



GAMES 301: 曲面参数化 作业报告

GAMES 301: Surface Parameterization Homework Report

作者: Mason Wu

组织: Infinite Heaven

版本: 0.02

Mason Wu: Simplicity ≠ Mediocrity.



ElegantLATEX Program

目录

第 1 章 Boundary First Flattening	1
1.1 摘要	1
1.2 引言	1
1.3 符号表示	1
1.4 预备知识	2
1.4.1 保角变换与共形映射 (Conformal Maps)	2
1.4.2 连续曲率与离散曲率	5
1.5 边界优先的平面参数化方法	8
1.5.1 泊松问题	8
1.5.2 Cherrier 公式	13
1.5.3 Poincaré-Steklov 算子	13
1.5.4 Hilbert 变换	14
1.5.5 插值	14
1.5.6 曲线积分	15
1.6 BFF 算法流程	16
1.7 实验结果	17
1.8 总结	20
附录 A 补充说明	21
A.1 算法流程	21
A.2 舒尔补 (Schur complement)	22
附录 B 代码修改说明	24
B.1 对原有文件的修改	24
B.2 新增文件	24

第 1 章 Boundary First Flattening

1.1 摘要

共形参数化 (Conformal Flattening) 方法可以将三维曲面映射到平面上，不对角度进行扭曲，是曲面参数化方法中常用的工具。目前的大部分曲面参数化方法要么缺少对映射区域的边界控制，要么采用昂贵的手段对映射结果进行优化。对于自由边界的参数化，作业 2 (Eigensystem [1]) 和作业 3 (LSCM [2]) 中的方法都可以做到。但是 Eigensystem 的参数化结果中，存在大量的局部大变形扭曲现象；而 LSCM 方法理论上需要固定两个边界点，导致平面参数化结果通常与直觉不符（如作业 3 报告中的图 1.4(l)，呈旋转对称的模型 Balls 的参数化结果并不具有旋转对称性）。本次报告，我们将实现 Rohan Sawhney 和 Keenan Crane [3] 提出的边界优先参数化 (Boundary First Flattening, BFF) 方法，实现从三维曲面到任意平面区域（自由边界、圆域和方域）的共形参数化映射。

- 渣代码越改越像 BFF，果然优秀的项目大道至简，辣鸡的项目千奇百怪。

1.2 引言

共形映射在近年来的研究中应用广泛，在许多领域的应用都体现了所谓保角的重要性。从计算的角度出发，共形映射对应着求解简单的线性方程组，提供了快速的算法，或者为困难的非线性任务提供了代价更小的初值。从性质出发，保角的性质保持了物理系统的机械属性或者成分属性。

共形映射十分灵活。圆盘类的三维曲面可以映射到平面上的任意形状，只要边界顶点可以在形状边界上任意滑动。BFF 是第一个通过稀疏矩阵分解提供对边界完全编辑能力的共形映射方法，包括：

- 自动进行平面参数化，具有最优的区域扭曲，
- 无缝圆锥参数化，
- 直接编辑边界长度或者角度，
- 单位圆盘上的正规化，
- 精确保持尖锐折角，
- 映射到给定形状。

BFF 可以直接替代广泛使用的 LSCM 等方案，同时还能对形状和面积扭曲等特征进行复杂的控制。

1.3 符号表示

表 1.1：文中所使用的符号及涵义

符号	涵义
\mathbb{C}	复数空间
x	标量，实数
\mathbf{X}	向量， $\mathbf{X} = [x_1, x_2, \dots, x_n]^T$
\mathbf{X}	矩阵， $X = [\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_m]$
V	集合
M	三维曲面
∂M	三维曲面的边界
Δ	Laplacian 算子

