

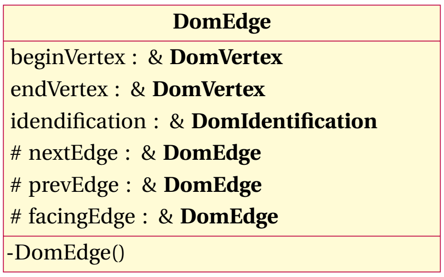


Figure 5.9: **DomEdge** class diagram.

The **DomEdge** class represents the *edges* of a **Domain**. Each edge is supposed to be an *oriented* geodesic arc starting on an explicit **DomVertex** called `beginVertex` and ending on a vertex called `endVertex`.

Furthermore, each **DomEdge** is involved in a unique **DomIdentification**, either as `beginEdge` or as `endEdge` (see page 85).

Finally, a *property* `facingEdge` is provided that returns the other **DomEdge** associated to the same identification, and two properties `nextEdge` and `prevEdge` which are, respectively the *next* and the *previous* **DomEdge**s on the oriented boundary.

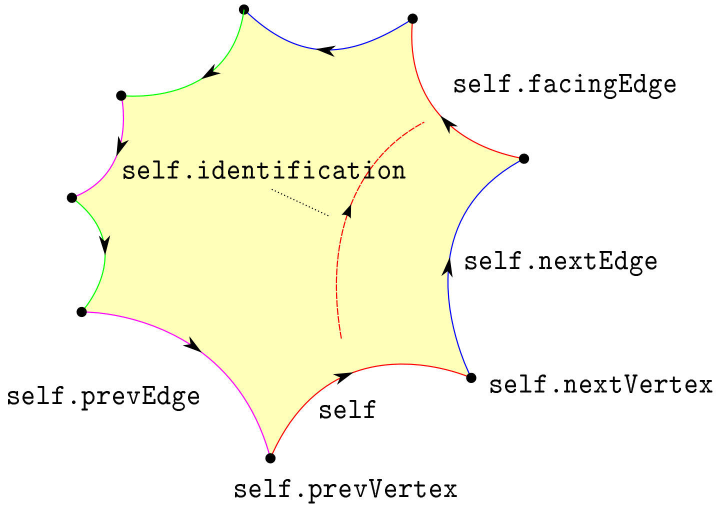
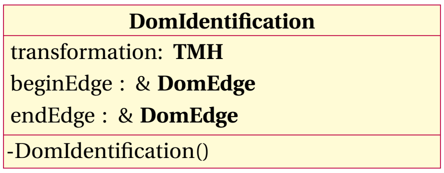


Figure 5.10: The members of **DomEdge** class, graphically.

5.4.3 The `DomIdentification` class



DomIdentification

transformation: **TMH**
beginEdge : & **DomEdge**
endEdge : & **DomEdge**

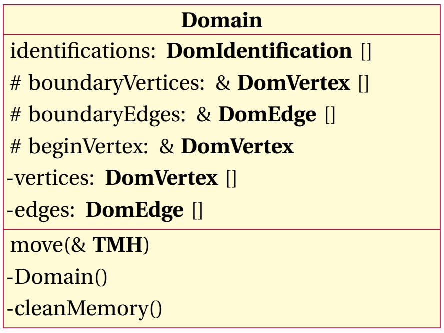
-DomIdentification()

Figure 5.11: **DomIdentification** class diagram.

By hypothesis a **Domain** is a *pairwise identified polygon*. Each pair of edges is then represented in our model by a **DomIdentification**, which is seen as *oriented* from `beginEdge` to `endEdge`.

For convenience, the **DomIdentification** also contains a Möbius transformation (represented by the class **TMH**). that is the *absolute transformation* sending `beginEdge` to `endEdge`. This *attribute* might also have been designed as a *property* to be deduced on the fly based on the `positions` of the *vertices*.

5.4.4 The `Domain` class



Domain

identifications: **DomIdentification** []
# boundaryVertices: & **DomVertex** []
# boundaryEdges: & **DomEdge** []
# beginVertex: & **DomVertex**
-vertices: **DomVertex** []
-edges: **DomEdge** []

move(& **TMH**)
-Domain()
-cleanMemory()

Figure 5.12: **Domain** class diagram.

The **Domain** class, certainly the most important one of the whole `HyperbolicDomain` module, represents the base class of any *pairwise identified domain* of a given **Surface**.

A **Domain** is characterized by its list **DomIdentification** that contains all the pairwise side `identifications`. The list is considered as orderless.

The remaining information that one needs to know are the lists of the **DomVertex**es, respectively the **DomEdge**s met along the oriented boundary. These lists are available through the `boundaryVertices` and `boundaryEdges` properties. They are thus just properties and not actual members of the **Domain**. We have designed it in this way because their content can be deduced from the content of the `identifications` list.

Finally, at least in the proposed implementation, the **Domain** class is supposed to be the actual *container* [10] of the **DomVertex**es and **DomEdge**s involved in it's boundary. For this reason the **Domain** class contains the two following *private* lists: `vertices` and `edges` and the *private* method `cleanmMemory()`.

In Section 5.6, we describe in details how a **Domain** can be modified (in place) step by step until it reaches the state of being a **CanonicalDomain**. This process will, once in a while, *destroy* some vertices of the boundary (but never create any). For this reason, from the memory management point of view, it is better to allow the lists `vertices` and `edges` to temporarily contain more vertices and edges than there are on the actual boundary. The supernumerary elements are destroyed only at the end of the process, by the `clanMemory()` method.

The **Domain** class has a *protected* constructor, the actually constructible domains are the *derived* classes of **Domain** i.e. the **BasicDomain** and the **CanonicalDomain** that are discussed in the next Sections.

The other derived classes of **Domain** are **HexDom**, **YDom** and **CollarDom**. These do not represent domains of a whole compact surface but domains of some subsets of the surface (hexagon, Y-*piece* and a collar respectively).

5.4.5 The `YDom` class

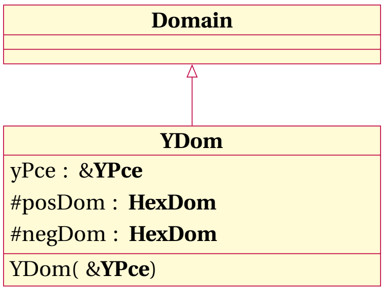


Figure 5.13: **YDom** class diagram.

The definition 19 of a *fundamental domain* can be generalized for the *subsets* of a surface with geodesic boundary. In this case we will rather speak of *sub-domains*.

Let $Y \subset S$ be an **YPce**.

The class **YDom** provides an explicit fundamental (sub-)domain for any given Y-*piece* in the following way.

First, we recall that any Y-*piece* can be cut into two isometric right-angled hexagons by cutting it along the roads u_0, u_1 and u_2.

---

[10] It posses a list containing the actual objects and not references to them and is supposed to free the memory every time the objects are deleted.

The shape of each of these two hexagons depends only on the length of the three borders of the *Y-piece* and can be computed using the formulas given in Section 3.3.

Each of these hexagons will be represented by objects of the class **HexDom** and are also sub-domains themselves.

Note that each vertex of these hexagons covers a **Slot** of the **Surface**.

If we travel around the *boundary* of the first **HexDom** in the counterclockwise direction (which is the orientation we choose on the Surface), we meet the following **Slot**s (on the right hand side in figure 5.14):

`b0.s0, b0.s1, b0.s2, b2.s0, b2.s1, b2.s2, b1.s0, b1.s1, b1.s2`

This **HexDom** is said to be *positive*. The other **HexDom** of the **YPce** will give the following slots, by following it's boundary counterclockwise (on the left hand side in figure 5.14)

`b0.s2, b0.s3, b0.s0, b1.s2, b1.s3, b1.s0, b2.s2, b2.s3, b2.s0`

By opposition, this **HexDom** will be called *negative*.

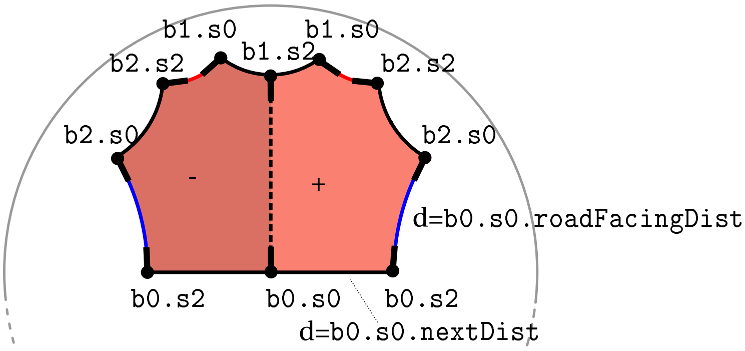


Figure 5.14: A fundamental domain of an *Y-piece* in $\mathbb{D}$. The two **HexDom**s are contrasted by a difference in shading.

Finally, by *gluing* the two **HexDom**s of an **YPce** along one of the **Road**s (here we took `u0`), we obtain a right-angled octagon like on figure 5.14.

This kind of octagon will be modeled by the class **YDom** and represent a *fundamental domain* of the corresponding **YPce**.

Now let us give a more technical description of the class **YDom** itself.

The *parent* class of **YDom** is **Domain**, by *inheritance*, **YDom** has a list of **DomVertex**s, a list of **DomEdge**s and a list of **DomIdentification**s.

As shown in Figure 5.14, an **YDom** consists at the end of its *construction* of 10 *vertices* and 10 *edges.*

Two of these 10 vertices have an *interior angle* of π, however they are still useful later, in Section 5.5, when we have to put together some **YDom**s with some **CollarDom**s in order to construct a **BasicDomain**.

The class **YDom** has the following attributes.

YPce`yPce`**:** A reference to the **YPce** of which this **YDom** is a *fundamental sub-domain.*

`property`
HexDom `posDom`**:** The *positive* **HexDom** of this **YPce**.

`property`
HexDom `negDom`**:** The *negative* **HexDom** of this **YPce**.

Constructor
YDom `(&`**YPce** `yPce)`

The constructor of the **YDom** class takes a (constant) reference to the **YPce** `yPce` of which it will be a *fundamental domain* as its only parameter.

The constructor is supposed do do the following things (we try here to avoid speaking of the implementation details). Figure 5.14 is a visual aspect of the result we achieve [11].

- Create 10 **DomVertex**es and attribute to them the following **Slot**s, in the same order as **DomVertex**.`slot`

  `b0.s0, b0.s2, b2.s0, b2.s2, b1.s0, b1.s2, b1.s0, b2.s2, b2.s0, b0.s2`

- Compute the **HFrame** of each of these **DomVertex**es and store these in the **DomVertex**.`frame` attribute.

  The **HFrame** we chose for the first **DomVertex** (the one that lies over the slot `b0.s0`) is the *canonical frame* $\vec{0}$ (see **DomVertex**.`frame` attribute).

  The other 9 **HFrame**s can then be deduced to be certain products of "vertical", "horizontal" and "half turn" transformations $V(d)$, $H(d)$, $O \in \mathbb{MH}$ as defined in Definition 14, on page 25.

---

[11] This picture is absolutely realistic in the sense that it has been produced by the "actual" implementation of the library that we try to describe here. It has nevertheless been finalized using the Inkscape freeware.

- Create the 10 **DomEdge**s of the **Domain**.
- Create 2 **DomIdentifications**s for the 2 pairs of **DomEdge**s that are *pairwise identified*. These are drawn in blue and red in Figure 5.14.

  The other 6 **DomEdge**s are not part of a pairwise identification.

5.4.6 The `HexDom` class

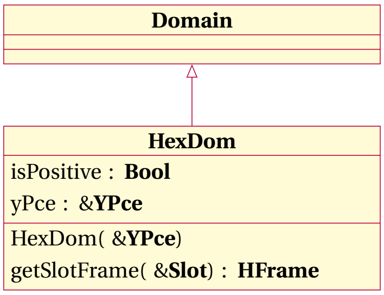


Figure 5.15: **HexDom** class diagram.

The **HexDom** is an hyperbolic right-angled hexagon which is a domain, one half of an **YPce**.

This class has already been explained in the class description of **YDom**, page 86.

5.4.7 The `CollarDom` class

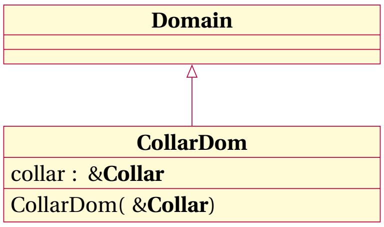


Figure 5.16: **CollarDom** class diagram.

Any **Surface** can be "decomposed" in $2g-2$ **YPce**s and $3g-3$ **Collar**s, which makes $5g-5$ subsets.

We can create a *sub-domain* for each **YPce** using the **YDom** class.

In a similar manner, each **Collar** will posses a *sub-domain* in the form of a **CollarDom**.

As shown in Figure 5.17, there are 6 **DomVertex**ex and then 6 **DomEdge**s in the boundary of a **CollarDom**.

The two **DomEdge**s that are drawn in blue are pairwise identified (the identification is by a “horizontal translation along the *main geodesic*”).

In the implementations the value of ϵ will be set to be zero. The domain is then degenerated with all the vertices aligned. All the operations to be carried out, however, will still be well defined.

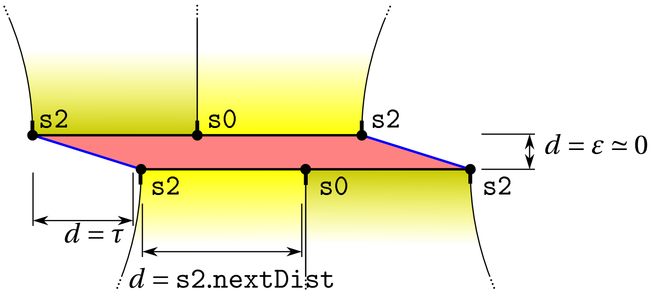


Figure 5.17: A fundamental domain **CollarDom** of a **Collar** in $\mathbb{D}$ is drawn in red. For the ease of exposition we gave a "width" to the **CollarDom**, but in reality it is totally flat and has zero area. In addition, two yellow **YDom**s have been "glued" to this **CollarDom** for better visualization. The two blue **DomEdge**s are pairwise identified.

5.4.8 The `BasicDomain` class

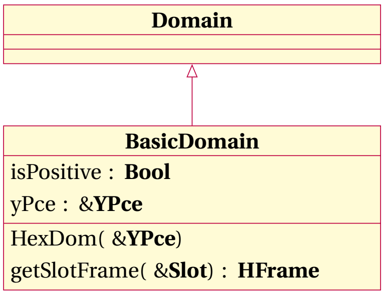


Figure 5.18: **BasicDomain** class diagram.

5.4.9 The `CanonicalDomain` class

In Theorem 6, we described the conditions that a *domain* of a surface must respect to be called *canonical*.

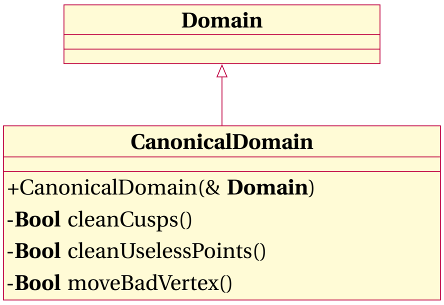


Figure 5.19: **CanonicalDomain** class diagram.

5.5 Basic Domain construction

The goal of this section is to provide the reader with a clear view of how it is possible to create a *fundamental domain* of a given **Surface** S.

The class **BasicDomain** will represent the **Domain**s that are obtained through this algorithm.

The **BasicDomain** class has an interest in itself since it remembers its sub-domains (**YDom**, **HexDom**, etc.); it can be useful to draw "tilings" of $\mathbb{H}$ in which each **YPce** has its own color for example.

Another example of usage of the **BasicDomain** is the creation of some "good" triangulations of the **Surface**, this is not treated here but we would like to point out the thesis of Marc Maintrot [8], where this is carried out.

Rather than detailing the precise implementation of the **BasicDomain** class, we will explain in geometrical terms how it works .

Since it is difficult to produce a satisfactory graphical illustration of what happens in $\mathbb{H}$ or $\mathbb{D}$, Figure 5.20 represents an Euclidean equivalent of the algorithm. In this example, the red triangles "behave" a little bit like **YDom**s and the green rectangles are metaphors of the **CollarDom**s. Finally, in this example Euclidean isometries play the role of the Möbius transformations that are used in the actual algorithm.

Nevertheless, for the hyperbolic situation Figure 5.21 shows the result of the creation of a **BasicDomain** in $\mathbb{H}$. The Fenchel-Nielsen graph that has been used is the `graph3` example given page 49.

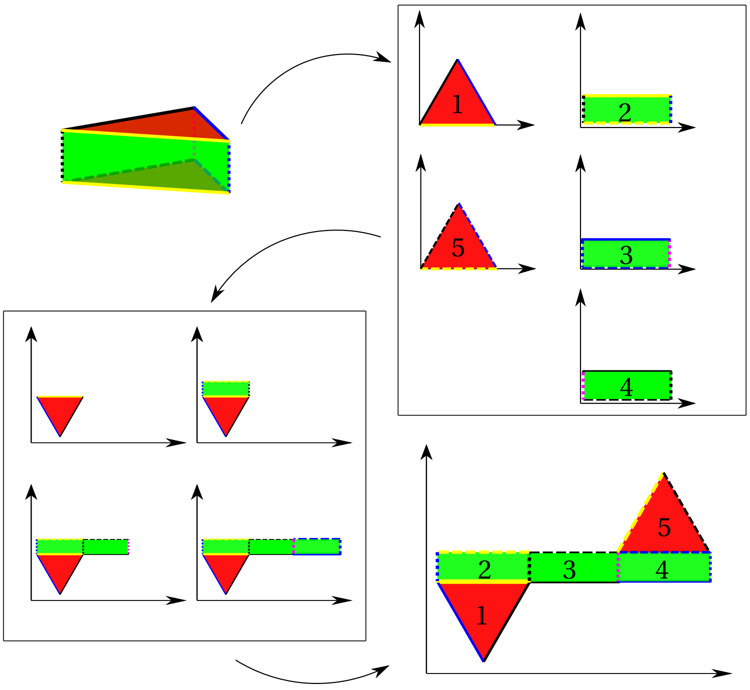


Figure 5.20: Metaphor of our algorithm, creating the *fundamental domain* of a prism in $\mathbb{R}^2$.

Step 1:

Firstly, for each of the $2g-2$ **YPce**s of the **Surface**, create an **YDom** (see page 86) and put these in a list `yDomsToPlace`.

Then, for each of the $3g-3$ **Collar**s of the **Surface**, create a **CollarDom** (see page 89) and put these in a list `collarDomsToPlace`.

These $5g-5$ *sub domains* are the "tiles" that we have to put together to form the **BasicDomain**, like one would solve a "jigsaw puzzle".

At this step, the *absolute positions* of these tiles do not matter. One can think of them as lying in their own copy of $\mathbb{H}$.

Step 2:

Each **YDom** possesses 6 "outer edges" which are **DomEdge**s that are not pairwise identified within the **YDom**. Each **CollarDom** possesses 4 "outer edges".

The given situation is is that *each* "outer edge" of the $2g-2$ **YDom**s is identified with a unique "outer edge" of one of the $3g-3$ **CollarDom**s.

Step 2 of this algorithm consists in completing these *identifications* by constructing the remaining $12g-12$ **DomIdentification**s

At the end of this step we then have a set of $5g-5$ *marked* polygons. These polygons, taken together, have *pairwise side identification* in the sense that any **DomEdge** of any of these $5g-5$ **Domain**s is *pairwise identified* with another such **DomEdge**; but the two members of a pair need not be in the same **Domain**.

The situation of this step is drawn as part B (upper right) in Figure 5.20.

Step 3:

“Select” one of the **YDom**s of the list `yDomsToPlace` (the first one for example). Copy it as the initial state of the **BasicDomain** we are creating. Then, since this *tile* has been *placed*, delete it from the `yDomsToPlace` list.

At the end of this step, the **BasicDomain** we are creating is then exactly like an **YDom**, with 6 *outer edges*, and 2 pairs of internally pairwise identified **DomEdge**s.

In our Euclidean allegory of Figure 5.20, this step is shown in the part C 1, where a single triangle has been "placed".

Remark 14. *At this point of our description we have not been specific about* how *the initial* **YDom** *should be* selected. *The idea is that different selection routines may be useful and added to the algorithm at a later stage depending on the particularly given geometrical situation. In Section 5.7 we discuss one such routine (see Remark 18, page 110).*

Step 4:

Scan the list **BasicDomain**.`edges` of the *edges* of the actual state of the **BasicDomain** we are building.

If among these **DomEdge**s you find an *outer edge* (that we will call `outer`), then go to Step 5.

If, on the other hand, there are no more *outer edges*, then the **BasicDomain** is now a valid *pairwise identified polygon*. At this moment it is complete and is a *fundamental domain* of the **Surface** S.

Step 5:

At the start of this step, we have an "incomplete" **BasicDomain** and an *outer edge* `outer` that has been found on its boundary.

Due to Step 2, we know that `outer` is pairwise identified with some other **DomEdge**, called here *outer2*.

The `outer2` edge is the edge of one of the "tiles" (sub-domain) that has still to be placed (it may be a **YDom** or a **CollarDom** depending on `outer`). Let us call `tileToPlace` this sub-**Domain** containing `outer2`.

Calculate the (unique) Möbius transformation $\psi \in M\mathbb{H}$ that brings the **DomEdge** `outer2` over the **DomEdge** `outer` (note that the orientation of the edge matters).

Apply the transformation ψ to each of the vertices of `tileToPlace` (to do that one may use the `tileToPlace.move(`ψ`)` method).

Copy the **DomVertex**es and the **DomEdge**s of `tileToPlace` into the **BasicDomain**.

Finally "glue" this new tile correctly to the **BasicDomain**, delete `outer` and `outer2` since they are not part of the boundary anymore. Also delete `tileToPlace` of the list of the tiles that still have to be placed.

After having done that go back to Step 4 until there are no more tiles to be placed.

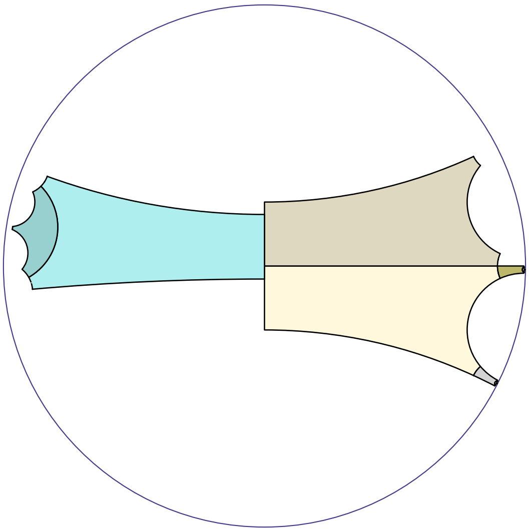


Figure 5.21: A **BasicDomain** in $\mathbb{D}$ corresponding to the `graph3` in Figure 4.1, page 49.

5.6 Canonical Domain construction

In this chapter, we assume that we have been able to create a *basic domain* i.e. a fundamental domain in the sense of Theorem 6, given in the form of a **BasicDomain** object, as described in the previous section.

The goal is now to "transform" it into a *canonical domain* that will be represented by a **CanonicalDomain** object.

The algorithm we would like to present for this consists of a succession of simple steps which themselves consist of "cutting" the current *fundamental polygon* into two parts, and then "moving" one part to "glue" it to the other in a different way.

In the course of this process, it may happen that the *non-degenerated polygon* that we are dealing with as a *fundamental domain* becomes *degenerated.*

For that matter we provide an "extended" definition of *fundamental domain* that coincides with the classical Definition 19 for *non-degenerated* pairwise identified marked polygons and which is also well defined for the *degenerated ones.*

This definition is the following.

Definition 33 (Fundamental polygon)**.** *Let $S = \mathbb{H}/\Gamma$ by a surface and let $\Pi : \mathbb{H} \to S$ be the associated projection.*

Let P be a marked pairwise identified polygon of $\mathbb{H}$, possibly degenerated.

Let $\Pi(P)$ represent the points of the surface that lie under a vertex or an edge (P is identified with it's trace and has a Lebesgue measure 0).

P is said to be a generalized fundamental polygon *of S if and only the following conditions hold:*

(1) All identifying transformations of P belong to Γ.

(2) If v, w are vertices of P that are not cusp or peak points, then $\Pi(v) = \Pi(w)$ if and only if v and w belong to the same identified cycle.

(3) $\forall s \in S \setminus \Pi(p), \quad \sum_{p\in\Pi^{-1}(s)} \alpha_P(p) = 2\pi.$

In the last item $\Pi^{-1}(s)$ is the orbit *of all the* lifts *of s in $\mathbb{H}$ and $\alpha_P(p)$ is the* winding angle *around p with respect to P as in Definition 29, page 75.*

In the following the term "*fundamental polygon*" shall always mean a *generalized fundamental polygon.*

Remark 15. *From Lemma 4 (page83) follows that the boundary of a* fundamental domain *in the sense of Definition 19 (page 67) is a* fundamental polygon .

Remark 16. *Any* fundamental polygon *that is* non-degenerated *is the boundary of a* fundamental domain *in the sens of Definition 19.*

Proof. If a *fundamental polygon P* is *non-degenerated* then, by the Jordan curve theorem (Remark13), it is the boundary of a simply connected domain $F \subset \mathbb{H}$, such that $\partial F = P$. The *winding angle $\alpha_P(p)$* can take the following values depending on the position of $p \in \mathbb{H}$.

$$\alpha_P(p) = \mathrm{or}_P \begin{cases} 2\pi & p \in F \\ 0 & p \notin F, \end{cases} \tag{5.6.1}$$

where or_P is the orientation of P. The condition on $\alpha_P(p)$ implies that $\mathrm{or}_P = 1$. Now let us first prove that $\bigcup_{g\in\Gamma} g(F) = \mathbb{H}$.

Let $p \in \mathbb{H} \setminus P$. Since P is a *fundamental polygon*, by definition, $\sum_{q\in[p]_\Gamma} \alpha_P(q) = 2\pi$. This means, combined with Equation 5.6.1, that there exists a point $q \in [p]_\Gamma$ with $q \in F$. This proves this first point.

Secondly, it remains to prove that $\forall x, y \in \mathrm{int}(F), \qquad [x]_\Gamma = [y]_\Gamma \Leftrightarrow x = y$.

By contradiction, one can show that if $\exists x, y \in \mathrm{int}(F)$ with $[x]_\Gamma = [y]_\Gamma$ and $x \neq y$, then we must have that $\sum_{z\in|x|_\Gamma} \alpha_P(z) \geq 4\pi$ since both $\alpha_P(x)$ and $\alpha_P(y)$ will appear in this sum. This is a contradiction, which concludes this proof. □

We proceed to detailing the algorithmic process that can progressively *transform* a **BasicDomain** into a *canonical* one.

Technically speaking, we introduce the **CanonicalDomain** class to represent this type of domain. The *constructor* of the **CanonicalDomain** is *public* and takes a **Domain** as argument.

Let us now describe the ingredients of this process. For each ingredient we shall prove that the new polygon obtained satisfies again the conditions of Definition 33.

Killing cusp and peak points

Algorithmically *cusps* and *peaks* will be treated in the same way and we shall use the term "*cusp*" for both.

Let `cuspVertex` be a **DomVertex**, along the boundary of some **Domain**, such that the following condition holds.

```
cuspVertex.nextIdentified is cuspVertex
#Equivalent to
cuspVertex.prevIdentified is cuspVertex
```

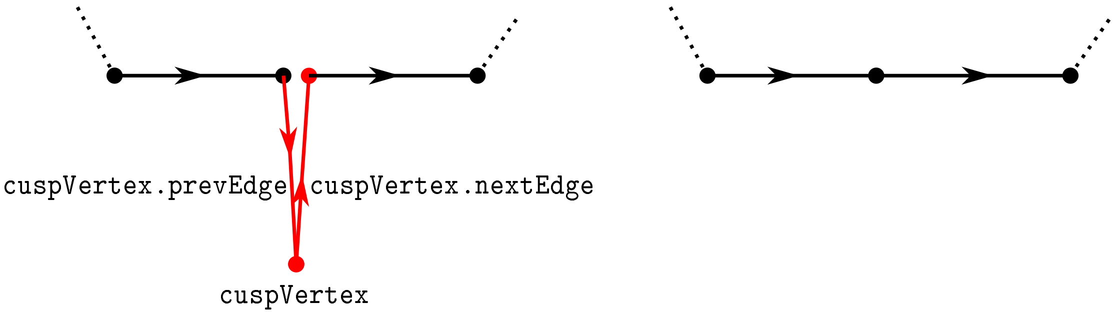


Figure 5.22: Deleting a *cusp vertex*.

Then `cuspVertex` is a *fixed point* under the action of the *Möbius transformation* `cuspVertex.nextIdentification.transformation`. Since, by hypothesis, we have a *fundamental polygon*, the *Fuchsian group* Γ contains this transformation.

The only possibility is then that this **DomIdentification** is the *identity* and that `cuspVertex.prexVertex` and `cuspVertex.nextVertex` coincide as points of $\mathbb{H}$.

It is then possible to remove `cuspVertex`, `cuspVertex.nextVertex`, `cuspVertex.prevEdge` and `cuspVertex.nextEdge` from the boundary, while suppressing `cuspVertex.prevIdentification` from the list `identifications` of the **Domain**.

To concretely achieve this goal here are the data that have to be modified, note that the new values are expressed in terms of the older state of the domain.

Attributes whose value changes.	New values
cuspVertex.prevVertex.nextEdge	cuspVertex.nextVertex.nextEdge
cuspVertex.nextEdge.nextEdge.beginVertex	cuspVertex.prevVertex
cuspVertex.prevIdentification	deleted from `domain.identifications`

Read with left bracketing, i.e. “((cuspVertex.nextEdge).nextEdge).beginVertex”, etc.

Note also that the "New values" have to be understood in terms of the *old* states of the attributes.

By taking out this flat triangle from the polygon, it is clear that we did not affect it’s geometry and that the new **Domain** obtained is still a *fundamental polygon* in the sense of Definition 33.

The **Domain** resulting from this action possesses two vertices less, two edges less and one identification less.

The **CanonicalDomain** will then implement the following method, based on the following algorithm:

Bool `cleanCusps()`

It's purpose is the following:

- For every **DomVertex** of the boundary,
- Check if it is a *cusp vertex*.
- Delete it as explained if this is the case.
- Restart the procedure until no more vertex of the polygon is a cusp vertex.

Finally the method returns `True` if some vertices were deleted and `False` if the polygon was already free of cusps since the beginning.

Killing useless vertices

We define a *useless vertex* to be a **DomVertex**, here called `uV`, that satisfies the following condition.

```
(uV.prevIdentified is not uV) and
    (uV.prevIdentified is uV.nextIdentified)
```

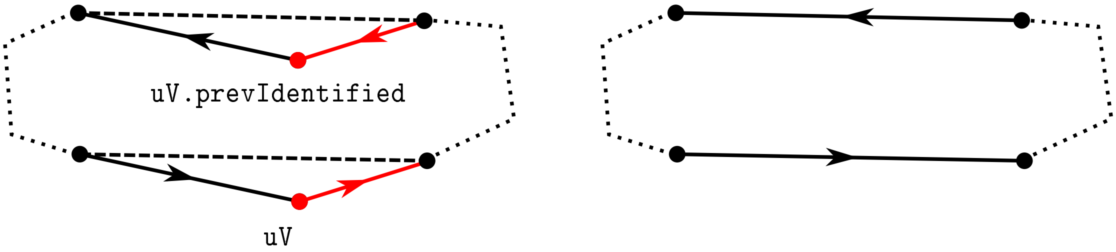


Figure 5.23: Deleting a *useless vertex*.

The consequence of this is that `uV.prevIdentification` brings the triangle

$$T = (\mathtt{uV}, \mathtt{uV.prevVertex}, \mathtt{uV.nextVertex})$$

to the triangle

$$T' = (\texttt{uV.prevIdentified},$$

$$\texttt{uV.prevIdentified.prevVertex},$$

$$\texttt{uV.prevIdentified.nextVertex})$$

(vertices 2 and 3 of T go respectively to 3 and 2 of T').

Let $P \subset \mathbb{H}$ be the current *fundamental polygon*. From the cut paste lemma that will be proved below it follows that $P' = T' \cup P \setminus T$ is also a fundamental polygon. The concrete modifications to the **Domain** data to achieve this operation are the following.

Attributes whose value changes.	New values
uV.prevEdge.endVertex	uV.nextVertex
uV.prevEdge.facingEdge.beginVertex	uV.prevIdentified.prevVertex
uV.nextIdentification	deleted from `domain.identifications`

According to this procedure **CanonicalDomain** implements the **Bool** `cleanCusps()` method. Like for the cusps, this method finds and destroys every existing useless point of the **Domain** and returns `True` if any points were deleted.

Moving a *bad* vertex

The definition of **CanonicalDomain** stipulates that all the vertices of its boundary belong to the same equivalence class (i.e. to coverings of the same **Slot**).

Let us assume that `baseSlot` is a **Slot** of a surface and `domain` a **Domain** containing at some point n vertices among which exactly m, with $0 < m < n$, lie in the equivalence class of `baseSlot` (this information is held by the `vertex.slot` member).

Without loss of generality we may assume that `domain` has already been set in a state where it contains neither *cusps* neither *useless vertices*.

The algorithm to be described here transforms `domain` into a new fundamental polygon also containing n vertices, but this time $m+1$ of them will lie over `baseSlot`.

Firstly, we scan the **DomEdge**s of the `domain.boundaryEdges` until we find an edge `edgeA`$= \vec{e}_A$, satisfying the following condition.

```
(edgeA.beginVertex.slot is baseSlot) and (edgeA.endVertex.slot is not baseSlot)
```

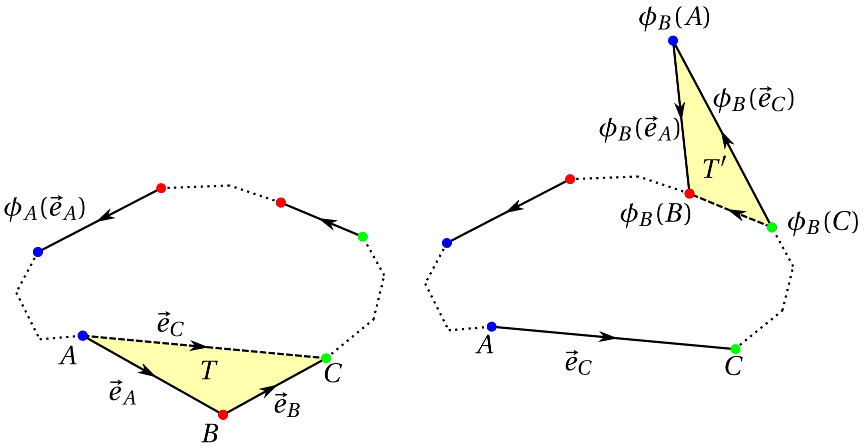


Figure 5.24: Getting rid of a *bad* vertex. In this picture, the colors indicate in which equivalence class each vertex lies. The red class is supposed to be distinct from the blue one.

We define $\vec{e}_B$:=`edgeB=edgeA.nextEdge` and we define A, B, C or `vertexA, vertexB, vertexC` to be `edgeA.beginVertex`, `edgeA.endVertex` and `edgeB.endVertex` respectively. Finally, we define $\phi_A, \phi_B \in \Gamma$ to be the Möbius transformations associated with the identifications of $\vec{e}_A$ and $\vec{e}_B$, as in Figure 5.24.

The vertex B does not lie in the "good" equivalence class, that is why we choose to call such a vertex a *bad vertex*.

We can now create a "new" edge $\vec{e}_C$ from A to C.

Let P be the actual *fundamental polygon*.

The algorithm presented here is supposed to "remove" T from P and replace it by $T' := \phi_B(T)$, as in Figure 5.24. That the new polygon P' obtained in this way satisfies the conditions of Definition 33 will again follow from the cut paste lemma.

Looking at Figure 5.24 we see that P' contains exactly the same vertices as P with the exception of the vertex B which did not lie over `baseSlot` and has been replaced by the new vertex $\phi_B(A)$ which does.

The "operation" translates itself into a method of the class **CanonicalDomain**:

Bool `moveBadVertex(&`**DomVertex** `baseVertex)`

If this method finds a *bad* vertex of the *fundamental polygon*, then it transforms the polygon as explained above and returns `True`; if, however, every vertex of the polygon is in the same equivalence class as `baseVertex`, then the method returns `False`.

We now provide a table showing what data have actually to be changed during this operation. The terms `edgeA`, `vertexA`, etc. are as defined above.

Attributes whose value changes.	New values
vertexA.nextEdge	edgeB
vertexB.nextEdge	edgeA
vertexB.prevEdge	edgeB.facingEdge
vertexB.nextIdentified.prevEdge	edgeA
edgeA.beginVertex	vertexB
edgeA.endVertex	vertexB.nextIdentified
edgeB.beginVertex	vertexA
edgeB.facingEdge.endVertex	vertexB
vertexB.position	modified to contain $\phi_B(A)$.
edgeA.identification.transformation	modified to contain $\phi_A\phi_B^{-1}$.

On the memory management point of view this method *reuses* all the already existing objects, the **DomVertex** used to represent the vertex B now represents $\phi_B(A)$, the **DomEdge**s are "reused" according to the following pattern :

$$B \to \phi_B(A), \qquad \vec{e}_A \to \phi_B(\vec{e}_A), \qquad \vec{e}_B \to \vec{e}_C, \quad \phi_B(\vec{e}_B) \to \phi_B(\vec{e}_C)$$

$$\phi_A \to \phi_A\phi_B^{-1}, \qquad \phi_B \to \phi_C = \phi_B$$

Bringing every vertex to the same equivalence class

Let us now describe the first steps of the *constructor* of the **CanonicalDomain** class.

The argument only needs to be a reference to a **Domain** (usually it will be a **BasicDomain** since it is the only kind that is constructible directly from a **Surface**).

The first step of the construction consist of making a "deep" copy of this **Domain** into the current object. We do not describe exactly how to achieve this copy since it is quite straightforward.

Then, the current object is modified in place using the following loop (explicitly given in `Python` for clarity)

```
bool1 = bool2 = bool3 = True
#This loop will restart until no more change happens in the Domain.
while(bool1 or bool2 or bool3):
    #Destroy every cusp vertices and return True if any were found.
    bool1 = self.killCusps()
    #Destroy every useless vertices and return True if any were found.
    bool2 = self.killUseless()
    #Move one bad vertex to a vertex equivalent to baseVertex  and return True if any
      were found.
    bool3 = self.moveBad(baseVertex)
```

When the loop stops, the actual state of the **CanonicalDomain** under construction will be a *a fundamental polygon* of the surface, containing no *cusp vertices*, nor useless vertices. Furthermore, every vertex will lie in the same equivalence class, the same as `baseVertex`.

We just still have to justify why we can guarantee that it indeed stops.

Let $N(i)$ and $B(i)$ denote the total number of vertices, respectively the total number of *bad* vertices after i iterations of the loop.

We have to show that $\exists k \in \mathbb{N}, \quad B(k) = 0$, because then the algorithm must stop.

At each loop, some vertex can possibly be killed, but none of the 3 methods can increase the number of vertices. Thus, $N(j+1) \leq N(j)$.

Since the property of being a *fundamental polygon* is preserved by the loop we know that N cannot in any way go below 3 ($4g$ is probably the minimum).

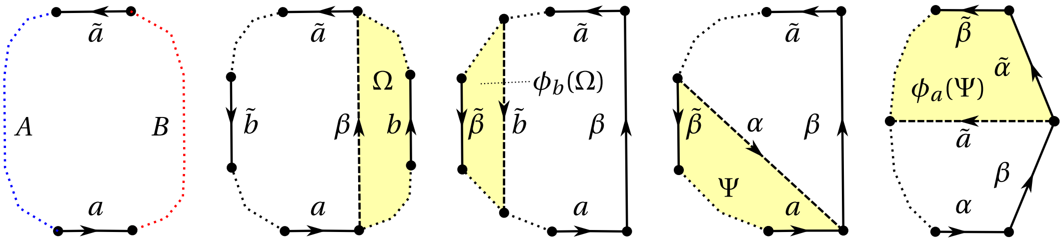
Since N is bounded by below, $\exists j^* \in \mathbb{N}, \quad N(j^*) = N(t), \forall t \geq j^*$.

If we take $k = j^* + B(j^*)$, we can see that $B(k) = 0$, which concludes this proof.

Rearranging the Domain such that it has a *canonical signature*

Theorem 6 states, among other things, that the *signature* of a *canonical domain* is itself *canonical* in the sense that it is of the form $\quad aba^{-1}b^{-1}cdc^{-1}d^{-1}\ldots$.

At this point, we have shown that, for a given surface, it is always possible to obtain a valid **Domain** of it such that every vertex lies over the same **Slot**. We will assume that these operations have been carried out and explain the last steps of the transformation of the **Domain**.



(a) (b) (c) (d) (e)

Figure 5.25: Steps to obtain a *canonical* signature.

Let `edgeA`, `edgeB`, `edgeC` and `edgeD` be four consecutive **DomEdge**s of the boundary.

If they are such that the following condition holds, then these edges are here said to form a *quartet*.

```
(edgeA.facingEdge is edgeC) and (edgeB.facingEdge is edgeD)
```

If the edges "only" respects the following condition

```
(edgeA.facingEdge is edgeC) and (edgeB.facingEdge is not edgeD)
```

then they are called here a *trio*.

Figure 5.25 shows different steps of the transformation of the domain. The edges have been labeled $a, b, \dots$ while their *facing edges* are denoted by $\tilde{a}, \tilde{b}, \dots$ for convenience.

Figure (e) shows an example where the edges $\alpha, \beta, \tilde{\alpha}, \tilde{\beta}$ form a *quartet*. The edges $a, \beta, \tilde{a}$ in figure (c) and (d) form a *trio*.

The *signature* of a polygon is *canonical* if and only if its edges consist of a certain [12] number of consecutive *quartets*.

The procedure to obtain a valid **CanonicalDomain** is as follows.

i) Take `edgeA` (or simply a) to be an edge of the polygon (the first one at the start of the algorithm).

   If this edge is the first edge of a *quartet* jump to step iv). If this edge is the first edge of a *trio* jump to step iii). Else, go to step ii).

ii) If `edgeA` is neither the first edge of a *quartet* nor the first edge of a *trio*, then it is possible to find an edge here referred to as `edgeB` (or b) such that b lies between a and $\tilde{a}$ and `edgeB.facingEdge` (or $\tilde{b}$) lies beyond the `edgeA.facingEdge`, $\tilde{a}$, on the oriented polygon like on Figure (b).