1.4 预备知识

我们部分采用文章 [3] 中的符号，斜体（如 A, x, \dots ）表示连续量，正体（如 A, x, \dots ）表示离散量。

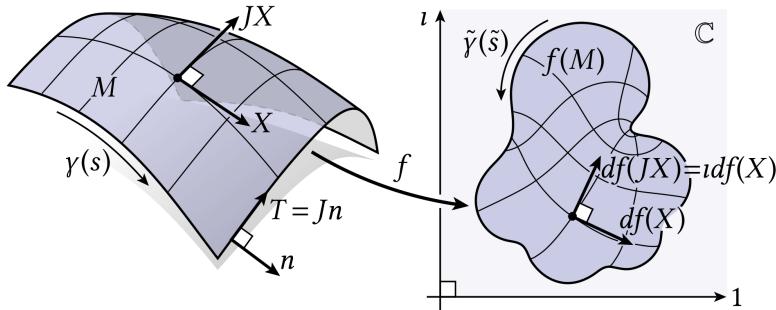


图 1.1: 共形映射平面参数化

基于共形映射的参数化方法的目的是，找到映射 $f : M \mapsto \mathbb{C}$ ，将类圆盘曲面 M （黎曼度量下）映射到复平面 \mathbb{C} 上，如图 1.1 所示。在曲面 M 上， X 为曲面上的切向量，映射 J 将切向量 X 以曲面的法向为轴顺时针旋转¹了 $\pi/4$ 。曲面边界 ∂M 是一条关于弧长参数 s 的闭合参数曲线 $\gamma(s)$ 。类似地，使用参数曲线 $\tilde{\gamma}(\tilde{s})$ 表示平面参数化映射的边界 $f(\partial M)$ ，也是 BFF 方法的控制对象。在曲面边界处的单位切向量 T 方向为 $d\gamma/ds$ ， $n = -JT$ 表示单位外法向。使用 K 表示曲面 M 的高斯曲率， κ 和 $\tilde{\kappa}$ 分别表示 γ 和 $\tilde{\gamma}$ 的测地曲率。

1.4.1 保角变换与共形映射 (Conformal Maps)

1.4.1.1 解析函数的特征

在复变函数论中，解析函数具有如下性质：

定义 1.1 (保域定理)

设 $w = f(z)$ 在区域 D 内解析且不恒为常数，则 D 的像 $G = f(D)$ 也是一个区域。



定理 1.1 说明区域 G 中任意两点 $w_1 = f(z_1)$ 和 $w_2 = f(z_2)$ 均可以用一条完全含于 G 的曲线 $\Gamma : w = f[z(t)]$, ($t_1 \leq t \leq t_2$) 联结起来。

推论 1.1

设 $w = f(z)$ 在区域 D 内单叶解析，则 D 的像 $G = f(D)$ 也是一个区域。



定义 1.2

设函数 $w = f(z)$ 在 z_0 解析，且 $f'(z_0) \neq 0$ ，则 $f(z)$ 在 z_0 的一个邻域内单叶解析。



解析变换 $w = f(z)$ 将 z_0 的一个充分小邻域变成了 $w_0 = f(z_0)$ 的一个曲边邻域。在区域 D 内，解析函数 $w = f(z)$ 有导数 $f'(z_0) \neq 0$ 。任意引一条经过 z_0 的定向光滑曲线

$$C : z = z(t), \quad (t_0 \leq t \leq t_1),$$

¹不确定理解是否正确，原文中说是将 X 逆时针旋转 $\pi/4$ 使得满足条件 $J^2X = J(JX) = -X$

$z_0 = z(t_0)$, 则必存在 $z'(t_0)$ 存在且 $z'(t_0) \neq 0$, C 在 z_0 处的切向量为 $z'(t_0)$, 倾角为 $\psi = \arg z'(t_0)$ 。经过变换, 曲线 C 的像 $\Gamma = f(C)$ 的参数方程为

$$\Gamma : w = f[z(t)], \quad (t_0 \leq t \leq t_1).$$

又 $w'(t_0) = f'(z_0)z'(t_0) \neq 0$, 所以 Γ 在 $w_0 = f(z_0)$ 也有切线, 切向量 $w'(t_0)$, 倾角为

$$\Psi = \arg w'(t_0) = \arg f'(z_0) + \arg z'(t_0) = \psi + \arg f'(z_0), \quad (1.1)$$

令 $f'(z_0) = Re^{i\alpha}$, 有 $|f'(z_0)| = R$, $\arg f'(z_0) = \alpha$, 所以

$$\Psi = \psi + \arg f'(z_0), \quad \lim_{\Delta z \rightarrow 0} \left| \frac{\Delta w}{\Delta z} \right| = R \neq 0.$$

上式说明 Γ 在点 $w_0 = f(z_0)$ 的切线正向, 可由曲线 C 在 z_0 的切线正向旋转 $\arg f'(z_0)$ 得到, $\arg f'(z_0)$ 仅与 z_0 有关, 与曲线 C 的选择无关, 称为变换 $w = f(z)$ 在 z_0 点处的旋转角, 如图 1.2 所示:

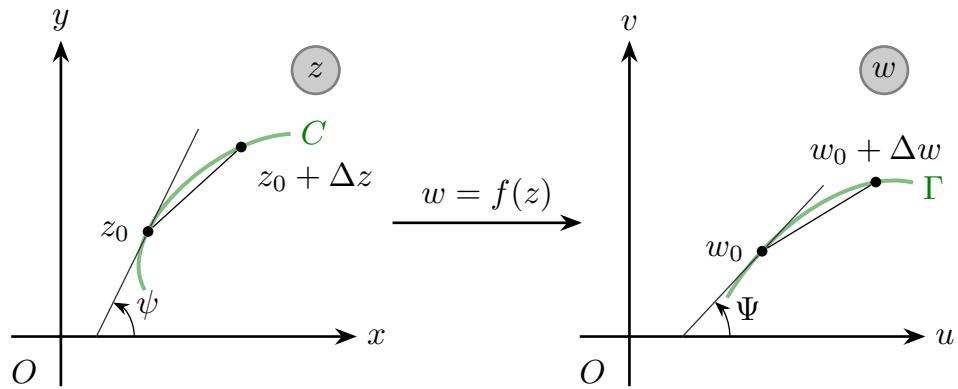


图 1.2: 解析函数的导数辐角

同时, 像的点的无穷小距离与原像点间的无穷小距离之比的极限是 $R = |f'(z_0)|$, 仅与 z_0 有关, 与经过 z_0 的曲线 C 的方向无关, 称之为变换 $w = f(z)$ 在点 z_0 处的伸缩率。解析函数的旋转角与 C 无关的性质称之为旋转角不变性, 伸缩率与 C 无关的性质称之为伸缩率不变性。

1.4.1.2 保角变换 [4]

我们称经过 z_0 的两条有向曲线 C_1 和 C_2 的切线方向所构成的角, 称为两曲线在该点的夹角。令 ψ_1 和 ψ_2 分别为曲线 C_1 和 C_2 在 z_0 处切线的倾角; Ψ_1 和 Ψ_2 分别为曲线 Γ_1 和 Γ_2 在点 $w_0 = f(z_0)$ 处切线的倾角, 如图 1.3 所示。