   To guarantee the existence of such an `edgeB`, we can argue by contradiction, assuming that we are in the case of Figure (a), where each edge of the part A of the polygon is identified with an edge of the part A and each edge of the part B is identified with an edge of B.

   Then, the list `edgeA.endVertex.identifiedCycle` does only contain vertices in B and must end with the vertex `edgeA.facingEdge.beginVertex`. This list then cannot contain `edgeA.beginVertex`.

   This is a contradiction with point 2 of Definition 33 and the assumption that the domain was set in a state where every vertex lies over the same slot of the surface.

   It is then possible to "create" a new edge β joining `edgeA.endVertex` to `edgeA.facingEdge.beginVertex`, defining a new *sub-polygon* Ω like in Figure (b).

---

[12] Due to topology, it is known that this number must precisely be g, the *genus* of the surface.

The **Domain** is then modified by a *cut and paste* procedure quite similar to the one described in the `moveBadVertex` method. The result, shown in Figure (c), is a new **Domain**, where Ω has been replaced by $\phi_b(\Omega)$, ϕ_b being the identification associated to edge b.

Note that on the implementation point of view it is possible to apply this transformation without destroying or creating any *object*, indeed, the **DomEdge**s used to represent b and $\tilde{b}$ can now be used to represent β and $\tilde{\beta}$, while the **DomIdentification** ϕ_b is now used to represent ϕ_β.

On the geometrical point of view, this operation preserve every constraints that the former **Domain** was satisfying, and now the edge a is at the start of a *trio* $a\beta\tilde{a}$, which allows us to jump to the next step.

iii) Let a be at the start of a *trio* $a\beta\tilde{a}$ as in figure (d).

Then, it is possible to create a new edge α joining the *begin vertex* of $\tilde{\beta}$ to the *end vertex* of a, defining a *sub-polygon* Ψ shown in Figure (d).

As in the previous step, a simple *cut-past* procedure allows us to replace Ψ by $\phi_a(\Psi)$, as shown in Figure (e). The edges $\alpha, \beta, \tilde{\alpha}, \tilde{\beta}$ now form a *quartet*.

iv) None of the previous steps changes the number of edges of the domain, but now 4 more edges are adjacent and form a *quartet*.

Take the *next edge of* $\tilde{\beta}$ as the new `edgeA` and return to step i) until you reach the end of the `domain.vertices`.

At the end of this process, the **CanonicalDomain** will have exactly $4g$ edges[13]. Furthermore, the *signature* of the *domain* is of the form $aba^{-1}b^{-1}cdc^{-1}d^{-1}\ldots$.

We come to the proof that the algorithm produces a *fundamental polygon* at each step:

Lemma 5 (Cut-Paste 1)**.** *Let P be a* fundamental polygon *and P' the polygon obtained by applying one of the above operations. Then P' is again a* fundamental polygon.

Proof. We check properties (1), (2), (3) of Definition 33.

Since all new identifying Möbius transformations are products of previous ones they all belong to Γ. This proves that property (1) remains valid.

In all operations an *identified cycle* may either obtain more members or loose members or disappear completely. But no new cycles are ever created. Therefore (2) remains also valid.

The operation "killing cusp and peak points" has no effect on the "Umlaufszahlen", so this operation does not change the validity of (3).

---

[13] This can be proven by purely topological arguments and is well known.

To look at the remaining operations we decompose them into *elementary* ones. The first is to *insert a cut*: Let $P = [v_1, \ldots, v_n]$ be the oriented polygon. For simplicity we describe a cut from v_1 to v_k with $3 \le k \le n-1$. It consists of replacing P by the two separate polygons $P' = [v_1, \ldots, v_k]$ and $P'' = [v_1, v_k, v_{k+1} \ldots, v_n]$. By the definition of α_P we have

$$\forall p \in \mathbb{H} \quad \alpha_P(p) = \alpha_{P'}(p) + \alpha_{P''}(p) \tag{5.6.2}$$

The second elementary operation is to *remove* a cut. It is the inverse of the first.

The third elementary operation is to *apply a member of* Γ to the vertices of P. It consists of replacing a polygon $Q = [w_1, \ldots, w_m]$ by $\phi(Q) = [\phi(w_1), \ldots, \phi(w_m)]$ for some $\phi \in \Gamma$. It follows from the definition of α_Q and the fact that ϕ is a direct isometry that

$$\forall p \in \mathbb{H} \quad \alpha_{\phi(Q)}(\phi(p)) = \alpha_Q(p). \tag{5.6.3}$$

For any $s \in S = \mathbb{H}/\Gamma \setminus \Pi(Q)$ this implies

$$\sum_{p \in \Pi^{-1}(s)} \alpha_{\phi(Q)}(p) = \sum_{p \in \Pi^{-1}(s)} \alpha_{\phi(Q)}(\phi(p)) = \sum_{p \in \Pi^{-1}(s)} \alpha_Q(p). \tag{5.6.4}$$

Since all the remaining operations to be considered in this proof are product of elementary ones it follows from (5.6.2) and (5.6.4) that they preserve the validity of (3). □

In the next section we prove that the resulting polygon is convex., here we finish with a few lines about the *calculability* of this algorithm.

Remark 17. *We add a few comments about the* calculability *of this algorithm which addresses objects that involve numerical values.*

1. ***All*** *boolean decisions that have to be taken (during the building of a* **BasicDomain** *as well as during the steps of its "transformation" into a* **CanonicalDomain***) are always based on comparisons of* pointers*, but not on the actual* values *of the positions of the vertices, which are usually numerical.*

2. The positions *of the vertices and the* identification transformations *are represented by some* Möbius transformations*, represented here by the class* **TMH** *(Hyperbolic Möbius Transformation).*

3. Depending on the context, the class **TMH** *can implement either numerical values or, if needed, the whole process can also be carried out by real algebraic numbers which are exact.*

5.7 Convexity of the polygon

The material of this Section is the subject of a forthcoming joint paper with Peter Buser.

The main question asked is whether the Canonical Domain construction algorithm actually produces a *convex fundamental polygon* which is then the boundary of a convex fundamental domain as in Theorem 6, page 78.

We would have liked to investigate this on the basis of the conditions of Definition 33, page 96. But this has not worked out, so far. We present therefore an approach that goes a second time through the classical proof of the *Surface Classification Theorem* (e.g. in Massey [9, Chapter I]) and then uses isotopy arguments borrowed from [4, Appendix].

Firstly we show the following.

Lemma 6 (Cut-Paste 2)**.** *At each step of the algorithm where the current* fundamental *polygon P has no cusp or peak points there exists a fundamental domain F of Γ (in the sense of Definiton 19, page 67) with the following properties*

(1) The vertices of P lie on ∂F.

(2) For any edge $\overrightarrow{v_i v_{i+1}}$ $(i = 1, \ldots, n,\ v_{n+1} := v_1)$ of P there exists a piecewise geodesic arc $f_i \subset \partial F$ homotopic to $\overrightarrow{v_i v_{i+1}}$ (with fixed endpoints v_i, v_{i+1}).

(3) ∂F is the curve $f_1 f_2 \cdots f_n$.

(4) P and F have the same side paring transformations.

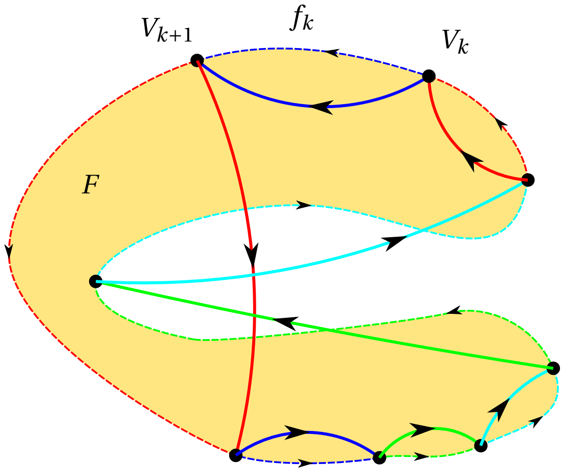


Figure 5.26: *Fundamental polygon inscribed in a fundamental domain*

Proof. This is true at the beginning of the algorithm, by hypothesis.

Now we assume by induction that at some step of the algorithm the polygon P and the domain F are as in the lemma, and that the algorithm is going to perform one of the "operations" described in Section 5.6 and then afterwards will be removing possibly occurring peak and cups points. Let $\tilde{P}$ designate the then resulting polygon. We must find some $\tilde{F}$ for it.

Case 1. The operation is "killing a useless vertex". This consists of replacing a couple of consecutive edges $\overrightarrow{v_{i-1}v_i}$, $\overrightarrow{v_iv_{i+1}}$ by the single edge $\overrightarrow{v_{i-1}v_{i+1}}$ ($v_{1-1} := v_n$, $v_{n+1} := v_1$) and some corresponding couple $\overrightarrow{v_{k-1}v_k}$, $\overrightarrow{v_kv_{k+1}}$ by $\overrightarrow{v_{k-1}v_{k+1}}$.

In this case we set $\tilde{F} = F$ and consider the composed arcs $f_{i-1}f_i$ and $f_{k-1}f_k$ each as just one side of $\tilde{F}$. We can easily verify that $\tilde{F}$ has all the required properties.

Case 2. The operation is one of the remaining ones. We may assume that P has no peak and cusp points and no useless vertices.

As in the proof of Lemma 5 the operation is a product of three elementary operations: 1. insert the arc $\overrightarrow{v_1v_k}$ (with $3 \le k \le n-1$) to obtain $P' = []v_1, \dots, v_k]$, $P'' = [v_1, v_k, v_{k+1}, \dots, v_n]$; 2. then move P'' to $\phi(P'')$ with the appropriate $\phi \in \Gamma$; 3. attach $\phi(P'')$ to P', say along the edge $\overrightarrow{v_mv_{m+1}} = \phi(\overrightarrow{v_lv_{l+1}})$ and remove this edge.

We apply the same procedure to F: Since F has pecewise geodesic boundary and is homeomorphic to a disk, we can insert a piecewise geodesic arc f_{1k} in F going from v_1 to v_k such that $f_{1k} \setminus \{v_1, v_k\} \subset \mathrm{int}(F)$. It separates F into two domains F', F''. By the Jordan curve theorem F' and F'' are both homeomorphic to a disk. Like P and F the couples P', F' and P'', F'' both have the properties (1), (2), (3) of the statement of the Lemma.

Now we move F'' to $\phi(F'')$. By property (4) of F we have $\phi(f_l) = f_m$. This means that $\phi(F'')$ is attached to F' along side f_m. We set

$$\tilde{F} = F' \cup \phi(F'')$$

and then have $f_m \setminus \{v_m, v_{m+1}\} \subset \mathrm{int}\tilde{F}$. We can easily see that the pair $\tilde{P}, \tilde{F}$ has the properties (1)–(4) of the Lemma. (One must use that P has no useless vertices and so the only sides that are "lost" in the process of building the union $F' \cup \phi(F'')$ are the coinciding sides f_m of F' and $\phi(f_l)$ of $\phi(F'')$.)

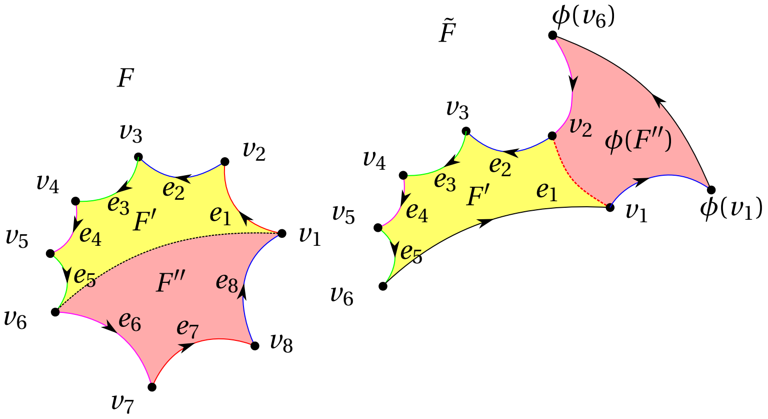


Figure 5.27: Cutting and pasting a part of a *fundamental polygon*

Finally, we show that $\tilde{F}$ is again a *fundamental domain* of S:

- $\bigcup_{g\in\Gamma} g(\tilde{F}) = \mathbb{H}$:
  Let $p \in \mathbb{H}$ be a point. Since F is, by hypothesis, a *fundamental domain*, $\exists \gamma \in \Gamma$ such that $q := \gamma(p) \in F$. If $q \in F'$ then $\gamma(p) \in \tilde{F}$ and $p \in \bigcup_{g\in\Gamma} g(\tilde{F})$. If $q \in F''$ then $\phi(q) = \phi(\gamma(p)) \in \phi(F'') \subset \tilde{F}$, which concludes the point.

- $\forall x, y \in \text{int}(F') \cup \text{int}(\phi(F'')), \qquad [x]_\Gamma = [y]_\Gamma \Rightarrow x = y$:
  -If $x, y \in \text{int}(F')$, then $x = y$ because $\text{int}(F') \subset \text{int}(F)$.
  -If $x, y \in \text{int}(\phi(F''))$ then $\phi^{-1}(x), \phi^{-1}(y) \in \text{int}(F'')$, therefore $\phi^{-1}(x) = \phi^{-1}(y)$ and $x = y$.
  -If $x \in \int(F')$ and $y \in \int(\phi(F''))$, then $\phi^{-1}(y) \in F''$. Since$[x]_\Gamma = [\phi^{-1}]_\Gamma$ and F is a Fundamental domain this last case cannot occur.

Since $\text{int}(F') \cup \text{int}(\phi(F''))$ differs from $\text{int}(\tilde{F})$ only by finitely many arcs this is sufficient to show that $\tilde{F}$ is a fundamental domain.

The proof of the Lemma is now complete. □

For the rest of this Section $P = [v_1, \dots, v_{4g}]$ is the canonical fundamental polygon for $S = \mathbb{H}/\Gamma$ resulting from the algorithm in Section 5.6.

By point (2) of the Lemma both, the polygon P and the canonical fundamental domain F have the same side pairing transformations. Since F is canonical we get the following.

Corollary 3. *The side pairing transformations of P form a set of canonical generators of* Γ.

We come to the question of when P is itself the boundary of a "classical" fundamental domain. For this we shall prove the following.

Lemma 7. *For* $k = 1,\dots,4g$, *consider the Möbius transformation* $\gamma_k \in \Gamma$ *that sends vertex* v_1 *of* P *to vertex* v_k. *Select a point* w_1 *on the axis of* γ_2 *and set*

$$w_k = \gamma_k(w_1), \quad k = 1,\dots,4g; \quad P^* = [w_1,\dots,w_{4g}].$$

Then P^* *is the sequence of vertices of a convex geodesic canonical fundamental domain* F^* *of* $S = \mathbb{H}/\Gamma$.

Remark 18. *In Remark 14, page 94, we have mentioned that one may add to the canonical domain algorithm suitable selection routines for the initial slot on the boundary of the basic domain.*

In our implementation the slot is selected on a side of the basic domain that corresponds to a non-separating *main geodesic of* S. *A search routine along the boundary of the domain carries this out. It then turns out that on the resulting polygon* P *vertex* v_1 *lies automatically on the axis of* γ_2. *Hence we have the following.*

Corollary 4. *With a suitable selection of the initial slot for the canonical domain algorithm the resulting polygon* P *is the boundary of a convex geodesic canonical fundamental domain.*

For the proof of Lemma 7 we use isotopies.

Definition 34. *An* isotopy *of a topological space* X *is a* continuous *mapping* $J : [0,1] \times X \to X$ *with the following properties:*

1. $J(0,p) = p, \quad \forall p \in X$.
2. *For any* $s \in [0,1]$ *the mapping* $j_s : X \to X$ *given by*

   $$j_s(p) = J(s,p), \quad p \in X$$

   is a homeomorphism.

Isotopies provide particularly interesting homotopies of curves: If $c : [a,b] \to X$ is a curve then we may set

$$H(s,t) = J(s,c(t)), \quad s \in [0,1], t \in [a,b].$$

This may be applied to several curves simultaneously. If, e.g. $c_1,\dots,c_n$ are pairwise disjoint simple closed curves and we apply an isotopy, then the deformed curves are again pairwise disjoint and simple.

Proof. (Lemma 7) We recall that for $k = 1,\dots,4g$ we have the arcs f_k from v_k to v_{k+1} ($v_{4g+1} := v_1$) homotopic to the edges $e_k = \overrightarrow{v_k v_{k+1}}$. Under the projection $\Pi : \mathbb{H} \to S = \mathbb{H}/\Gamma$ the curves f_k are mapped to simple closed arcs (loops) $\alpha_k = \Pi(f_k)$, $k = 1,\dots,4g$.

They intersect each other at a common base point $v = \Pi(v_1) = \cdots = \Pi(v_{4g})$. The loops $\alpha_1, \alpha_2, \alpha_5, \alpha_6, \ldots, \alpha_{4g-3}, \alpha_{4g-2}$ represent generators of the fundamental group of S.

Let a_1 be the closed geodesic in the free homotopy class of α_1. Then the axis $\tilde{a}_1$ of the Möbius transformation $\gamma_2 \in \Gamma$ that sends v_1 to v_2 is a lift of a_1. Hence, the point $w := \Pi(w_1)$ lies on a_1.

Now we make use of isotopies. It is shown in the proof of Theorem 6.7.4, page 166 of [4] that there exists an isotopy $J : [0,1] \times S \to S$ such that for the final homeomorphism

$$j : S \to S, \quad j(p) = J(1,p), \forall p \in S$$

the following points are satisfied:

1. $j(\alpha_1) = a_1$.
2. $j(v) = w$.
3. $a_k := j(\alpha_k)$ is a geodesic loop with base point w, $\forall k = 1, \ldots, 4g$.
4. The complement $S \setminus \bigcup_{k=1}^{4g} a_k$ is a convex geodesic polygon.

J induces free homotopies of curves: Assuming that all α_k are represented as parametrized curves $\alpha_k : [0,1] \to S$ these homotopies are

$$H_k(s,t) = J(s, \alpha_k(t)), \quad s \in [0,1], t \in [0,1]$$

Now look at the curve

$$c(s) = J(s,c) = H_1(s,0) = \cdots = H_{4g}(s,0), \quad s \in [0,1].$$

It leads from v to w. Lifting the homotopy H_1 to the universal covering $\mathbb{H}$ we get a covering curve c_1 of c going from v_1 to w_1 and a covering curve c_2 of c with initial point v_2. Since Γ acts without fixed points and since $\gamma_2(v_1) = v_2$ we have $c_2 = \gamma_2(c_1)$. We conclude that the endpoint of c_2 is $\gamma_2(w_1) = w_2$. The geodesic arc $\Pi(\overrightarrow{w_1 w_2}$ is therefore homotopic, with base point w fixed to a_1. By the uniqueness of geodesic arcs in such homotopy classes we must have $\Pi(\overrightarrow{w_1 w_2} = a_1$.

In other words, $\overrightarrow{w_1 w_2}$ is a lift of a_1. Proceeding in the same manner we se that $\overrightarrow{w_k w_{k+1}}$ is a lift of a_k for all k. By point 4. above, P^* is the set of vertices of a convex geodesic polygon.

□

6 Exploration algorithms

One of the most useful informations on a *Riemann surface* is the set of the closed geodesics on it. The important feature is that this information is purely intrinsic to the global geometry of the surface and does not depend on the choice of the marking.

Among the closed geodesics the *simple* ones, i.e. those without self-intersections give us even more information.

Finally, the set of the *systoles*, i.e. the shortest closed geodesics of the surface, (which are always simple for obvious reasons) almost constitutes the *DNA* of the surface.

For example, knowledge of the systoles can be useful to discover the non-trivial automorphisms of a given surface.

In this chapter, we describe a procedure, very inspired by Peter Buser's ideas [3], that provides a very efficient and useful way to "code" the simple closed geodesics of a given marked surface.

Then, we will explain how we can enumerate the different closed geodesics of a surface. The algorithm is quite general and we can adapt the "stop conditions" of it so as to obtain the systoles of a surface, or the set of all closed curves shorter than a given length.

The question of how to enumerate the simple closed geodesic is intrinsically connected to the question of their representation (in term of classes).

Here is a summary of how we proceed in this chapter.

1. We firstly assume that one knows everything about S and about a simple closed geodesic $\gamma \subset S$ (in the sense that these objects are given geometrically). Starting from these informations we introduce a new data structure called **Path** which comes along with a number of *sub-classes*.

   Then, we explain how we theoretically could translate the geodesic γ "by hand" into a valid **Path** object, the result being uniquely determined by γ and by the structure of the `surface` object modeling S.

2. We describe a list of constraints that a `path` object of type **Path** is supposed to respect in order to be called *simple* (or simply *valid*[1]).

   We also provide a condition that `path` should respect to be called *closed*.

   As an important fact, one can show that any **Path** deduced from an actual *simple closed geodesic* $\gamma \subset S$, as in the previous item will itself be *simple* and *closed* as a **Path**.

3. Then, conversely to item1, we assume that a **Path** object `path` is given and that it is both *simple* and *closed*.