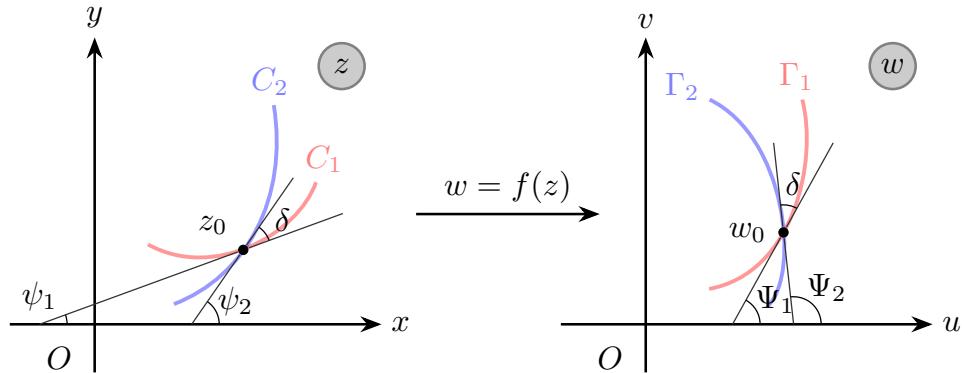


图 1.3: 解析函数的保角性

于是根据公式 1.1 可得

$$\begin{cases} \Psi_1 - \psi_1 = \alpha, \\ \Psi_2 - \psi_2 = \alpha, \end{cases}$$

所以有 $\Psi_1 - \psi_1 = \Psi_2 - \psi_2 = \alpha$, 即 $\Psi_1 - \Psi_2 = \psi_1 - \psi_2 = \delta$ 。角度包含方向（逆时针方向为正），保角性既保夹角的大小，又保持夹角的方向（图 1.3）。具有保角性的保角变换定义如下：

定义 1.3 (保角性与保角变换的定义)

若函数 $w = f(z)$ 在点 z_0 的邻域内有定义，且在点 z_0 处具有：

- (1) 伸缩率不变性，
- (2) 过 z_0 的任意两曲线的夹角在变换 $w = f(z)$ 下，既保持大小，又保持方向，

则称函数 $w = f(z)$ 在点 z_0 是保角的，或称 $w = f(z)$ 在点 z_0 处是保角变换。如果 $w = f(z)$ 在区域 D 内处处都是保角的，则称 $w = f(z)$ 在区域 D 内是保角的，或称 $w = f(z)$ 在区域 D 内是保角变换。



于是可知：

定理 1.1

如果 $w = f(z)$ 在区域 D 内解析，则它在导数不为零的点处是保角的。



注 定理说明了解析变换和保角映射的关系。从解析函数的

1.4.1.3 共形映射

复平面上 [4] 的共形映射的定义如下：

定义 1.4 (共形映射)

如果 $w = f(z)$ 在区域 D 内是单叶且保角的，则称此变换 $w = f(z)$ 在 D 内是共形的，也称它为 D 内的共形映射。



注 解析变换 $w = f(z)$ 在解析点 z_0 处有 $f'(z_0) \neq 0$ ，于是 $w = f(z)$ 在点 z_0 保角，因而在 z_0 的邻域内单叶保角，从而在 z_0 的邻域内共形（局部）；在区域 D 内 $w = f(z)$ （整体）共形，必然在 D 内处处（局部）共形，反之不一定。

类似地可以定义三维曲面的共形映射 [5]。从拓扑圆盘曲面 S 到单位圆盘 D 的共形映射记为 $f : S \rightarrow D$ 。令 C_1 和 C_2 是曲面 S 上任意的两条曲线，映射 f 将曲线分别映射到 $\Gamma_1 = f(C_1)$ 和 $\Gamma_2 = f(C_2)$ 。如果 C_1 和 C_2 的交角是 θ ，那么 Γ_1 和 Γ_2 的交角也是 θ ，那么 f 是共形的，具有保角性。

以图 1.1 为例，令 df 表示 f 的微分，决定了曲面上给定切向量 \mathbf{X} 如何映射到复平面上的切向量 $df(\mathbf{X})$ （坐标系下的 df 可以用雅可比矩阵表示）。对于所有的切向量 \mathbf{X} ，如果映射 f 满足

$$df(J\mathbf{X}) = i df(\mathbf{X}), \quad (1.2)$$

则映射 f 是全纯的²。意味着曲面上的 $1/4$ 周转动对应于复平面上的 $1/4$ 周转动。

如果 df 是非退化的（例如将非零向量映射到非零向量），那么 f 是共形的，缩放因子为 $e^u = |df(\mathbf{X})|/|\mathbf{X}|$ ，共形因子描述了每个点处的长度变化，而与 \mathbf{X} 的方向无关。函数 $u : M \rightarrow \mathbb{R}$ 叫做对数