   On the one hand we explain how a unique *closed* geodesic $\gamma \subset S$ as well as any of its geometrical properties can be deduced from the discrete data stored in `path`.

   On the other hand, we show how a closed *piecewise geodesic* curve $\gamma_\epsilon \subset S$ homotopic to γ can be deduced. This γ_ϵ will be shown to be *simple*, from which we can deduce that the geodesic γ itself is simple.

   According to that we can conclude that there is a *bijection* between the *simple closed geodesics* of S and the *simple closed* **Path** objects that can be constructed.

4. Under the hypothesis that a *valid* **Path** object is given, we show how to translate it into a *codeword* object of the form

   (&**Slot**, `Int[]`)

   through the method **Path**.`toCode()`.

   Any codeword that can be obtained from a *valid* **Path** is defined to be a *valid code word*.

   We see later that there is a *bijection* between the *valid* **Path**s and the *valid codewords*.

5. At first glance, it is not easy to know if a given *code word* is *valid* or not. That is why, instead of providing algorithms that directly enumerate every possible *valid code word*, we will use the useful *redundant* informations of a **Path** object to provide algorithms that enumerate every possible *valid* **Path**, among which we then find the *closed* ones.

   The different algorithms we provide here are themselves *classes*. The parent of all these enumeration algorithm classes is the class **PathExplorer**.

   We will see how this class is able to create a new **Path** object, and then sequentially modify it in place so as to make it sweep out all its possibles states.

6. As an example, we describe the most simple non-trivial child of **PathExplorer**, the **PathDecoder**, which is able to translate any *valid codeword* into a new *valid* **Path**.

   This class is used by the static **Path**.`fromCode(codeWord)` method. It concludes the proof of the existence of a bijection between the *valid codewords* and the *valid* **Path**s.

---

[1] Technically speaking, we proceed in such a way that our algorithm will never build nor modify a **Path** into an invalid state.

That is why there is no method `isValid()`, since there is supposedly no way to obtain an invalid **Path**.

Items 1) and 4) of this enumeration are treated in Section 6.1. Section 6.2 constitutes the item 2), Section 6.3 item 3), while items 5) and 6) are treated in Section 6.4. Finally, the Sections 6.5 and 6.6 contain the technical details about the implementation of the involved *classes*.

6.1 Encoding of simple closed geodesics

This section has been designed to be the first one of this chapter in order to introduce, one by one, the different *classes* that will appear in the `SurfacePath` module, parallel to the mathematical objects they are supposed to be models of.

The reader may consult the section 6.5, on page 137 while reading this one in order to to know exactly what data are effectively handled by its *classes* and how.

The goal is to *explain* how to *translate* a given simple closed geodesic $\gamma \subset S$, (we may think of γ as being drawn on S) into a certain (finite) number of *objects* with distinct *identities*.[2] We also explain what will be the *classes* of these objects, and what *members* they store.

Our arguments will also provide a proof that this can be done in a unique way[3].

Thus, assume that a Riemann surface $S = \mathbb{H}/\Gamma$ is given in the form of an object `surface` of type **Surface**, and assume that $\alpha_1, \alpha_2, \ldots, \alpha_{3g-3}$ are the $3g-3$ *main* geodesics of the marked surface and that `c1`, `c2`,... are the corresponding **Collar**s around these geodesics of `surface`.

Let us now consider a given *simple closed geodesic* $\gamma \subset S$.

The goal is then to give a representation of γ as an object of class **Path**.

The main geodesics α_i are themselves valid simple closed geodesics, and should therefore also be encodable as **Path** objects, but this is treated as a *special* case. This is why, for now, we suppose that $\gamma \notin \{\alpha_1, \ldots, \alpha_{3g-3}\}$. The case of encoding the *main* geodesics α_i is treated at the end in Definition 39.

Since by hypothesis the set of the α_i is a maximal set of pairwise disjoint closed geodesics, γ must intersect at least one of the α_i. Note that all these intersections are *transversal*, i.e. the geodesics form non-zero angles at the intersection points.

Let us denote by $q_1, q_2, \ldots, q_m \in S$ the $m \geq 1$ points of intersection of γ with all the *main* geodesics α_i. Since, by hypothesis γ is simple, these points are distinct.

Furthermore, we temporarily choose an arbitrary orientation of γ, and assume that $q_1, \ldots, q_m$ have been sorted such that γ is the union of m consecutive geodesic arcs

$$e_1 := \overrightarrow{q_1 q_2}, e_2 := \overrightarrow{q_2 q_3}, \ldots, e_m := \overrightarrow{q_m q_1}$$

as in Figure 6.1. We later describe a more intrinsic orientation to give to the geodesic.

Definition 35 (Depth)**.** *The above number m will be called the* depth *of the simple closed geodesic γ. By extension, the* **Path** *class provides the* `depth` *property.*

---

[2] Here, on a technical point of view, the address in the memory.

[3] Up to a reordering of the *identities*.

The *main* geodesics α_i are defined to have *depth* of 0.

Note that simple closed geodesics of *depth* 1 can (and will) only occur on *Q-pieces*, of the *Fenchel-Nielsen* graph (if there are any).

At this stage we have m different possibilities for the choice of q_1 and two choices for the orientation of γ.

The **Path** object that will be constructed to model γ depends on both of these choices. This makes $2m$ different **Path** objects that are all models of γ.

These $2m$ paths will be said to be *conjugated*[4].

The $2m$ **Path**s will be considered as "valid" objects and dealt with in the same way. As for now, we assume a choice of q_1 and an orientation has been made and continue describing the "encoding process" based on this choice.

We later describe a condition of *being canonical* a **Path** can provide or not (the truth value can be accessed by the **Path**.`isCanonical` method()). This condition is useful in the sense that each **Path** will be *conjugated* to a unique *canonical* **Path**. It's existence will allow us to see this "encoding procedure" as a *bijective application*[5] from the set of the (unoriented) simple closed geodesics to the set of the possible *canonical* **Path**s.

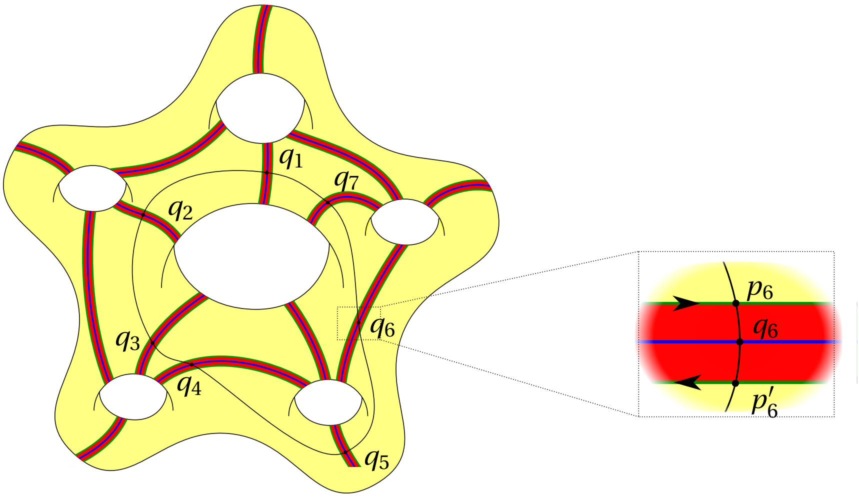


Figure 6.1: A closed geodesic of *depth* 7, on a **Surface** of genus 6.

Lemma 8. *Let $\epsilon > 0$ be sufficiently small.*

If a simple closed geodesic γ of depth m *intersects a* main geodesic α_j *of S at a point q_i, then either $\gamma = \alpha_j$ or there exists a subarc of γ that cuts either* border *of the ϵ*-collar *around α_j exactly*

---

[4] They may be seen as the cyclic permutations of the q_i and their reversions.

[5] In the mathematical sense.

once.

These two points will be referred to as path points *and denoted by* p_i *and* p'_i*, according to to the orientation chosen for* γ*, as shown in in Figure 6.1.*

Furthermore γ *contains exactly* $2m$ *distinct path points.*

Proof. If $\gamma \neq \alpha_j$, then the angle θ between γ and α_j at q_j is not zero. From the basic hyperbolic geometry formulas of page 23, it is easy to see that γ intersects then each of the *borders* exactly once, for any sufficiently small value of $\epsilon > 0$.

All these intersection points are distinct since γ is simple. □

In the model we develop, we will only deal with the *path points* $\{p_1, p'_1, p_2, p'_2, \dots, p_m, p'_m\}$ and forget about the q_i. Note that we think of ϵ as arbitrarily small. In the limit $\epsilon \to 0$ the points p_i, p'_i and q_i coincide, seen as geometrical points of S.

The class we use to describe the *path points* will be **PathPoint**. For a full description of the content of this class, see page 146.

We will soon say more about this class, but, before that, we introduce the smallest independent object that can be added to a **Path**: the **PathElem** class.

Topologically speaking, we can see γ as a connected loop (oriented) of $2m$ geodesic segments that go from one *path point* to another. These $2m$ segments come by pairs. The first geodesic segment is of the form $[p_i, p'_{i+1}]$ and lies entirely within an **YPce**. The second geodesic arc is of the form $[p'_{i+1}, p_{i+1}]$ and lies within a **Collar**.

Each of these pairs will now be represented by an object of the new class **PathElem** (see page 149).

Each **Path** of depth m will then contain a list of m **PathElem**s called `pathElements` [6].

Let us take `ei` to be the i-th *path element* of a **Path**. There are three *path points* that are related to `ei`. They correspond to p_i, p'_{i+1} and p_{i+1}.

These correspond to the following members of `ei` (and of the class **PathElem**): `ei.begPP` `ei.midPP` and `ei.endPP`.

Note that `ei.midPP` and `ei.endPP` are actually stored as data of `ei` while `begPP` is only a reference to the `endPP` of the previous **PathElem**.

We will also later consider **Path** objects that do not correspond to closed geodesics, but only to open piecewise geodesic curves. The *property* **Path**.`isClosed` returns `True` iff a current **Path**

---

[6] For an optimal memory management, `pathElements`, should be a *stack* (last in first out).
I.e. `std::stack<PathElem>` in C++

is closed[7].

That is why, for technical reasons, one more **PathPoint**, called **Path**.`begPP` is held directly by the **Path** class.

If `e1` is the first **PathElem** of a **Path** `path`, then `e1.begPP` is set to be a *reference* to `path.begPP`.

If `em` is the last **PathElem** of a **Path** `path`, then `em.endPP` will be referred to as the `path.endPP` *property*.

`path.begPP` and `path.endPP` are two **PathPoint**s with distinct *identities* (i.e. distinct addresses in memory) but representing the same point of the surface S. If we think of the *universal covering* $\mathbb{H}$, we may interpret these two **PathPoint**s as two different lifts of the same point, being the extremities of a geodesic arc $\tilde{\gamma} \subset \mathbb{H}$ of the same length as γ and covering it.

Each **PathPoint** will also hold the references `fPP` and `pPP`, that points respectively to the next and to the previous **PathPoint**s on γ, with respect to it's orientation (f for "future" and p for "past"). These two references can also be None if the given **PathPoint** lies at the extremity of the **Path**. i.e. for any **Path** `path`, `path.begPP.pPP` and `path.endPP.fPP` contain the `None` value.

Let `ei` be one of the **PathElem**s of a **Path**. `ei` can be seen as the geodesic arc $[q_i, q_{i+1}]$. Using the Lemma 3 of page 52, we can see that each **PathElem** is *locally homotopic* to one and only one **Road** of the **Surface**. That is why the **PathElem** class contains the `roadLayer` property (see page 149). Furthermore, since both the **Road** and the **PathElem** `ei` are oriented, each **PathPoint** can be associated to a *unique* **Slot** of the surface. That is why the **PathPoint** class contains the `slotLayer` *reference*.

Winding index

Finally, there is a last very important data that each **PathElem** has to store, it is called the *winding index* of the **PathElem**.

To define it we have to think in terms of *lifts* in the universal covering.

Figure 6.2 illustrates the objects to which the next definition refers.

---

[7] See page 144.

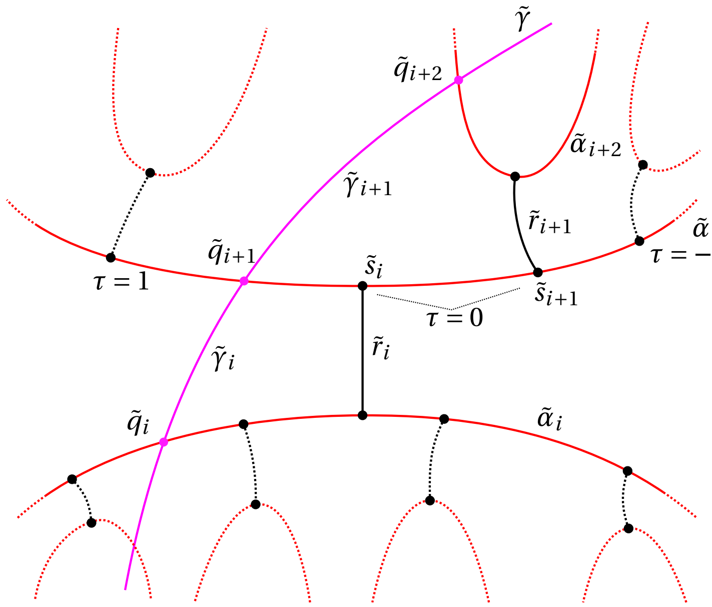


Figure 6.2: Two consecutive lifts of **PathElem**s in $\mathbb{H}$ and the *winding index* associated to the first one. In this example we have $\tau = 0$.

Let $[q_i, q_{i+1}] \subset S$ and $[q_{i+1}, q_{i+2}] \subset S$ be two consecutive *path elements* corresponding to the `ei1` and `ei2` **PathElem**.

Let $\tilde{\gamma}_i \subset \mathbb{H}$ be a *lift* of $[q_i, q_{i+1}]$ in the *universal covering* $\mathbb{H}$. The endpoints of $\tilde{\gamma}_i$ will be written $\tilde{q}_i, \tilde{q}_{i+1} \in \mathbb{H}$. The (unique) geodesic that passes through $\tilde{q}_i$ and $\tilde{q}_{i+1}$ will be written $\tilde{\gamma} \subset \mathbb{H}$.

There will then be a *unique* point $\tilde{q}_{i+2} \in \tilde{\gamma}$ such that the geodesic arc $\tilde{\gamma}_{i+1} := [\tilde{q}_{i+1}, \tilde{q}_{i+2}] \subset \mathbb{H}$ is a lift of $[q_{i+1}, q_{i+2}] \subset S$.

Let us denote by $\alpha_i, \alpha_{i+1}, \alpha_{i+2} \subset S$ the three main geodesics that respectively pass through q_i, q_{i+1} and q_{i+2}. (Ignoring the original enumeration of the main geodesics.)

Let $\tilde{\alpha}_i, \tilde{\alpha}_{i+1}, \tilde{\alpha}_{i+2} \subset \mathbb{H}$ be the (unique) lifts of α_i, α_{i+1} and α_{i+2} that pass through $\tilde{q}_i, \tilde{q}_{i+1}$ and $\tilde{q}_{i+2}$.

Let $r_i, r_{i+1} \subset S$ be the **Road**s associated to `ei1` and `ei2`. Due to Lemma 3, page 52, one can also conclude that there are uniques lifts $\tilde{r}_i, \tilde{r}_{i+1} \subset \mathbb{H}$ of these roads with the configuration as shown in Figure 6.2

Let now $\tilde{s}_i, \tilde{s}_{i+1} \subset \mathbb{H}$ be the two endpoints of $\tilde{r}_i$, respectively of $\tilde{r}_{i+1}$ that lie on α_{i+1}. Note that these points are lifts of the **Slot**s associated to `ei1.midPP` and `ei1.endPP`.

We have then two *lifts* of **Slot**s facing the same **Collar** and associated to each of the *facing borders* of this **Collar**. On page 54, in section 4.3, we described the concept of *winding index* which is an integer $\tau = \tau(\tilde{s}_i, \tilde{s}_{i+1})$ that "encodes" the distance from $\tilde{s}_i$ to $\tilde{s}_{i+1}$.

This value $\tau \in \mathbb{Z}$ will be called the *winding index* of the **PathElem** `ei1` and will be stored by the **PathElem**.`twistParameter` member.

6.2 Simple Path

In the previous section we presented how any simple closed geodesic $\gamma \subset S$ can be "encoded" into a (unique) **Path** object composed of m **PathElem**s. See also the exact class description 6.5.6 to see what exact data are.

In this section, we do the opposite.

We assume in the first place that we have a **Path** object (that may be considered "randomly generated") and we describe a special kind of piecewise geodesic curve on the **Surface** S that we can deduce from the **Path** data. We then call the curves obtained in such manner *ε-path.*

Then, we see under what conditions we can affirm that a given *ε-path* is simple (and/or closed) as a piecewise geodesic curve on S. These conditions are entirely "combinatorial" (as opposed to numerical).

Let S be a surface. Let $\varepsilon > 0$ be a sufficiently small real number. In order that the following arguments will be correct, ε must be strictly smaller than half the smallest non zero hyperbolic distance that exists between any two different **Slot**s of the given **Surface**. Note also that this implies that the conclusion of the Collar Lemma (Lemma 2, page 51) is verified for each of the *ε-collars* of the current surface.

Definition 36. *Given a* **Slot** *s of S, and an integer $q \in \mathbb{N}$, and an $\varepsilon > 0$ small enough as above, then it is possible to define q points on S in the following way.*

Take a chart *of the surface that covers an* ε-neighborhood *around s in S. For convenience, we may assume that the chart consistst in lifting a small neighbourhood of s to a neighbourhood of a lift $\tilde{s}$ in the universal covering $\mathbb{H}$. Furthermore, we choose the covering such that the lift of the frame associated to s is the* canonical frame *(the one whose origin is $i \in \mathbb{H}$ and whose axis is vertical). In other words, we will, from now on, use the expression "putting ourselves in the* frame *of s" to express the preceding idea. Also, we will not always make a distinction between the neighbourhood on S and its lift in $\mathbb{H}$.*

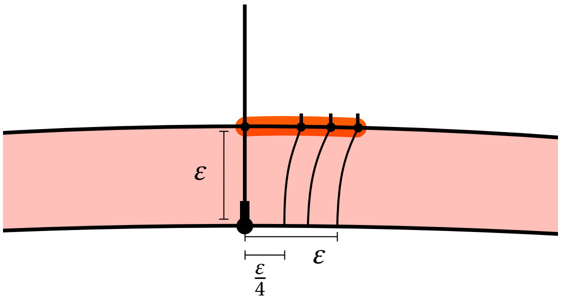


Figure 6.3: A free hand representation of what is meant by an ε-**SlotLayer**.
There are four **PathPoint**s in this example.

Let us take $V(t), H(t)$ to be respectively the vertical *and the* horizontal hyperbolic translations *of a distance t, as defined in section 3.4, page 25.*

We can take the following q hyperbolic transformations

$$H(\frac{0}{q}\varepsilon)\cdot V(\varepsilon),\quad H(\frac{1}{q}\varepsilon)\cdot V(\varepsilon),\quad H(\frac{2}{q}\varepsilon)\cdot V(\varepsilon),\ldots,H(\frac{q-1}{q}\varepsilon)\cdot V(\varepsilon)$$
$$:= \phi_s^0,\phi_s^1,\ldots,\phi_s^{q-1} \tag{6.2.1}$$

and interpret them as frames *in* $\mathbb{H}$. *The* origins *of these* frames *are here denoted*

$$\tilde{p}_s^0,\tilde{p}_s^1,\ldots,\tilde{p}_s^{q-1}\in\mathbb{H}.$$

They are points lying on the **Border** *of the* ε-collar *associated to the* **Slot** *s, as in Figure 6.3.*

The number q will be called the size *of the* slot layer.

We have taken ε so small that we can guarantee that two different ε-**SlotLayer**s cannot interfere. This is why we can also guarantee that their *projections* $p_s^0,\ldots,p_s^{q-1}\in S$ on the surface uniquely define q distinct points on the surface S itself.

Note that the ε we are using here serves to justify that we are building simple curves. In the calculations and implementations we will go to the limit $\varepsilon = 0$.

From now on we will consider that a *slot layer* is the data given by a **Slot** s and an ordered list of q points $\tilde{p}_s^0,\ldots,\tilde{p}_s^{q-1}$.

We shall call these points "the *q pathpoints* of an *ε-slot layer s*".

Each *slot layer s* will be represented as an object of the class **SlotLayer**, described on page 142, whereas the q points are represented by objects of the class **PathPoint**, described on page 146.

Given the set of all the **Slot**s of a **Surface**, it is possible to replace each of them by a **SlotLayer**. This collection of $24g-24$ **SlotLayer** is represented by the class **SurfaceLayer** described on 138.

We now describe how to construct an explicit set of curves on the given **Surface** S.

The name we give to this kind of curves on the surface is *ε-path* or simply *path,* since the actual value of ε does not matter (in the algorithms these curves are defined in exactly the same way but with $\varepsilon = 0$). The class that represent these curves will be **Path**, described on page 144.

Let us now now explain step by step how to "construct" an (ε-)*path*. At intervals we shall add restrictions or constraints our objects have to verify at any step.

All these constraints are grouped together on page 124.