²复解析函数是全纯函数，满足柯西—黎曼方程

共形因子。共形因子的知识超出了我的知识范围，可能在附录会讨论吧。

共形映射可以表示为一对共轭调和函数。满足 Laplace 方程 $\Delta a = 0$ 的实函数 $a : M \rightarrow \mathbb{R}$ 是调和的，其中 Δ 为域 M 上的 Laplacian (Laplacian-Beltrami) 算子：

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}.$$

那么由一对坐标系下的函数 $a, b : M \rightarrow \mathbb{R}$ 组成的全纯映射 $f = a + ib$, 满足：

$$J\nabla a = \nabla b, \quad (1.3)$$

即这对函数的梯度 ∇ 是正交的，且梯度大小相等。因为梯度场的 $1/4$ 周旋转是无散的，所以有

$$\Delta a = \nabla \cdot \nabla a = -\nabla \cdot (J\nabla b) = 0,$$

类似地，有 $\Delta b = 0$ ，即全纯函数的一对实成分都是调和的。

公式 1.3 说明了对于共形映射目标区域的形状有限制，象征区域边界的两个任意函数并不一定能扩展到区域内部的一对满足共轭调和关系的函数。然而，根据黎曼映射定理：

定理 1.2 (黎曼映射定理)

单一边界的简单联通曲面，如拓扑圆盘，可以共形地映射到单位圆盘上。



因此单位圆盘 $U \subset \mathbb{C}$ 可以通过一个黎曼映射到任意形状。

上述两个事实并不矛盾，尽管一个是到任意区域的满射 ($f(M) = U$)，另一个是任意“钉”在每个边界点的固定位置处 ($f|_{\partial M} = \tilde{\gamma}$) 并保持共形。所以 BFF 算法想要维持两个几何量：长度或沿边界的曲率密度，但是不能同时做到。

1.4.2 连续曲率与离散曲率

根据微分几何的相关知识，曲面 $S : \mathbf{r} = \mathbf{r}(u, v)$ 的第一、二基本形式分别为

$$\begin{cases} I = E du^2 + 2F du dv + G dv^2, \\ II = L du^2 + 2M du dv + N dv^2, \end{cases}$$

其中第一类基本量 E 、 F 和 G 以及第二类基本量 L 、 M 和 N 可以由 u -曲线和 v -曲线对应的切向量 \mathbf{r}_u 、 \mathbf{r}_v 以及曲面外法向 \mathbf{n} 表示：

$$\begin{cases} \mathbf{n} = \frac{\mathbf{r}_u \times \mathbf{r}_v}{|\mathbf{r}_u \times \mathbf{r}_v|}, \\ E = \mathbf{r}_u \cdot \mathbf{r}_v, \quad F = \mathbf{r}_u \cdot \mathbf{r}_v, \quad G = \mathbf{r}_v \cdot \mathbf{r}_v, \\ L = \mathbf{r}_{uu} \cdot \mathbf{n}, \quad M = \mathbf{r}_{uv} \cdot \mathbf{n}, \quad N = \mathbf{r}_{vv} \cdot \mathbf{n}. \end{cases}$$

曲面在给定点沿任一方向的法曲率 k_n 为：

$$k_n = \begin{cases} +k_0, & \text{法截线向 } \mathbf{n} \text{ 的正侧弯曲,} \\ -k_0, & \text{法截线向 } \mathbf{n} \text{ 的反侧弯曲.} \end{cases}$$

其中 k_0 表示方向 $(d) = du : dv$ 所确定的法截线 C_0 在 P 点的曲率，所以有

$$k_n = \frac{II}{I}. \quad (*)$$

此外，对于曲面上的一条曲线，如果它的每一点处的测地曲率为零，则称为测地线。曲面上的直线一定是测地线。

1.4.2.1 高斯曲率与测地曲率

对于曲面上任意一点的任意方向 $(d) = du : dv$, 法曲率公式为

$$k_n = \frac{\text{II}}{\text{I}} = \frac{L du^2 + M du dv + N dv^2}{E du^2 + F du dv + G dv^2},$$

沿 u —直线 ($dv = 0$) 的方向对应的主曲率是 $k_1 = L/E$, 而 v —直线 ($du = 0$) 方向对应的主曲率是 $k_2 = N/G$.