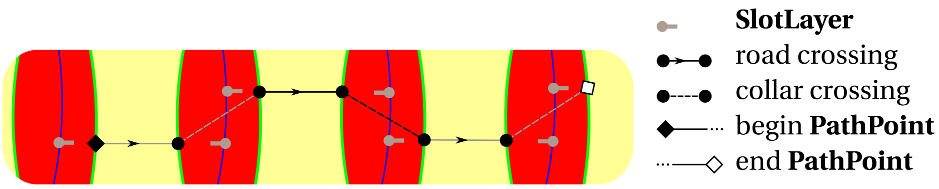


Figure 6.4: A free hand representation of what is meant by an ε-**Path**. To be accurate, a *lift* of it in $\mathbb{H}$.
There are three **PathElem**s in this example (gray then black then gray), which makes a total of 7 **PathPoint**s.

Definition 37 (Simple path)**.** *Let $\varepsilon > 0$ be be as described above.*

Let $\gamma \subset S$ be an open oriented piecewise geodesic curve, as on Figure 6.4.

Let $p_0, p_1, \ldots p_{n-1}$ be the n "discontinuous" points.

We will call p_0 the begin path point, *p_{n-1} the* end path point *and treat them as special cases. The other $n-2$* **PathPoint***s will be called* standard path points.

If γ respects all the following "axiomatic" conditions it will be considered as a valid ε-path.

Condition 1 (Structure)**.** *Each of the n points p_i along γ is precisely one of the ε*-path point *$p_i = p_s^u$ associated to one of the ε*-slot layers *of the* surface layer *for a certain u and a certain* **SlotLayer** *s.*

This condition justifies the fact that from now on we will also speak of the **PathPoint***s along a* **Path**.

Furthermore, the n **PathPoint***s are supposed to be* distinct.

A consequence of that is that the sum of the sizes[8] *of the different* **SlotLayer***s of the* **SurfaceLayer** *is precisely n.*

Using the "class vocabulary" described in the `SurfacePath` *6.5 Module, this constraint can be written (in* `Python` *) as follows.*

```
sum(slotLayer.size for slotLayer in surfaceLayer.slots) == path.numPathPoints
```

---

[8] The number of **PathPoint**s in a given **SlotLayer**

Let $P = (p_1, \dots, p_{n-1})$ be a **Path** and p_i its **PathPoint**s.

If p_{i-1}, p_i, p_{i+1} are three consecutive points of its **PathPoint**s we shall say that p_i is *standard*.

This condition leads us to introduce the following notations.

Notation 1. *Given a* **PathPoint** `pathPoint`*, there are (at most) four other* **PathPoint***s that are "logically" connected to* `pathPoint`.

- *Firstly, since the* **Path** *is an oriented curve, we can think of the* **PathPoint***s* $\dots, p_{i-1}, p_i, p_{i+1}, \dots$ *as coming "one after the other" in a time-like fashion. For this reason we shall use the words "future" and "past". Point* p_{i+1} *is the* future path point *of* p_i *while* p_{i-1} *is its* past path point *and can respectively be reached through the* **PathPoint**`.fPP` *and* **PathPoint**`.pPP` properties.

  The first **PathPoint** p_0 *(or* **Path**`.begPP`*) has no* past path point *i.e.* **Path**`.begPP.pPP is None`.
  Similarly, the last **PathPoint** p_{2n} *(or* **Path**`.endPP`*) has no* future path point *i.e.* **Path**`.begPP.fPP is None`.

- *Secondly, each* **PathPoint** *has at most two "neighbors" as an element of the list of the* **PathPoint***s of its* **SlotLayer***. We will use the notion of "left" and "right" to express this. "Going to the right" means to follow the orientation of the underlying* **Border** *(see 6.5.6).*

  These two **PathPoint***s can be reached through the* **PathPoint**`.lPP` *and* **PathPoint**`.rPP` *members*[9] *(which can be* `None` *for the first and the last element of* **SlotLayer**.*pathPoints list).*

Condition 2 (Proximity)**.** *For any* standard **PathPoint**`pathPoint` *of the* **Path***, exactly one of the two conditions holds*

- *The geodesic arc* r*, from* `pathPoint` *to* `pathPoint.fPP` *lies in an* ε*-neighborhood of a* **Road** *of the surface. To be more accurate, we state this condition in* `Python` *grammar*

  ```
  pathPoint.slot.roadFacingSlot is pathPoint.fPP.slot
  ```

  And

  The geodesic arc c*, from* `pathPoint` *to* `pathPoint.pPP` *lies in an* ε*-neighborhood of a* main geodesic *of the surface, within an* ε*-***Collar** *I.e.*

  ```
  pathPoint.slot.border.facingBorder is pathPoint.pPP.slot.border
  ```

---

[9] references

In this case, we say that the **PathPoint** *is "going outside" the* **Collar** *(see also the* **Path**.*isEnteringCollar property).*

- ***Or*** *The role of* `fPP` *and* `pPP` *are reversed in the following formal sense:*

```
    pathPoint.slot.roadFacingSlot is pathPoint.pPP.slot
    pathPoint.slot.border.facingBorder is pathPoint.fPP.slot.border
```

In this case, the **PathPoint** *is said to "enter into" the* **Collar**.

The first *and the* last **PathPoint***s* p_0 *(=:***Path**`.begPP` *) and* p_{2n} *(=:***Path**`.endPP` *) are supposed to be "going outside" a* **Collar** *in the sense that*

```
path.begPP.slot.roadFacingSlot is path.begPP.fPP.slot
path.endPP.slot.border.facingBorder is path.endPP.pPP.slot.border
```

The consequence of this second condition is that an open **Path** of *depth n* is a succession of *n* pairs of geodesic arcs such that the first one is a *road crossing* geodesic arc connected to a *collar crossing arc*, as in Figure 6.4.

Each of these pairs forms an object called **PathElem**, see 149.

Any **Path** respecting Conditions 1 and 2 is called *valid*.

It is now possible to "extend" this definition of a *valid open* **Path** to also speak of the *closed ones.*

Definition 38 (Closed path)**.** *If a* path *is such that*

```
path.endPP.rPP is path.begPP
```

then it is possible to merge the `begPP` *and* `endPP` **PathPoint***s into a single* standard *one.*

This is what we call a closed **Path***. See also the* **Path**`.isClosed` *property.*

The goal of our study is to focus on paths that, in addition are *simple* (as curves).

Proposition 8 (simple path)**.** *Let* `path` *be a* valid **Path** *(open or closed).*

Then, the corresponding ε`-path` *is* simple *(seen as a curve) if and only if the subsequent Conditions 3 and 5 are verified.*

Condition 3 (Parallel road crossing)**.** *This condition states that there are no intersections between the q parallel* road crossings *of a* **RoadLayer** *(see also page 140).*

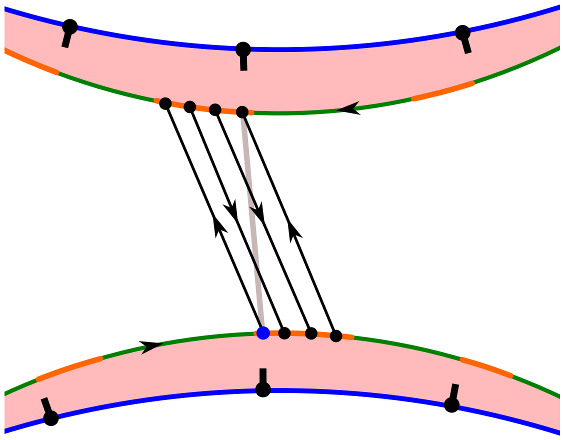


Figure 6.5: The *lift* of some *road crossings* along a **RoadLayer**.

As example Figure 6.5 shows the lifts of a **RoadLayer** *(gray)* used *four times by the* **Path***. The* **SlotLayer***s in which the* **PathPoint***s are are drawn in orange.*

In `Python`*, according to the* classes *detailed in Sections 6.5 and 6.6, this condition is equivalent to asking that for each* **PathPoint** pp *of the* **Path** *we have*

```
pp.roadFacingPP.lPP is pp.rPP.roadFacingPP
pp.roadFacingPP.rPP is pp.lPP.roadFacingPP
```

unless one of the subexpression is `None`[10]*, which is also considered valid and simple.*

Two different **Road**s of the same **YPce** may or may not intersect each other.

To know which **Road**s intersect a given one, see the **Road**`.corssingRoads()` method, page 62.

Let `roadA` and `roadB` be two **Road**s (two **RoadLayer**s to be accurate) and let c_A, c_B be two *road crossings* along `roadA`, respectively `roadB`.

One sees that c_A intersects c_B if and only if `roadA` intersects `roadB`.

We can then introduce the following condition.

Condition 4 (Incident roads)**.** *If some* road r *is* used *by a* road crossing, *then all the other* roads *intersecting* r *must be empty (not* used*).*

---

[10] For example, if `pp` is the `endPP` of the **Path**, then `pp.roadFacingPP` is `None`.

To allow the **PathExplorer** class to respect this condition, the **RoadLayer** class *stores* the number of times it has been *used* through the `timesUsed` attribute (see pages 62 and 140).

This allows to avoid entering a **Road** that *crosses* an already *used* road.

Any **Path** that respects these two Conditions 3 and 4 is guaranteed to have no self-intersection within the **YPce**s of the **Surface**.

The last Condition we need to guarantee the *simplicity* of a **Path** is then the following.

Condition 5 (CollarCrossing)**.** *The different* collar crossings *within a common* **Collar** *(***CollarLayer** *in fact) must not intersect each other.*

We discuss later, in Section 6.4, how to construct all the **Path**s that respect these constraints.

This Condition 5 allows the algorithm to limit the *winding indices* that have to be tried, when going across a non-empty **CollarLayer**.

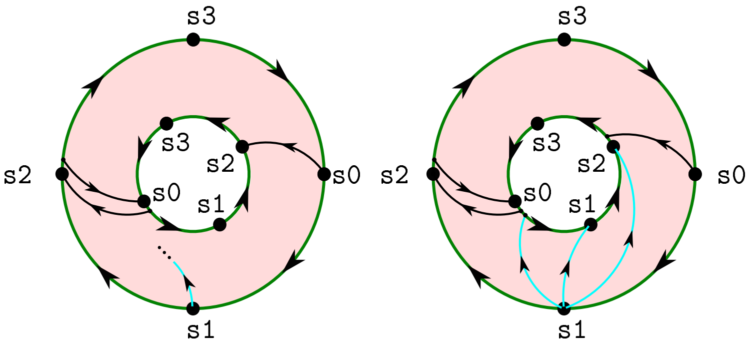


Figure 6.6: Crossing a non-empty **CollarLayer**.

The Figure 6.6, on the left, shows a case where a **Path** is extended by adding a new *collar crossing* (blue) across a nonempty **Collar**. The preexistent *collar crossings* are drawn in black.

The right part of the picture shows in blue the 3 possible *collar crossing* allowed by the Condition 5.

According to these condition, we can say that any **Path** that respects the Conditions 1-5 is *simple* as a curve.

We have seen earlier that every *closed* **Path** of *depth* m is *conjugated* to $2m$ other **Path**s, which corresponds in fact to the same curve but with a different choice of the *start point* and/or a change in the orientation.

For this reason we add a last condition we expect the **Path** objects to respect and call it *being canonical.* This condition is designed such that there will be exactly **one** *canonical* **Path** in each of the *equivalence classes* induced by the conjugation relation.

In order to formulate this condition, we recall that we may assume that we have a given *order relation* < over the different **Slot**s of a **Surface** (one can for example compare their address in memory).

Condition 6 (Canonical **Path**). *Let* `p` *be a* **Path***, let* `begPP := p.begPP` *be the first* **PathPoint** *of the* **Path***, and let* `s := begPP.slot` *be the* **Slot** *(***SlotLayer***) in which* `begPP` *lies.*

The **Path** `p` *is said to be* canonical *if and only if the two following conditions are verified.*

- `begPP` *lies over the* smallest **Slot**.

  I.e. if `pp` *is any* **PathPoint** *of the* **Path***, then*

  ```
  begPP.slot <= pp.slot
  ```

- `begPP` *is the leftmost* **PathPoint** *of its* **SlotLayer**. *In other terms,*

  ```
  begPP.lPP is None
  ```

  like the blue path point of Figure 6.5.

6.3 Path geometry

In the previous Section, we described how we can interpret the data of a **Path** object and the constraints it must satisfy in order to "encode" a valid simple and closed curve, that we called a *ε-path* $\gamma_\varepsilon \subset S$.

Since the **Surface** is hyperbolic, there will be only one geodesic $\gamma \subset S$ in the equivalence class of a given *ε-path*. Furthermore, since the *ε-path* is a simple curve, then γ must be simple as well, again because the surface is hyperbolic:

Proposition 9. *Let $\phi \subset S$ be a simple closed curve on an hyperbolic surface S, not homotopic to a point. Then, there exists a unique simple closed geodesic $\gamma \subset S$ that is homotopic to ϕ.*

Proof. This proposition is very well known and we do not need to provide an actual proof here. The existence of a geodesic γ, homotopic to ϕ comes from the Arzela-Ascoli theorem. The uniqueness is a consequence of the negative curvature [10]. The fact that this γ is simple can be shown by contradiction, knowing that S is hyperbolic and ϕ is simple [4, Theorem 1.6.6]. ☐

The goal is now to explain, given a **Path** object, how to obtain geometrical informations, such as the length of the unique simple closed geodesic γ it models.

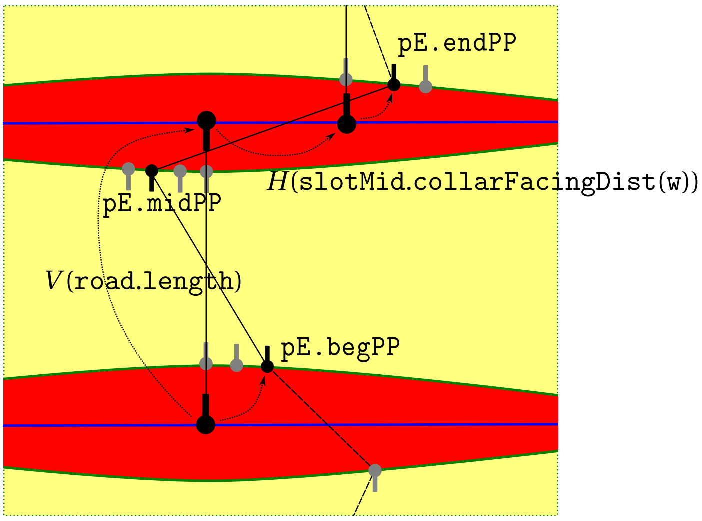


Figure 6.7: The *transformation* associated to a **PathElem**.

Let P be a simple closed **Path** of *depth* n.

As we have seen in the two previous Sections, P consists of n **PathElem**s, each of these **PathElem**s consists itself of a *road crossing* followed by a *collar crossing*.

Let `pE` be one of the **PathElem**s of the **Path** and let `wIdx = pE.windingIndex` be the *winding index* associated to the *collar crossing* of `pE`.

Let then `bPP = pE.begPP`, `mPP = pE.midPP` and `ePP = pE.endPP`, be respectively its *begin*, *middle* and *end* **PathPoint**, as shown in Figure 6.7, and explained in Section 6.5.7.

Finally, let `bS = bPP.slot`, `mS = mPP.slot` and `eS = ePP.slot` be the **SlotLayer**s in which these **PathPoint**s are.

Lemma 9. *According to the notations preceding the definition 14, page 25, of* $V(t), H(t), O \in M\mathbb{H}$ *and according to the definitions of the* relative frames *as in Section 5.1, page 69, the* relative position $\psi \in M\mathbb{H}$ *of the* frame `eS` *with respect to the* frame `bS` *is*

$$\psi = V(\texttt{bS.roadFacingDist}) \cdot H(\texttt{mS.collarFacingDist(wIdx)}).$$

This transformation ψ *will be called the* relative transformation *associated to the* **PathElem** `pE`.

The **PathElem**.`relativeTransformation` property *is designed so as to return this* ψ.

Proof. By construction, we know that the **Road** `bS.road` that connects `bS` to `mS` is the *vertical axis*, in the *frame* of `bS`. Its length is known and is stocked in the `bS.road.length=:bs.roadFacingDist` member. The relative position of `mS` with respect to `bS` is then $V(\texttt{bS.roadFacingDist}) \cdot O$.

According to the definition of the *winding index*, page 54, the *oriented distance* from `mS` to `eS` (along the main geodesic, with respect to the orientation of the **Border**`mS.border`) can be obtained trough the `mS.collarFacingDist(wIdx)` method.

The position of the `eS` in the frame of `mS` is then $O \cdot H(\texttt{mS.collarFacingDist(wIdx)})$.

Finally, we have then

$$\psi = V(\underbrace{\texttt{bS.roadFacingDist}}_{\in\mathbb{R}_+}) \cdot \underbrace{O \cdot O}_{\mathrm{Id}_{\mathbb{H}}} \cdot H(\underbrace{\texttt{mS.collarFacingDist(wIdx)}}_{\in\mathbb{R}}).$$

□

This Lemma leads us to the following Proposition.

Proposition 10. *Let* `path` *be a* simple closed **Path** *of* depth n *and let* `pE1`, `pE2`, ..., `pEn` *designate its* n **PathElem***s.*

Let `s0:=pE1.slot` *be the* initial **SlotLayer** *of the* **Path**.

Let $\psi_1 := \texttt{pE1.relativeTransformation}, \ldots, \psi_n := \texttt{pEn.relativeTransformation}$ *be the* n *transformations described by the preceding Lemma.*

Let $\gamma_\varepsilon \subset S$ *be the* ε*-path described by* `path` *and let* $\gamma \subset S$ *be the unique simple closed geodesic in the homotopy class of* γ_ε.

Let G *be the following Möbius transformation*

$$G = \psi_1 \cdot \psi_2 \cdot \ldots \cdot \psi_n \in M\mathbb{H}.$$

Then G *is an* hyperbolic transformation *and its* axis $\tilde{G} \subset \mathbb{H}$ *is a* lift *of* γ *expressed in the* frame *associated to the* **Slot** `s0`.

Furthermore the length *of* γ *can be deduced from the* trace *of* G*, using the* matrix model formula *3.2, page 21.*

Proof. Let `bPP1:=pE1.begPP`,…, `bPPn:=pEn.begPP` be the *n begin* **PathPoint**s and let `bS1 := bPP1.slot`,…, `bSn := bPPn.slot` be the **SlotLayer**s associated to these **PathPoint**s.

Similarly, let `ePP1:=pE1.endPP`,…, `ePPn:=pEn.endPP` be the *n end* **PathPoint**s, associated to the **Slot**s `eS1 := ePP1.slot`,…, `eSn := ePPn.slot`.

Since `path` is closed, `ePP1 = bPP2`, `ePP2 = bPP3`,… and `bPP1 = ePPn`.

Using recursively the Proposition 7, page 70, we can interpret the Möbius transformation G as the "relative position" of one of them. More precisely, if we lift `path` to a (piecewise geodesic) arc in $\mathbb{H}$ that has the same length, then G maps its initial point to the endpoint. If we lift, similarly, the closed geodesic γ in the homotopy class of `path` to a geodesic arc γ' in $\mathbb{H}$ that has the same length, then G also maps the initial point of γ' to the endpoint of γ'. Finally, γ' lies on the lift $\tilde{\gamma}$ of γ which is the axis of G. Therefore the displacement length of G equals the length of γ. This concludes the proof.

□

6.4 Path exploration

In Section 6.1, we described how any *simple closed geodesic* $\gamma \subset S$ can be encoded into a *valid simple closed* **Path** object.

Then, in Section 6.2, we precisely described what are the constraints a **Path** must respect to be *valid* and *simple*.

Finally, in Section 6.3 we explained how, given a *simple closed* **Path** `psiPath`, we can deduce both the length and the trace of a *simple closed geodesic* ψ. Furthermore, due to Proposition 5 we can deduce that `psiPath` would indeed be the result of the "encoding process" of Section 6.1, applied to ψ.

This shows that there is a *bijection* between the "abstract" set of the simple closed geodesics of S, sometimes written $\text{Spec}(S)$, and the set of all the possible *valid closed simple* **Path**s.

Obviously, both these sets are *infinite* (though still *countable*). This is why it would not make sense to try to enumerate "all" of the possible *valid closed simple* **Path**s.

Rather than that, we will try to produce several "exploration algorithms". Each of these algorithms will be specialized in spanning a different kind of finite subset of the form

$$\{\gamma \in \text{Spec}(S) \,|\, P(\gamma)\}. \tag{6.4.1}$$

Here P has to be understood as an "additional constraint" that can be *true* or *false* depending on γ. This constraint P is the *characteristic* constraint of the algorithm.

For example $P(\gamma) := (\text{length}(\gamma) < 15)$ is a constraint that would lead to find every simple closed geodesic whose length is less than 15.

In order to achieve this goal, let us introduce the notion of the *codeword* of a **Path**.

6.4.1 Codeword

The **Path** class is designed to provide a rich *programmatic interface* well suitable to a developer to "encode" any (or at least as much as possible) new constraint P he wishes into an actual *boolean function* that applies to the **Path** objects.

It is also designed to be well optimized on the memory-management strategy, but still, at any point of its lifetime it is a "big" object [11].

That is why we introduce the concept of *codeword*.

Definition 39. *Let* `path` *be a* simple **Path** *of depth* n.

---

[11] In fact it consumes very very little memory, but still much more than a *codeword*.

Then, we can build the following pair *(2*-uple*)*

$$(\texttt{path.begPP.slot}, [w_1, w_2, \dots, w_n]) \tag{6.4.2}$$

where `path.begPP.slot` *is (a reference to) the* start **SlotLayer** *of the* **Path**. *I.e. the* **SlotLayer** *in which the first* **PathPoint** *lies.*

The second element, the list $[w_1, w_2, \dots, w_n]$ *are the n* winding indexes *(see Section 6.1), associated to the n successive* collar crossings *of the* **Path**.

Such a pair is called the codeword *of the* **Path**.

See also the description of the trivial **Path**`.toCode()` *method which returns such a* codeword*, page 144.*

Finally, the encoding of the main geodesics $\alpha_1, \dots, \alpha_{3g-3}$ *themeselves is as follows:*

$$(\texttt{slot}, [\,]), \tag{6.4.3}$$

where `slot` *is the smallest* **Slot** *of the* **Collar** *around the corresponding* α_i.

We will also see at the end of this chapter that a *codeword* represents enough information to re-build the original **Path** through the **Path**`.fromCode(codeword)` method; or detect the fact that the *codeword* is invalid in the sense that it is the *codeword* of no valid simple **Path**.

Definition 40 (partial order relations)**.** *Let* `pathA` *and* `pathB` *be two* **Path** *objects of depth n and m respectively.*

Let $(s_A, [w_1^A, w_2^A, \dots, w_n^A])$ *and* $(s_B, [w_1^B, w_2^B, \dots, w_m^B])$ *be their respective* codewords.

Then, we can introduce a partial order relation *denoted by "<" over the* **Path** *class in the following way.*

`pathA < pathB` *if and only if*

- $s_A = s_B$
- *and* $m > n$ *and* $w_i^A = w_i^B$*, for all* $i \in \{1, \dots, n\}$.

Philosophically speaking `pathA < pathB` *means that* `pathA` *can somehow be* completed *to obtain* `pathB` *by appending* $m - n$ *additional* **PathElem***s at the end of the* future.

Note that the relation is only partial*: the negation of* `pathA < pathB` *does not imply anything special.*

We will also need a global order relation in our argumentation; we take then the following one.

Definition 41 (global order relations)**.** *Let* `pathA` *and* `pathB` *be two* **Path** *objects of depth n and m respectively.*

Let $(s_A, [w_1^A, w_2^A, \dots, w_n^A])$ *and* $(s_B, [w_1^B, w_2^B, \dots, w_m^B])$ *be their respective* codewords.

We also recall that we already have an order relation "$\leq$" over the the **Slot** *class.*

Then, we can introduce a partial order relation *denoted by "$\prec$" over the* **Path** *class in the following way.*

`pathA ⪯ pathB` *if and only if*

- $s_A \leq s_B$
- *and* $m \geq n$ *and* $([w_1^A, w_2^A, \dots, w_n^A] \leq [w_1^B, w_2^B, \dots, w_m^B]$ *in the sense of the traditional* lexicographic ordering *of two lists.*

6.4.2 Algorithms

We now have enough material to explain how our algorithms works.

Definition 42 (Filter)**.** *A* filter **Bool** `P(`**Path**`)` *is a* constraint *on a* **Path** `path`*, i.e. a boolean condition, that behaves well in the following sense*

$\forall$ **Path** `subPath`, $\big($`subPath < path and P(path)`$\big) \Rightarrow$ `P(subPath)`.

Furthermore the filter P *is said to be* finite *if the set* $\{\gamma \in Spec(S) \mid P(\gamma)\}$ *is finite.*

We can now represent the set of all the *valid simple* **Path**s as a finite set of *trees*, as in Figure 6.8. The *roots* of each of these trees are the $24g - 24$ possible *start slots*.

The *nodes* of these trees are the possible valid and simple states of the **Path**. In this context it is clearer to think of these nodes in terms of *codewords*.

On page 150, we define the `PathAlgorithms` *module*.

In this module, there is a class called **PathExplorer**.

This **PathExplorer** is the container of a **SurfaceLayer** that "contains" *one* **Path** object `path` [12].

Here is what the **PathExplorer** does.

It possesses a *method* called **PathExplorer**`.exploreStates()` and another, *purely virtual* (i.e.*abstract*) method called **PathExplorer**`.checkState()`.

---

[12] To be accurate it is in fact also designed to handle multiple **Path** at the same time that do not intersect each other. This is notably used for the **BersDecompositionFinder** class (7.5) , which is the *child* class of **PathExplorer** that find the *Bers decompositions* of a surface.

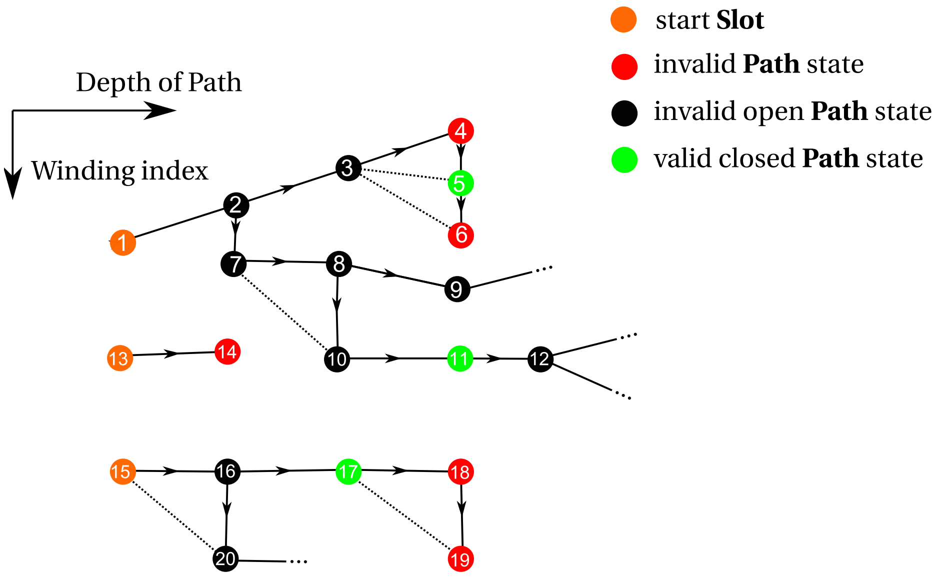


Figure 6.8: A (partial) view of the *tree* of the different possible valid simple **Path**s. The numbers represent the order in which a **PathExplorer** will explore the possible *states*.

When the method `exploreStates()` is called, it will start to "explore" the tree of the valid simple states by applying some in-place modifications to its **Path** object.

After each modification, the `checkState()` method is called.

Note also that the different simple states are tested in increasing order with respect to the $\prec$ relation defined in Definition 41.

The fact that `checkState()` is a *purely virtual method* means that the **PathExplorer** has no actual implementation of this method.

6.5 The `SurfacePath` module

This *module* collects the necessary material allowing us to enumerate representations of *simple closed geodesics*, satisfying certain restrictions (such as length), and their geometrical properties.

The classes of this *module* are the following:

SurfaceLayer: Given a **Surface**, it is possible to *construct* a **SurfaceLayer** that is a kind of *drawable copy* of it. Indeed, during the algorithms presented later, some *simple curves* over the **Surface**, represented by **Path** objects, will have to *store* some data about what places they went through and what they do in the corresponding **SurfaceLayer**.

RoadLayer: Class used (and stored) by the **SurfaceLayer** class. It is a *decoration* of one given **Road** of the **Surface**. It stores an integer `timesUsed`, which represents the number of **PathElem**s that come along the **Road**.

CollarLayer: Class used (and stored) by the **SurfaceLayer** class. It is a *decoration* of one given **Collar** of the **Surface**. It stores an integer `timesCrossed`, which represents the number of **PathElem**s that cross the **Collar**.

SlotLayer: This class is a decoration attached to each **Slot** of a **Surface** and stored by a **SurfaceLayer**. It acts as a list of **PathPoint** elements.

Path: This path represents a certain kind of curve over the surface, the set of the *valid*[13] *closed* **Path**s that can be constructed are in one to one correspondence with the simple closed geodesics of the surface. A path acts as a list of **PathElem**s. A list of **PathPoint**s can also be deduced.

PathPoint: A **PathPoint** is a point on a **Surface** that is related to a **SlotLayer**. A **Path** is a piecewise geodesic curve on the surface that goes from one **PathPoint** to another.

PathElem: This is the smallest element that can be appended to a **Path**.

---

[13] Defined later.

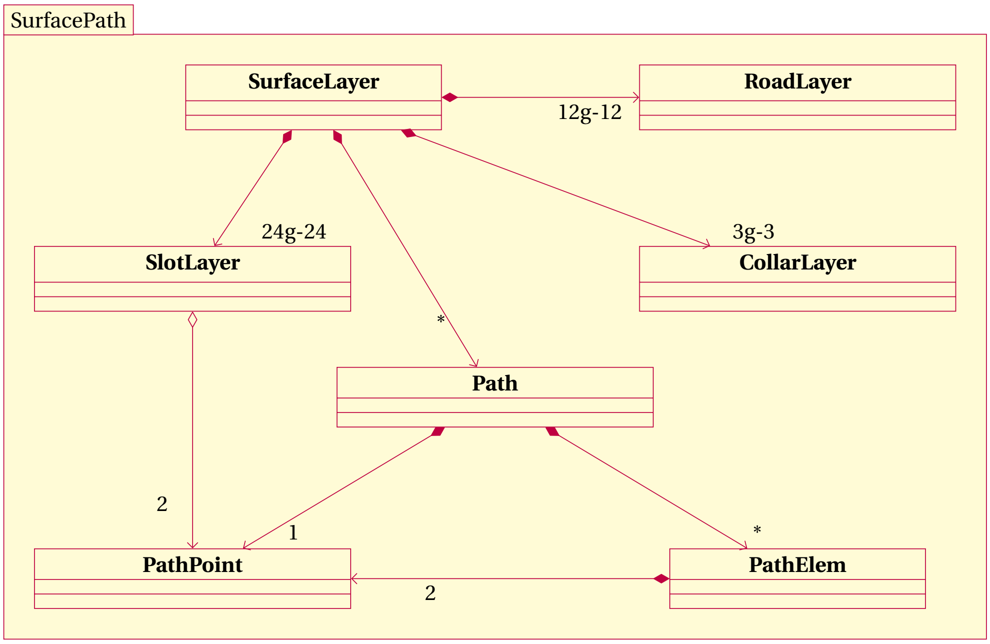


Figure 6.9: Classes of the `SurfacePath` module.

6.5.1 The `SurfaceLayer` class

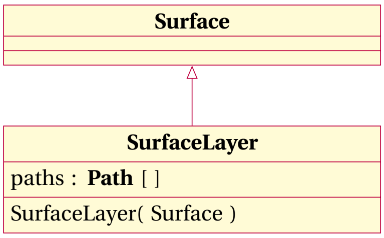


Figure 6.10: **SurfaceLayer** class diagram.

The ultimate goal that justifies these classes is to give an enumeration of the possible simple closed geodesics of a given surface.

The most trivial algorithm to do that, for example, enumerates each lexicographically ordered *codeword* and then checks whether it corresponds to a simple closed geodesic. But the second part is in fact very inefficient.

We rather have a more subtle and complete data structure that will evolve *in place* and take each valid state one after the other.

Metaphorically, these algorithms consist in taking a virtual pen and drawing some piecewise geodesic curve (in the form of a **Path** object) on a virtual surface.

At each step there are a finite number of combinatorial decisions that can be taken in order to build a new (deeper) state for the **Path** object. The fact that we are looking only for *simple* curves will be very helpful, since the decisions that have been taken at the previous steps drastically restrict the possible decisions of the current step.

That is why, from the *computational complexity* point of view, it can be useful to "store" a track of which parts of the surface have been crossed by the **Path** and how.

The class that is the "ultimate" holder of this data structure is the **SurfaceLayer** class.

The **SurfaceLayer** is a child of the **Surface** class and then it *inherits* all of its attributes.

The **Surface** class is thought of as *immutable*[14], whereas a **SurfaceLayer** is meant to be a "copy" of a given **Surface** that contains a list of **Path** objects. We will describe how the **SurfaceLayer** is affected by the **Path**s it contains.

The difference between a **Surface** and a **SurfaceLayer** (besides the **Path** list), is that a **SurfaceLayer** instead of containing **Road**s like a **Surface**, it contains a list of **RoadLayer** objects. Instead of containing **Collar**s it contains **CollarLayer**s and instead of containing **Slot**s, it contains **SlotLayer**s.

Each of these three classes are designed to hold cross-references with the **PathElem**s and the **PathPoint**s contained by the **Path**s of the **SurfaceLayer** that go "through" or "across" them.

The **SurfaceLayer** can be created by a *public constructor* that takes a **Surface** object and copies it.

The **SurfaceLayer** class is meant to be used by the **PathExplorer** sub-classes. In most of the algorithms, only one **Path** is involved at a time, for example looking for simple closed geodesics, systoles, and so on.

Some other algorithms involve two ore more mutually non intersecting **Path**s, for example those looking for Bers decompositions, or the algorithm for the graphical representation of the Birman-Series set.

See also:

Path class page 144, **RoadLayer** class page 140, **CollarLayer** class page 141.

---

[14] That means it is not supposed to be modified after it's instanciation.

6.5.2 The `RoadLayer` class

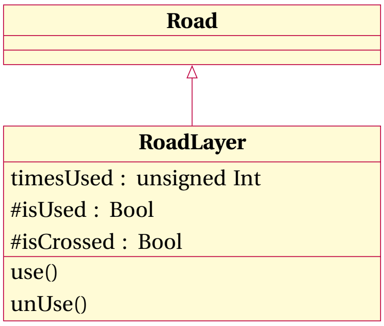


Figure 6.11: **RoadLayer** class diagram.

The **RoadLayer** class is a "specialized" version of the **Road** class (see page 62) that is supposed to interact with certain **Path** objects.

The objects of this class are created by the **SurfaceLayer**'s *constructor*. It *inherits* all the members of the **Road** class (it's *parent*) but, instead of having **Slot**s as *end points*, it has **SlotLayer**s.

The only actual member that is characteristic of the **RoadLayer** class, is the *timesUsed* (positive or null) integer.

We have seen that each **Path** can be decomposed into a certain number of **PathElem**s, that represent a number of geodesic arcs, within an **YPce**, that go from one of its bounding *main* geodesic to another (or the same one). We have seen in Lemma 3, page 52, that any of these arcs is *locally homotopic* to one single **Road** (or **RoadLayer**, to be accurate).

Each time a given **PathElem** happens to be *locally homotopic* to a given **RoadLayer**, we say that it *uses* this **RoadLayer**.

The *timesUsed* member is simply supposed to store the number **PathElem**s that are *locally homotopic* to the current **RoadLayer**, among all the **PathElem**s contained by all the **Path**s (usually only a single one is present) of the current **SurfaceLayer**.`path` list.

The `isUsed` property returns the truth value of `self.timesUsed > 0`.

Geometrically speaking, two **RoadLayer**s may intersect each other or not (they are said to be *crossing* if they do). Through the inheritance of the **Road** class, each **RoadLayer** has a method **RoadLayer**.`crossingRoads()` that returns a list[15] of the **RoadLayer**s that intersect this one.

The `isCrossed` property is designed to return `True` iff at least one of the *crossing road layers* `isUsed`.

---

[15] Of their *identities*, to be accurate.

Note that if there is a **RoadLayer** for which both `isUsed` and `isCrossed` are `True`, then this means that there are intersections among the **PathElem**s of this **SurfaceLayer**.

Finally, the `use()` and `unUse()` methods simply increment//decrement the `timesUsed` member by one unit.

See also:

PathElem class page 149, **SurfaceLayer** class page 138, **SlotLayer** class page 142.

6.5.3 The `CollarLayer` class

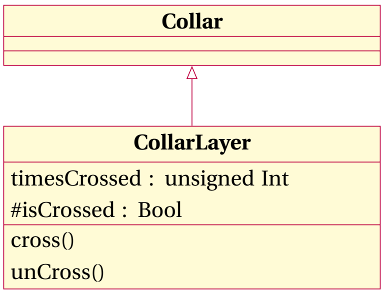


Figure 6.12: **CollarLayer** class diagram.

The **CollarLayer** class is a "specialized" version of the **Collar** class (see page 59) that is supposed to interact with **Path** objects.

The role that the **CollarLayer** plays with respect to the **Collar** class is very similar to the one of the **RoadLayer** with respect to **Road**. I.e. instead of containing 4 **Slot** objects, it contains 4 **SlotLayer**s.

Each **PathElem** of a **Path** is considered to go "through" a given **CollarLayer**. In that case the **PathElem** is said to *cross* the **CollarLayer**.

Besides the members of the **Collar** class, **CollarLayer** will then have the following members:

CollarLayer.`timesCrossed` is an unsigned integer that "stores" the exact number of **PathElem**s that go through this **CollarLayer**.

CollarLayer.`isCrossed` is a property that returns `True` iff `self.timesCrossed > 0`.

Finally, the `cross()` and `unCross()` methods simply increment//decrement the `timesCrossed` member by one unit.

See also:

PathElem class page 149, **SurfaceLayer** class page 138, **SlotLayer** class page 142.

6.5.4 The `SlotLayer` class

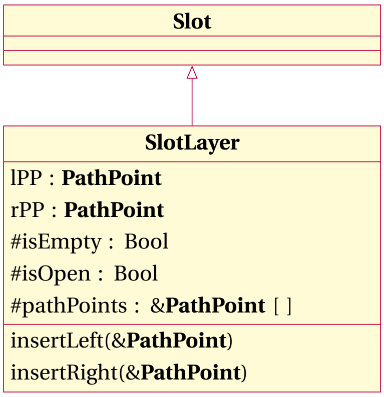


Figure 6.13: **SlotLayer** class diagram.

The **SlotLayer** is the "decorated" version of the **Slot** class (page 64) that is used in the context of a **SurfaceLayer**.

The **SlotLayer** *inherits* all the members of the **Slot** class, but instead of being the endpoint of a **Road**, it is the endpoint of a **RoadLayer**. Instead of being adjacent to a **Collar**, it is adjacent to a **CollarLayer**.

A **Path** of *depth* m can be seen as a list of m **PathElem**s, but it can also be seen as a list of $2m$ **PathPoint**s, respectively, $(2m+1)$ **PathPoint**s in the case of an "open" **Path** see page 144).

Each **PathPoint** is associated to a unique **SlotLayer** of the surface (see Section 6.1). Conversely, given a **SlotLayer** there is a finite list $[p_1, \ldots, p_n]$ consisting of the **PathPoint**s associated to this **SlotLayer**. The order of this list is very important. It is defined in Section 6.1.

This (ordered list) can be obtained through the `pathPoints` property of **SlotLayer**.

As far as memory is concerned, the only actual members of a **SlotLayer** object are two *nullable references* (or pointers) called `lPP` and `rPP` for "left" and "right" (according to their position in the list).

`lPP` points to the first **PathPoint** p_1 while `rPP` points to the last **PathPoint** p_n.

In the case where there is only one **PathPoint** associated to this **SlotLayer** we have `lPP == rPP`.

In the case where there is no **PathPoint** associated to this **SlotLayer** we have `lPP == rPP == None`. Furthermore, in that case and in that case only, the *property* **SlotLayer**.`isEmpty` will return `True`.

The `isOpen` *property* returns `True` iff the adjacent **RoadLayer** is not "crossed" by some path elements. It is equivalent to

```
return not self.road.isCrossed
```

and serves as a condition for the **PathExplorer** algorithms to know if this **SlotLayer** can be used to grow a **Path** without creating self-intersections.

When a **PathExplorer** "grows" a **Path** by appending to it a new **PathElem**, then two new **PathPoint**s are also created. They need to be "inserted" to the appropriate **SlotLayer**s. There are two ways to do this (both of them will be needed, depending on the situation).

The first way is through the **SlotLayer**.`insertLeft(pathPoint)` and **SlotLayer**.`insertRight(pathPoint)`, where `pathPoint` is a *virtual*[16] **PathPoint**.

The second way to "insert" a (virtual) **PathPoint** in a given **SlotLayer** is through the **PathPoint**.`insertLeft(pp)` and **PathPoint**.`insertRight(pp)` discribed on page 146.

The effects of these two methods are the following:
-They set a reference `self` (the current **SlotLayer**) in the `pathPoint.slot` member.
-They set appropriately the `lPP` and `rPP` members of the **SlotLayer**, and of the **PathPoint**s `pathPoint` and its new "neighbor".

See also:

PathPoint class page 149, **SurfaceLayer** class page 138.

---

[16] A *virtual* **PathPoint** is a technical "trick", it is a **PathPoint** that exists "data-wise" but has (still) not been cross-referenced with other objects.

6.5.5 The `Path` class

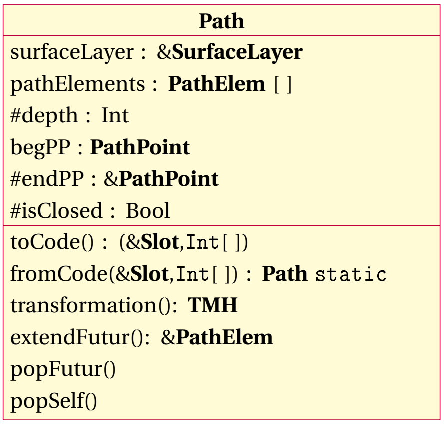


Figure 6.14: **Path** class diagram.