定义 1.5 (高斯曲率)

曲面 $S : \mathbf{r} = \mathbf{r}(u, v)$ 上任意一点的高斯曲率为 $K = k_1 k_2 = \frac{LN - M^2}{EG - F^2}$.



对于曲面 $S : \mathbf{r} = \mathbf{r}(u, v)$ 上的曲线 $C : u^\alpha = u^\alpha(s)$, $\alpha = 1, 2$, 由测地曲率定义如下:

定义 1.6 (测地曲率)

曲线 C 在点 P 的曲率向量 $\ddot{\mathbf{r}} = k\beta$ 在 $\epsilon = \mathbf{n} \times \alpha$ 上的投影 (在 S 上 P 点的切平面 Π 上的投影)

$$\kappa_g = \ddot{\mathbf{r}} \cdot \epsilon = k\beta \cdot \epsilon,$$

称为曲线 C 在 P 点的测地曲率.



1.4.2.2 离散曲率

为了导出对于离散的三维曲面 M 所对应的高斯曲率与测地曲率的表达式, 我们考虑高斯—波涅 (Gauss-Bonnet) 公式:

定理 1.3 (高斯—波涅 (Gauss-Bonnet) 公式)

假设单连通曲面域 C 是由曲面 S 上一个由 k 条光滑曲线段围成的曲线多边形区域, 多边形为 C 的边缘 ∂C 。设曲面 S 的高斯曲率和测地曲率分别为 K 和 κ_g , 曲面面积微元 dA , 弧长微元 ds , 则有高斯—波涅公式成立:

$$\iint_C K dA + \oint_{\partial C} \kappa_g ds + \sum_{i=1}^k (\pi - \alpha_i) = 2\pi, \quad (1.4)$$

其中 α_i 是 ∂C 的第 i 个内角的角度, $\pi - \alpha_i$ 是对应外角的角度.



对于离散化的三角网格曲面, 考虑围绕顶点 V_i 的单元 C_i , 示意图如右图 1.4 所示, 对应三维曲面上的情况如图 1.5 所示。 C_i 表示沿顶点的 Voronoi 元胞, 单元边界边垂直平分三角形边。令 γ_j 表示 C_i 的外角, Gauss-Bonnet 公式化简为

$$\iint_{C_i} K dA + \sum_{V_j \in N(V_i)} \gamma_j = 2\pi, \quad (1.5)$$

其中 $N(V_i)$ 是顶点 V_i 的 1—邻域的邻居数量, $|N(V_i)|$ 是与 V_i 相邻的顶点数/三角面片的数量。

注 注意到公式 1.4 化简为公式 1.5 的过程中, 测地曲率项为零。实际上, 在每个与 V_i 相邻的三角面内部有测地曲率 $\kappa_g = 0$ 。在三角形边上的测地曲率 κ_g 也为零。(观察图 1.5, 三角形边是曲面上的直线, 而曲面上的直线一定是测地线, 测

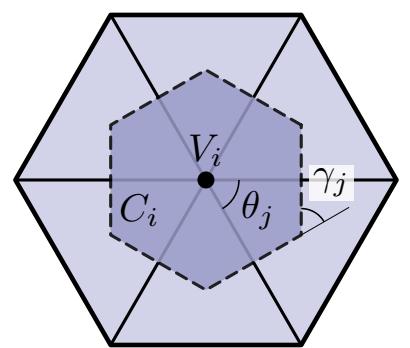


图 1.4: 离散曲面上的高斯曲率 (示意)