There are two kinds of mathematical objects a **Path** can represent. It either represents a closed curve (geodesic) on a surface, or it can represent an open segment of curve, as described in Section 6.1.

Here, we focus on the "technical" explanation on how we implemented it[17].

The *data* held by a **Path** object consists of
-a **PathPoint** named `begPP`,
-a list named `pathElements` of *m* **PathElem**s, where *m* is the *depth* of the path,
-a *reference* **Path**.`surfaceLayer` to the **SurfaceLayer** to which this **Path** belongs.

The *depth* can be obtained through the **Path**.`depth` property which returns the length of the `pathElements` list.

Attributes and properties

&SurfaceLayer `surfaceLayer`: A *reference* to the **SurfaceLayer** on which this **Path** lies.

PathElem `pathElements`: A *list* of the **PathElem**s of this **Path**.

In `C++` it is recommendable to use a `std::stack<PathElem>`.

`property`
Int `depth`: The *depth* of a **Path** is defined to be the number of its *path elements*, 0 for the *main* geodesics. Its value is deduced on the fly through the length of the `pathElements` list.

---

[17] There are certainly other (better) ways to do it.

`property`
Int `isClosed`**:** The *depth* of a **Path** is defined to be the number of its *path elements*, 0 for the *main* geodesics. Its value is deduced on the fly through the length of the `pathElements` list.

PathPoint `begPP`**:** The first **PathPoint** of a **Path**. Unlike the other **PathPoint**s which are held by the **PathElem**s, this one is held directly by the **Path**.

The reason for doing it this way is that we also have to handle non closed **Path**s. These have a total of $2m+1$ **PathPoint**s, where m is theirs *depth*. This makes 2 in each **PathElem** plus the `begPP`.

On the other hand, since the *main* geodesics of a surface need a *start slot* to be well defined the `begPP` of these **Path**s will store a reference to this **Slot**.

`property`
&PathPoint `endPP`**:** Returns a *reference* to the last **PathPoint** of the **Path**. This value depends on the fact whether the **Path** is *closed* or not. A possible `Python` implementation is the following:

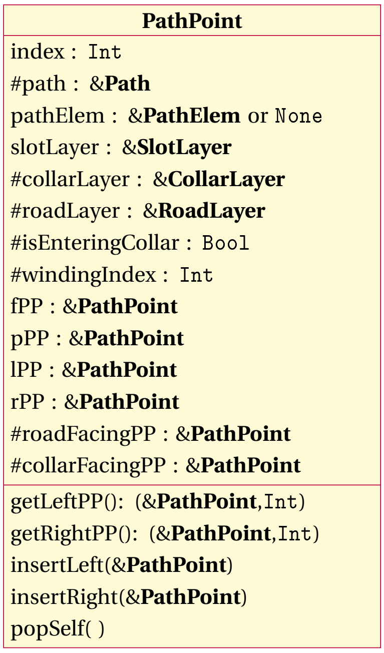
```python
if self.depth is 0: return self.begPP #For main geodesics.
lastPE = self.pathElements[-1] # The last Path Element of the Path.
if self.isClosed : return lastPE.midPP #For closed Path.
return lastPE.endPP #For non closed Path.
```

Methods

(&Slot, Int [ ]) `toCode()`**:** Each **Path** can return a "fingerprint" that is called its *code word.*

`static`
(&Slot, Int) `fromCode()`**:** Each **Path** can return a "fingerprint" that is callededits *code word.*

6.5.6 The `PathPoint` class



PathPoint

index : `Int`
#path : &**Path**
pathElem : &**PathElem** or `None`
slotLayer : &**SlotLayer**
#collarLayer : &**CollarLayer**
#roadLayer : &**RoadLayer**
#isEnteringCollar : `Bool`
#windingIndex : `Int`
fPP : &**PathPoint**
pPP : &**PathPoint**
lPP : &**PathPoint**
rPP : &**PathPoint**
#roadFacingPP : &**PathPoint**
#collarFacingPP : &**PathPoint**

getLeftPP(): (&**PathPoint**,`Int`)
getRightPP(): (&**PathPoint**,`Int`)
insertLeft(&**PathPoint**)
insertRight(&**PathPoint**)
popSelf( )

Figure 6.15: **PathPoint** class diagram.

As explained in the Sections 6.1 and 6.2, a **Path** may be seen as a piecewise geodesic curve going from one **PathPoint** to the next one. Furthermore, each **PathPoint** lies in the neighborhood of a **Slot**, within a **SlotLayer**.

According to that, in this section, we explain one by one the different *attributes* and *methods* of this *class*.

Attributes and properties

Int `index`**:** The *index* of the **PathPoint** with respect to its **Path**. The first **PathPoint** has index 0, the next one 1, etc...

`property`
&Path `path`**:** A reference to the **Path** to which this **PathPoint** belongs.

&PathElem `pathElem`**:** The *path element* to which this **PathPoint** belongs.

This reference is `None` for the *begin path point*, since this **PathPoint** belongs directly to the **Path** and not to a **PathElem**.

&SlotLayer `slotLayer`**:** The *slot layer* to which this **PathPoint** belongs.

`property`
&CollarLayer `collarLayer`**:** The **CollarLayer** that is adjacent to the current **PathPoint**.

`property`

&RoadLayer `roadLayer`**:** The **RoadLayer** that is adjacent to the current **PathPoint**.

`property`

Bool `isEnteringCollar`**:** Returns `True` if the next (with respect to the orientation of the **Path**) **PathPoint** is on the other side of the adjacent **CollarLayer**, and `False` otherwise.

Note also that this property is `True` if and only if the `index` of the **PathPoint** is odd.

`property`

Int `windingIndex`**:** The *winding index* associated to the adjacent *collar crossing*, or `None` for the first **PathPoint**, since it has no adjacent *collar crossing*.

See also Section 4.3 for the definition of the *winding index*.

&PathPoint `fPP`**:** The *future path point* i.e. the *next* **PathPoint** along the **Path**.

This reference is `None` for the *last* path point of the **Path**, **Path**`.endPP`.

&PathPoint `pPP`**:** The *past path point* i.e. the *previous* **PathPoint** along the **Path**.

This reference is `None` for the *first* path point of the **Path**, **Path**`.begPP`.

&PathPoint `lPP`**:** The *left path point* i.e. the **PathPoint** who lies **in the same SlotLayer** and is on the left of `self` by respect to the *frame* associated to the **Slot**.

In other terms, `self.lPP` comes *before* `self`, by respect to the orientation of the **Border** on which they are.

&PathPoint `rPP`**:** The *right path point* i.e. the **PathPoint** who lies **in the same SlotLayer** and is on the right of `self` by respect to the *frame* associated to the **Slot**.

In other terms, `self.rPP` comes *after* `self`, by respect to the orientation of the **Border** on which they are.

`property`

&PathPoint `roadFacingPP`**:** The **PathPoint** connected to `self` through the *road crossing*.

This *property* returns `self.pPP` if `self.isEnteringCollar` is `True` and `self.fPP` otherwise.

Note also that this means it returns `None` for **Path**`.endPP`.

`property`

&PathPoint `collarFacingPP`**:** The **PathPoint** connected to `self` through the *collar crossing*.

This *property* returns `self.fPP` if `self.isEnteringCollar` is `True` and `self.pPP` otherwise.

Note also that this means it returns `None` for **Path**`.begPP`.

Methods Note that the methods of this class are normally only meant to be used by the different methods of the **PathExplorer** class. Indeed, the **PathExplorer** is described as the only place where the **Path** objects can be created or modified.

In a language like `C++` , these methods could then have been declared a `private`, with the class **PathExplorer** declared as `friend`.

(&PathPoint,Int) `getLeftPP()`**:** This method is similar to `self.lPP`, but it is not limited to the **PathPoint**s lying in the same **SlotLayer**.

This method starts from the current **PathPoint** and follows the **Border** in the opposite direction to its orientation until it reaches a **PathPoint**.

The return value is a pair. The first element is a *reference* to the reached **PathPoint**. The second element of the return value is an integer (between 0 and 4 included) that states how much time it was necessary to "jump" from one **SlotLayer** to another.

Note also, as an example, that if `self` is the only **PathPoint** in the whole **CollarLayer**, the result of `self.getLeftPP()` will be `(self,4)`

(&PathPoint,Int) `getRightPP()`**:** This method does exactly the same thing as `getLeftPP()`, but this time it follows the **Border** along to its orientation

`insertLeft(&`**PathPoint**`pp)`**:** *Inserts* the (supposedly newly created) to the *left* of `self`.

Which means to settle the different attributes (`lPP`,`rPP`,`slotLayer`,etc.) of some objects (of class **PathPoint** and **SlotLayer**) accordingly.

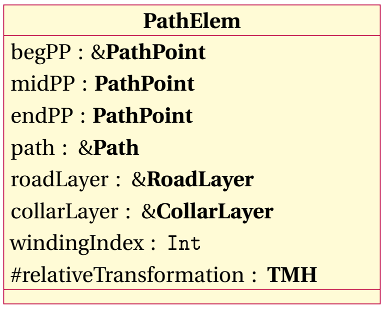
For example, after `self.insertLeft(foo)`, we must have `self.lPP == foo`, `foo.rPP == self` ,`self.slotLayer == foo.SlotLayer`, etc...

`insertRight(&`**PathPoint**`pp)`**:** Method similar to `insertRight(pp)`, but inserts on the *right*.

`popSelf()`**:** Deletes the current **PathPoint** from the current **SlotLayer** and sets the involved attributes in a new state.

The **PathPoint** `self` can be *inserted* at a new position after this method is executed.

6.5.7 The `PathElem` class



PathElem
begPP : &**PathPoint**
midPP : **PathPoint**
endPP : **PathPoint**
path : &**Path**
roadLayer : &**RoadLayer**
collarLayer : &**CollarLayer**
windingIndex : `Int`
#relativeTransformation : **TMH**

Figure 6.16: **PathElem** class diagram.

This class is the smallest element that can be added or substracted to a **Path**. The **PathElem** class is the actual *container* of the **PathPoint** objects [18].

For a detailed meaning of that a **PathElem** is we refer to Sections 6.1 and 6.2. Here, we onlypresent its attributes one by one.

Attributes and properties

&PathPoint `begPP`**:** The *first* **PathPoint** of the **PathElem**, at the start of the *road crossing*.

This is only a *reference*: it can either point towards the `endPP` of the previous **PathElem** or towards **Path**.`begPP` if the **Path** is *open* and if `self` is the first **PathElem** of the **Path**.

PathPoint `midPP`**:** The *middle* **PathPoint** of the **PathElem**, at the end of the *road crossing* and at the start of the *collar crossing*.

PathPoint `endPP`**:** The *end* **PathPoint** of the **PathElem**, at the end of the *collar crossing*.

Path `path`**:** A *reference* to the **Path** to which this **PathElem** belongs.

&RoadLayer `roadLayer`**:** A *reference* to the **RoadLayer** *used* by the *road crossing*.

&CollarLayer `collarLayer`**:** A *reference* to the **CollarLayer** *crossed* by the *collar crossing*.

Int `windingIndex`**:** The *winding index* associated to the *collar crossing* (see Section 4.3 page 54).

`property`
TMH `relativeTransformation`**:** The *Möbius Transformation of* $\mathbb{H}$ that represents the relative position of the *frame* associated to the **Slot** of **PathElem**.`endPP` by respect to the *frame* associated to the **Slot** of **PathElem**.`begPP`.

This transformation is denoted ψ in Lemma 9, page 131.

---

[18] With the exception of the **PathPointPath**.`begPP` who is directly hold by the **Path** object.

6.6 The `PathAlgorithms` module

The `PathAlgorithms` module collects the different *exploration algorithms* that we have. In Section 6.4, we described what we understand by the term *exploration algorithms*.

All the different variations of these *exploration algorithms* will be represented by some *classes* that derive from the common *base* class **PathExplorer**.

In this document we describe only the two most important cases, i.e. the (trivial) **PathDecoder** and the **SimpleGeodesicFinder** which can "enumerate" all the simple closed geodesics shorter than a given length.

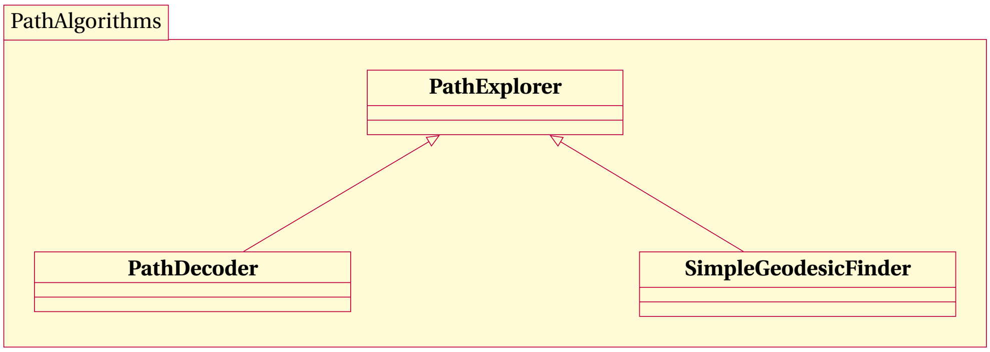


Figure 6.17: Classes of the `PathAlgorithms` module.

6.6.1 The `PathExplorer` class

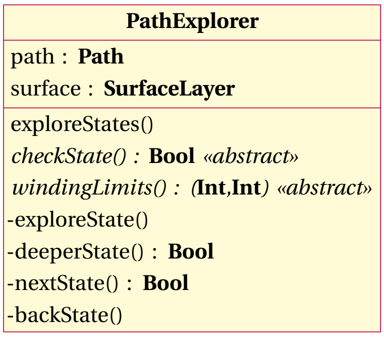


Figure 6.18: **PathExplorer** class diagram.

The **PathExplorer** is the *base class* of all the actual algorithms that involve testing one after the other all the possible simple **Path**s, with some *stop conditions*.

These stop conditions consist in two *abstract methods* which are **Bool** `checkstate()` and (**Int**,**Int**) `windingLimits()`. They **must** be implemented by the *children* of the **PathExplorer** class, which are the *actual* algorithms, like **PathDecoder** or **SimpleGeodesicFinder**.

In addition to these two public abstract methods, there is a last public method called `exploreStates()`. This is the method one has to call to *launch* the enumeration algorithm.

To explain what it does, it is easier to see what each methods of this class does one by one.

`exploreStates()`: Enumerates each possible (simple) **Path** states and launches the `checkstate()`, virtual and abstract method, for each reached state.

To achieve this goal, an empty **Path** is created with the *first* **Slot** of the **Surface** as start slot, this puts the **PathExplorer** in its initial *state*. Then, the *recurcive* (and *protected*) method `_exploreState()` is called.

When this is done, the method `checkstate()` has been called by `_exploreState()` for each *state* of the **Path** that starts with the current *start slot*.

The corresponding **SlotLayer** to the start slot is then "closed", which means that the different **Path** that will be enumerated in the next stages of this function will never be allowed to pass through it again. This condition may be seen as the *minimization condition*, to avoid counting twice the same geodesic, up to cyclic permutation.

Then we take the next smallest **Slot** as the start slot of the **Path** and do the same thing, until every **Slot** has been treated.

`abstract`
Bool `checkState()`: This method is called each time the **PathExplorer** is in a new *state*. This method is supposed do do the two following things:

- Check if the current `path` attribute is a solution to the current problem. (For example, “is the path *closed*? and does it correspond to a geodesic shorter than L?”, in the case of **SimpleGeodesicFinder**).

  If this is the case, add the corresponding *codeword* to a list of the solutions.
- Return `True` if there can possibly exist *bigger* **Path** than the current one, in the sense of the relation 40 of page 134 that is still a solution to our current algorithm.

  Return `False` if we can guarantee that there can be no solutions that "begins" with the current **Path**.

`abstract`
(Int,Int) `windingLimits()`: In the case where the algorithm has to pass over an *empty* **Collar**, there could theoretically be infinitely many possible *winding indices* to be chosen for the current **PathElem**.

This is why, in this circumstance, the **PathExplorer** calls this *abstract method* `windingLimits()`.

This method returns the minimal and the maximal *winding index* that needs to be taken into consideration.

See also:

The implementations of this method in **PathDecoder** and **SimpleGeodesicFinder**.

6.6.2 The `PathDecoder` class

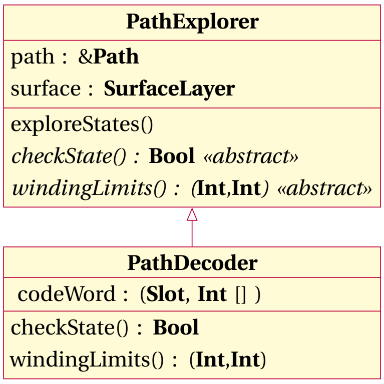

PathExplorer
path : &Path
surface : SurfaceLayer
exploreStates()
checkState() : Bool «abstract»
windingLimits() : (Int,Int) «abstract»
PathDecoder
codeWord : (Slot, Int [])
checkState() : Bool
windingLimits() : (Int,Int)

Figure 6.19: **PathDecoder** class diagram.

This class is the most *trivial* child of the **PathExplorer** class. The goal of the **PathDecoder** is simply to be able to *rebuild* a valid **Path** object, given a *codeword*.

The purposes of the existence of this class are, firstly, to let the **PathExplorer** class be the only place where the **Path** objects are created, modified (it is called the *factory design pattern*). This avoids to have redundancy in the code.

The second purpose is to provide a very simple example of an *actual* exploration algorithm. Though it is not really an *exploration*, in the sense that there is only one possible choice at each step.

This class overloads the two *abstract methods* `windingLimits()` and `checkState()` in the following way.

(Int,Int) **`windingLimits()`:** This method returns the minimal and maximal *winding index* to chose, when the *exploration algorithm* faces an empty collar. Since the *codeword* that we want to reproduce is given by the `self.codeWord` member, if m is the *depth* of the current **Path** `self.path`, then these minimal and maximal *winding indices* that we have to return have both the same value w_m, where w_m is the m-th *winding index* given in the codeword.

Bool`checkState()`**:** This method returns `False` if the last *winding index* of the current **Path** is the same as the one that is expected by the *codeword*. If the path is completed, the `path` is the desired solution. Finally, it returns `True` if the **Path** is correct, but incomplete.

6.6.3 The `SimpleGeodesicFinder` class

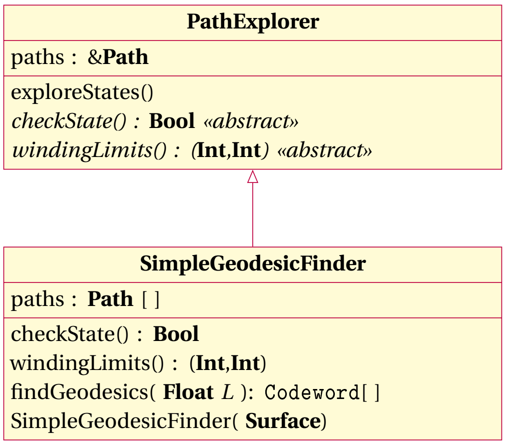

PathExplorer
paths : &Path
exploreStates()
checkState() : Bool «abstract»
windingLimits() : (Int,Int) «abstract»
SimpleGeodesicFinder
paths : Path [ ]
checkState() : Bool
windingLimits() : (Int,Int)
findGeodesics(Float L): Codeword[ ]
SimpleGeodesicFinder(Surface)

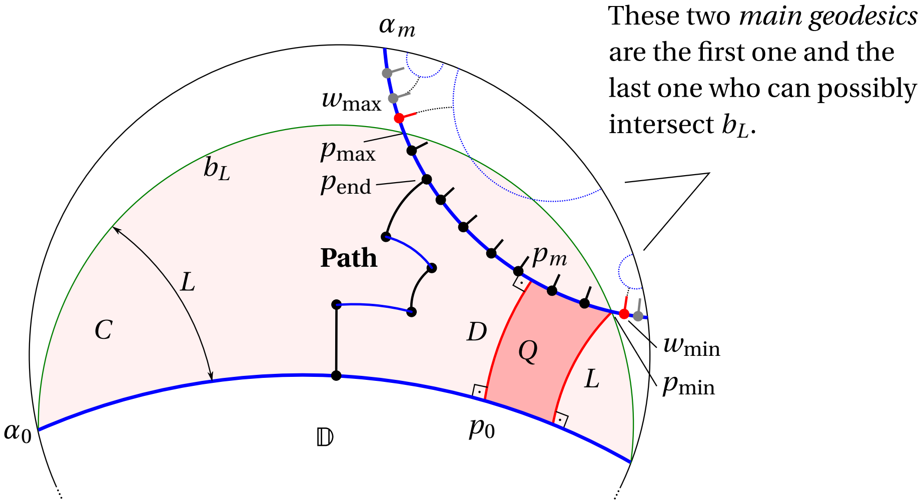
Figure 6.20: **SimpleGeodesicFinder** class diagram.

This class represents the algorithm that *enumerates* all the possible simple closed **Path**s up to a given length L.

This class is constructed with the **Surface** as argument. Once it is instantiated, the *user* can obtain a list of the codewords of *all*[19] the simple closed geodesics whose length is shorter than a length L, by calling the `findGeodesics(` **Float** L `)` method.

This method, will call the **PathExplorer**`.exploreStates()` method in its turn.

To achieve this goal, we need to explain on what kind of mathematical arguments the implementation of the two *overloaded methods* `checkState()` and `windingLimits()` relies.



These two *main geodesics* are the first one and the last one who can possibly intersect b_L.

Figure 6.21: Finding a minimal and a maximal *winding index*.

---

[19] Theoretically speaking, we can guarantee that the algorithm will miss none of them.

Figure 6.21 shows a representation of the situation at some *state* of the progression of the *exploration* algorithm.

The **Path** that the algorithm is acting on has a state of some depth m. We assume that the **Collar** we are searching to cross is empty, which forces us to decide, on the basis of the length L that has been given as the goal, what are respectively the *minimal* and the *maximal winding indices* $\alpha_{\min}, \alpha_{\max} \in \mathbb{Z}$ that have to be considered in the next stages of the algorithm.

In the figure, the first and the last *main geodesics* crossed by the **Path** are called α_0 and α_m respectively (blue).

A *half-collar* of *width* L around α_0 (light red) has also been drawn and is called C. The associated *border* (green) is called b_L.

The shortest geodesic arc from α_0 to α_m has been drawn in red. Let D denote its length and let p_0 and p_m be its endpoints.

There are two possibilities at this point:

1. If $D > L$, then we can guarantee that even if the **Path** can be completed into a closed one, its length is out of our range. If the **Path** is in such a state, the method `checkState()` will then return `False`.

   This has the effect of making the *exploration algorithm* go *back* one step and try some new **Path**s.

2. If $D \leq L$, then there are exactly 2 points of intersection [20] between α_m and b_L. We denote them by $p_{\min}$ and $p_{\max}$.

From now on we assume to be in the second case.

In Figure 6.21 several **Slot**s have been drawn on the *main geodesic* α_m.

Each of these **Slot**s can be considered to be *labeled* by the *winding index* between them and the current end **PathPoint**'s **Slot**.

These **Slot**s are the *candidates* for being the next **Slot** through which the **Path** will pass.

In Figure 6.21 the **Slot**s that lie on the geodesic arc $[p_{\min}, p_{\max}] = \alpha_m \cap C$ have been drawn in black. They have to be *tried* by the algorithm, since they possibly lead to simple closed geodesics shorter than L.

The two **Slot**s that are the two closest slots to C but lying **outside** of it are drawn in red. We call them $s_{\min}$ and $s_{\max}$, with $w_{\min}$ and $w_{\max}$ as their associated *winding indices*.

---

[20] Note that these two points may be considered equal in the limit case $D = L$.

These two **Slot**s also have to be *tested* by the exploration algorithm, since they can also lead to some solutions.

Finally, all the other **Slot**s (before $s_{\min}$ and after $s_{\max}$) have been drawn in gray, they cannot [21] lead to solutions shorter than L and then do not need to be tested.

The pair of *winding indices* $(w_{\min}, w_{\max}) \in \mathbb{Z}$ is then the result that we expect the method `windingLimits()` to return.

To calculate these two integers, the calculation we implemented is quite straightforward, but still requires several lines of code. We only present here an overview of how to actually obtain these two integers.

- Let p_{end} denote the position of the last **Slot** reached by the **Path**, on α_m, as in Figure 6.21.

  Using the results of Sections 3.2 and 6.3, one can quite easily compute the *oriented distance* from p_{end} to p_m as well as the distance D.

- Then, by symmetry, p_m is the midpoint of $p_{\min}$ and $p_{\max}$.

  Noting that the red shape Q of the figure is a *trirectangle*, we can use the formula (V) of Section 3.3.3, page 23, to compute the distance between $p_{\min}$ and p_m.

  One can deduce from this the oriented distances $d_{\min}$ and $d_{\max}$, from p_{end} to $p_{\min}$ and $p_{\max}$ respectively.

- Finally, from the definition of the *winding index* of Section 4.3, we can deduce how to compute $w_{\min}$ and $w_{\max}$ knowing $d_{\min}$ and $d_{\max}$.

  In other terms, if `endSlot` is the last **Slot** reached by the **Path**, $w_{\min}$ and $w_{\max}$ must be such that

$$
\begin{aligned}
\texttt{endSlot.collarFacingDist}(w_{\max}-1) &\leq d_{\max}\\
\texttt{endSlot.collarFacingDist}(w_{\max}) &> d_{\max}\\
\texttt{endSlot.collarFacingDist}(w_{\min}+1) &\geq d_{\min}\\
\texttt{endSlot.collarFacingDist}(w_{\min}) &< d_{\min}.
\end{aligned}
$$

---

[21] Indeed, taking one of these **Slot**s forces the **Path** to leave C forever. Even if we can later close it, it would be rejected because its length will be longer than L.

7 Applications

The goal of this chapter, which may be seen as a conclusion of this work, is to provide the reader with an overview of some "geometrical questions" that the concepts described in the previous chapters have allowed us to implement.

The precise description of the actual algorithms would be too long and technical to be of real interest and is thus not presented here.

These algorithms are intended to be delivered [1] in the form of a publicly accessible *program library* that will come along with its own detailed code-level documentation.

In the present chapter we will then stick to a purely informal description of these algorithms using the *spoken language*.

7.1 Finding a base of the fuchsian group

In Chapter 5, we described how to build a *canonical domain*, in the form of an *object* of class **CanonicalDomain** of any surface given through its Fenchel-Nielsen parameters.

Looking at the *identifications* associated to each of the consecutive *edges* of the **CanonicalDomain**, we obtain $4g$ Möbius transformations of the form $a, b, a^{-1}, b^{-1}, c, d, c^{-1}, d^{-1}, \dots$ and then the $2g$ elements $a, b, c, d, \dots$ form a *generating set* of the Fuchsian group associated to the surface.

7.2 Finding closed geodesics

A very important application of the `PathAlgorithms` module is to find all the closed simple geodesics of a surface that are shorter than a given length L.

To do that, we described in some details the class **SimpleGeodesicFinder** in Section 6.6.3.

---

[1] In the current of the year 2013 hopefully.

Starting from this one can as well implement very similar algorithms for example to find only the *systoles* of the *surface* or to find the N first elements of its *spectrum*.

7.3 Testing intersection between two simple closed geodesics

In Section 6.6.2, we described how a **Path** can be *reconstructed* from a *codeword*, by the class **PathDecoder**.

When two different *codewords* `w1,w2` are given (for example if they are two of the solutions of another algorithm), one can be interested in knowing whether the two corresponding **Path**s intersect each other.

To answer this question, one can implement a new class **IntersectionFinder** that works in a very similar way as the **PathDecoder**.

In Section 6.5, we briefly mentioned that the **SurfaceLayer** class is also conceived to be able to *contain* more than one **Path** at a time.

The **IntersectionFinder** then works in the following way.

- Firstly, the first *codeword* `w1` is decoded into a **Path** `p1`, exactly as a **PathDecoder** would do it.

  At the end of its creation, `p1` "stays alive" within the **SurfaceLayer**, which makes some **RoadLayer**s to be *used*, some **CollarLayer**s to be *crossed*, etc.
- Then, the **Path** `p2` is constructed within the **same SurfaceLayer** by a similar process.

  As a consequence, when the **Path** `p2` is being built, the **PathExplorer** algorithm will avoid creating intersections both with the **PathElem**s of `p1` and with the ones of `p2` itself.

If the construction of `p2` can be achieved (without breaking the Conditions of Section 6.2), this then means that the two **Path**s do not intersect each other.

On the other hand, if the **Path** `p2` cannot be reconstructed in the same **SurfaceLayer** as `p1`, we can conclude that they intersect each other.

Furthermore we can also use the geometrical results of Section 6.3 to find the *position* of these intersection(s).

7.4 *Bers* decompositions

Given a *Riemann surface* of *genus* g, there are (infinitely) many choices for the set of the $3g-3$ *main geodesics*. We call these sets *decompositions* of the surface.

Let $\alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_{3g-3}$ represent one of these *decompositions*, where the *main geodesics* have been sorted by increasing lengths. The length of the longest geodesic α_{3g-3} of a given *decomposition* is here called the "weight" of the decomposition.

Among all the possible *decompositions* of a given *surface*, the ones with the smallest *weight* are called *Bers decompositions* of the *surface*.

The *weight* of any *Bers decomposition* of a *surface* is called the *Bers constant* of the *surface*.

The algorithms developed here allow us to find a *Bers decomposition* of any given **Surface**.

To do that, one can do the following steps.

Firstly, since the **Surface** is given by *Fenchel-Nielsen parameters* and a *Fenchel-Nielsen graph* we already have a *decoponsition* of it, by construction. Let L be the *weight* of this decomposition.

Obviously the *Bers constant* of the surface is less than or equal to L.

Then, one can use the **SimpleGeodesicFinder** algorithm to get a list (ordered by increasing lengths) $c_1, \ldots, c_m$ of the *codewords* of the simple closed geodesics of the surface shorter than L.

Any *Bers decomposition* of the **Surface** must consists of $3g-3$ geodesics of this list.

Finding the good decomposition then simply becomes a combinatorial problem of finding the $3g-3$ smallest possible words, non intersecting each other among this finite list of m words.

Note also that we explained in the previous section how one can check if two given words intersect each other.

The class that provides this algorithm is named **BersFinder** and is part of our upcoming *library*.

7.5 *Birman-Series* set

Let S be a Riemann surface. Let $\mathrm{Spec}(S)$ denote the set of all the *simple* (*closed* or not) geodesics of S.

The set

$$B := \bigcup_{\gamma \subset \mathrm{Spec}(S)} \gamma \subset S$$

of all the points of the surface intersecting any simple geodesic is called the *Birman - Series* set. It has been defined in [2].

As an example of application of the concepts developed here, we have taken advantage of both the tools of the `HyperbolicDomain` module and of the `PathAlgorithms` module to build a class named **BirmanSeriesFinder**.

This class would be too long to be explained here, we simply explain what it is able to do.

Given a *goal* ϵ, a **Surface** S and a **Domain** $F \subset \mathbb{H}$ of S, the **BirmanSeriesFinder** is able to (numerically) *draw* a certain set $B_\epsilon \subset F$ that represents an approximation of the *Birman-Series set* in the sense of the following conditions.

- $B \subset B_\epsilon$ which means that no geodesic is "forgotten".
- If $p \in B_\epsilon$ then there exists $q \in B$ such that the hyperbolic distance $d_{\mathbb{H}}(p,q) < \epsilon$.

To conclude, Figure 7.1 shows an example of such a *Birman-Series* set.

Note also that this figure represents the "current state of the art" of our library and is not yet totally guaranteed to be correct.

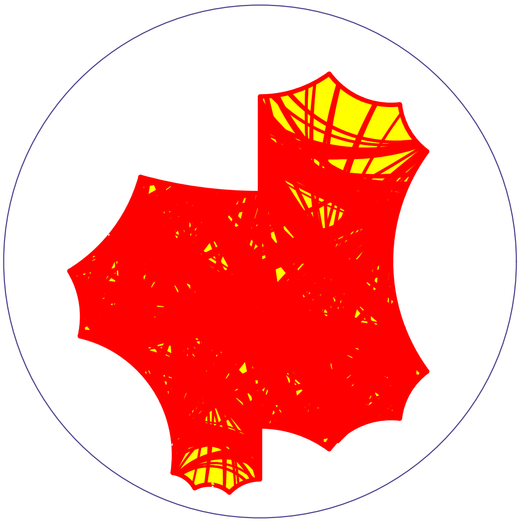


Figure 7.1: A *Birman-Series* set within a *basic domain*.

List of Figures

1.1 Transition map between two charts of a Riemann surface. 7

3.1 The hyperbolic plane with $\tilde{H}$ and $\tilde{M}$. 33

3.2 The hyperbolic plane with $\tilde{\phi}_i$ and $\tilde{\psi}_i$ in a case where $n = 3$. For better visibility the axis of $\tilde{M}$ has been drawn lower than its actual position. 36

3.3 An example of *class diagram*. 44

4.1 Examples of *Fenchel-Nielsen* graphs. 49

4.2 A surface of *genus* 2 with 3 collars and 2 *Y-pieces*, together with a topological representation of a *Y-piece* and a *collar*. 50

4.3 The **Road**s and the **Slot**s of a **YPce**. 52

4.4 The lift of a collar with lifts of **Slot**s facing each other. For ease of comprehension the *slots* have been drawn on their associated *borders*. Though they are in fact on the *main geodesic*. Here τ is the *Fenchel-Nielsen twist* oriented distance. w is the *winding index* related to some examples "connected" slots. 54

4.5 Aggregation diagram of the classes of module `SurfaceTopology`. 56

4.6 **Surface** class diagram. 57

4.7 **Collar** class diagram. 59

4.8 **Border** class diagram. 60

4.9 **YPce** class diagram. 61

4.10 **Road** class diagram. 62

4.11 **Slot** class diagram. 64

5.1 **HFrame** class diagram. 70

5.2 An 8-gon up to homeomorphism. 72

5.3 The *marked polygon* is degenerated. The *winding angle* is such that $\alpha_P(a) = 2\pi, \alpha_P(b) = 0, \alpha_P(c) = -2\pi$ 73

5.4 The angles in an "exagerated" view of a cusp. 74

5.5 A marked 9-gon up to homeomorphism, with v_3 a *cusp* vertex and v_6 a *peak* vertex. 74

5.6 Classes of the `HyperbolicDomain` module. 80

5.7 **DomVertex** class diagram. 80

5.8 The members of **DomVertex** class, graphically. 83

5.9 **DomEdge** class diagram. 84

5.10 The members of **DomEdge** class, graphically. 84

5.11 **DomIdentification** class diagram. 85

5.12 **Domain** class diagram. 85

5.13 **YDom** class diagram. 86

5.14 A fundamental domain of an Y-*piece* in $\mathbb{D}$. The two **HexDom**s are contrasted by a difference in shading. 87

5.15 **HexDom** class diagram. 89

5.16 **CollarDom** class diagram. 89

5.17 A fundamental domain **CollarDom** of a **Collar** in $\mathbb{D}$ is drawn in red. For the ease of exposition we gave a "width" to the **CollarDom**, but in reality it is totally flat and has zero area. In addition, two yellow **YDom**s have been "glued" to this **CollarDom** for better visualization. The two blue **DomEdge**s are pairwise identified. 90

5.18 **BasicDomain** class diagram. 90

5.19 **CanonicalDomain** class diagram. 91

5.20 Metaphor of our algorithm, creating the *fundamental domain* of a prism in $\mathbb{R}^2$. 93

5.21 A **BasicDomain** in $\mathbb{D}$ corresponding to the `graph3` in Figure 4.1, page 49. 95

5.22 Deleting a *cusp vertex*. 98

5.23 Deleting a *useless vertex*. 99

5.24 Getting rid of a *bad* vertex. In this picture, the colors indicate in which equivalence class each vertex lies. The red class is supposed to be distinct from the blue one. 101

5.25 Steps to obtain a *canonical* signature. 103

5.26 *Fundamental polygon inscribed in a fundamental domain* 107

5.27 Cutting and pasting a part of a *fundamental polygon* 109

6.1 A closed geodesic of *depth* 7, on a **Surface** of genus 6. 117

6.2 Two consecutive lifts of **PathElem**s in $\mathbb{H}$ and the *winding index* associated to the first one. In this example we have $\tau = 0$. 120

6.3 A free hand representation of what is meant by an ε-**SlotLayer**. There are four **PathPoint**s in this example. 122

6.4 A free hand representation of what is meant by an ε-**Path**. To be accurate, a *lift* of it in $\mathbb{H}$. There are three **PathElem**s in this example (gray then black then gray), which makes a total of 7 **PathPoint**s. 124

6.5 The *lift* of some *road crossings* along a **RoadLayer**. 127

6.6 Crossing a non-empty **CollarLayer**. 128

6.7 The *transformation* associated to a **PathElem**. 130

6.8 A (partial) view of the *tree* of the different possible valid simple **Path**s. The numbers represent the order in which a **PathExplorer** will explore the possible *states*. 136

6.9 Classes of the `SurfacePath` module. 138

6.10 **SurfaceLayer** class diagram. 138

6.11 **RoadLayer** class diagram. 140

6.12 **CollarLayer** class diagram. 141

6.13 **SlotLayer** class diagram. 142

6.14 **Path** class diagram. 144

6.15 **PathPoint** class diagram. 146

6.16 **PathElem** class diagram. 149

6.17 Classes of the `PathAlgorithms` module. 150

6.18 **PathExplorer** class diagram. 150

6.19 **PathDecoder** class diagram. 152

6.20 **SimpleGeodesicFinder** class diagram. 153

6.21 Finding a minimal and a maximal *winding index*. 153

7.1 A *Birman-Series* set within a *basic domain*. 160

Bibliography

[1] Aline Aigon-Dupuy, Peter Buser, and Klaus-Dieter Semmler. Hyperbolic geometry. In *Hyperbolic geometry and applications in quantum chaos and cosmology*, volume 397 of *London Math. Soc. Lecture Note Ser.*, pages 1–81. Cambridge Univ. Press, Cambridge, 2012.

[2] Joan S. Birman and Caroline Series. Geodesics with bounded intersection number on surfaces are sparsely distributed. *Topology*, 24(2):217–225, 1985.

[3] Peter Buser. Algorithms for simple closed geodesics. In *Geometry of Riemann surfaces*, volume 368 of *London Math. Soc. Lecture Note Ser.*, pages 38–87. Cambridge Univ. Press, Cambridge, 2010.

[4] Peter Buser. *Geometry and spectra of compact Riemann surfaces.* Modern Birkhäuser Classics. Birkhäuser Boston Inc., Boston, MA, 2010. Reprint of the 1992 edition.

[5] Martin Abadi; Luca Cardelli. *A Theory of Objects.* Springer Verlag, 1998.

[6] E.Balagurusamy. *Object-oriented programming with C++.* TaTa McGraw-Hill, 2007.

[7] H. M. Farkas and I. Kra. *Riemann surfaces*, volume 71 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, second edition, 1992.

[8] Marc Maintrot. *Finite element method on Riemann surfaces and applications to the Laplacian spectrum.* PhD thesis, École Polytechnique Fédérale de Lausanne, 2012.

[9] William S. Massey. *A basic course in algebraic topology*, volume 127 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, 1991.

[10] A. Preissmann. Quelques propriétés globales des espaces d riemann. *Comment. Math. Helv.*, 15:175–216, 1943.

[11] John Roe. Hyperbolic geometry lecture, www.math.psu.edu/roe/501-03/lecture22.pdf.

[12] Michael Spivak. *A comprehensive introduction to differential geometry. Vol. IV.* Publish or Perish Inc., Wilmington, Del., second edition, 1979.

[13] John Stillwell. *Classical topology and combinatorial group theory*, volume 72 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, second edition, 1993.

[14] Hsien-Chung Wang. Two theorems on metric spaces. *Pacific J. Math.*, 1:473–480, 1951.

Curriculum Vitae

Lausanne, 7 January 2013

Name	Racle
First name	Manuel
Date of birth	10.03.1983
Place of origin	La Neuveville, Switzerland
Marital status	single
Residence	Neu-Chemin 12 2533 Evilard Switzerland
Private phone	+41 32 322 45 34
Mobile	+41 76 223 09 73
E-mail	manuel.racle@a3.epfl.ch

Formation

2007 – now	PhD student at the Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland; chair of geometry (Prof. Jürg Peter Buser)
2001 - 2006	EPFL, Switzerland ; master in the physics section
2003 - 2004	University of Waterloo, Ontario, Canada 3rd year in physics as an exchange student
1998 - 2001	Gymnase français de Bienne, Switzerland
1993 - 1998	Ecole secondaire, Collège de la Suze, Bienne, Switzerland
1989 - 1993	Ecole primaire, Evilard, Switzerland

Certificates and degrees

June 2001	Certificat de maturité (high school certificate, with distinctive feature: bilingual, French-German), orientation physics and applied mathematics
April 2006	Master of Science MSc in physics section, EPFL (Ingénieur physicien dipl. EPF)

Professional experience

Apr. 2006-Oct. 2008	Rollomatic SA, Le Landeron, Switzerland Software engineer Research&Development / Mathematician in the R&D division. In this task, I actively participated in the mathematics' development of softwares (mathematical concepts and implementation). Specifically I developed: - a collision detection algorithm in a 3D simulation of the machines; - a trajectory optimisation between the machining operations; - the computation of “grinding” trajectories.
2007 – now	Teaching assistant duties for classes of geometry, chair of Geometry, Professor Jürg Peter Buser.

Conference Participations

July 2009	New Britain, Connecticut, USA
June 2010	Novosibirsk, Russia
November 2011	Monte Verita, Ascona, Switzerland

Knowledge and Competences

Language	
French	mother tongue
German	good oral and written knowledge, C1 level (bilingual high school certificate)
English	very good oral and written knowledge, C1 level (one year of studies in an English speaking university)
Informatics	Very good knowledge
Used softwares	Microsoft Office (Word, Excel, ...), LaTeX, Mathematica, Matlab, 3D Studio Max
Programming languages	C++, Python, C#
Internet	HTML, DHTML, Flash, MySQL databases, PHP servers, javascripts

Centres of interest

- Many sports (among others: climbing, skiing, snowboarding, sailing, biking); reading; board and computer games.
- Annual participation to the mathematics and logic contest organised by the *Fédération Française des Jeux Mathématiques*, with regular presence in national finals.

Diverse

- Prize of the best maturity work in my high school, for my work on solving the problems of the competition from the Swiss Federation of mathematics (June 2001).