地曲率为零)。

此外, 图 1.4 中, 外角 γ_j 与对应的顶角 θ_j 相等 (因为 C_j 在每个与 V_i 的相邻三角面内的部分是一个具有两个直角内角的四边形), 对应三维离散曲面的情况如图 1.5 所示。

公式 1.5 移项并用 $\gamma_j = \theta_j$ 替换, 得

$$\iint_{C_i} K dA = 2\pi - \sum_{V_j \in N(V_i)} \theta_j. \quad (1.6)$$

在 C_i 内, 可按 Voronoi 元胞在每个三角面片上分别计算积分。近似认为高斯曲率 K 在 V_i 的邻接三角面内都相同, 所以有

$$\iint_{C_i} K dA = \sum_{V_j \in N(V_i)} \iint_{C_i \cap T_j} K dA = AK(V_i),$$

所以有 V_i 处的离散的高斯曲率

$$K(V_i) = \left(2\pi - \sum_{V_j \in N(V_i)} \theta_j \right) / A, \quad (1.7)$$

其中 A 为邻域 C_i 的面积。但是 BFF 方法中用到的离散高斯曲率并不是逐点的曲率, 而是沿着顶点 V_i 的小邻域 C_i 内的曲率的积分 $\Omega_i = \iint_{C_i} K dA$, 即公式 1.5 的形式。

项 $2\pi - \sum_{V_j \in N(V_i)} \theta_j$ 项表示顶点的顶角之和与 2π 的差, 可以理解为将 V_i 的邻域沿一个临边剪开并摊平时, 结果与一个圆周的距离, 也称之为“角缺”, 用记号 Ω 表示。 A 的求值可以拆分为多个子区域的面积 A_j 之和, 如图 1.6 所示。 C_i 的面积可以表示为多个图中的红色三角形面积之和, 红色三角形面积 A_j 为

$$A_j = \frac{1}{8} (\cot \angle L + \cot \angle N) \|V_i - M\|^2,$$

所以 A 为

$$A = \sum_{V_j \in N(V_i)} A_j = \sum_{V_j \in N(V_i)} \frac{1}{8} (\cot \theta_p^{ij} + \cot \theta_q^{ij}) \|V_i - V_j\|^2.$$

式中 θ_p^{ij} 和 θ_q^{ij} 分别表示以 $V_i V_j$ 为公共边的两个三角形中, 与边 $V_i V_j$ 相对的内角。 p 、 j 和 q 分别为 V_i 邻居中依序的三个节点下标。

在曲面 M 的边界处顶点的高斯曲率 K 为 0, 边界顶点的小邻域一定可以不拉伸地平摊到平面上。所以高斯曲率无法反映边界点处的几何性质。BFF 使用测地曲率:

$$\kappa_g = \pi - \sum_{j=1}^{N(V_i)} \theta_j \quad (1.8)$$

描述曲面边界的几何性质。在平面上, κ_g 也就是“边界外角”, 描述了沿边界行走时切向的角度变化。但是, 仅使用外角描述平面参数化的几

何边界还不够, 如图 1.7 所示。逐顶点的测地曲率对应了平面参数化边界顶点对应的外角 k_i 。图 1.7 中的两个几何边界在对应点处具有同样的测地曲率, 但是几何边界相差较大。BFF 方法为了控制参数化边界的形状, 从“考虑离散曲率”改为“考虑单位度量的离散曲率”:

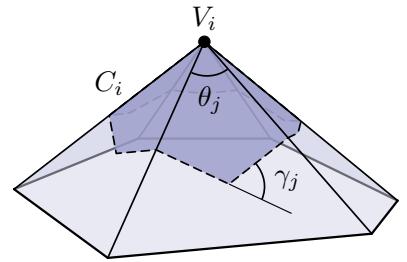


图 1.5: 离散曲面上的高斯曲率
(曲面)

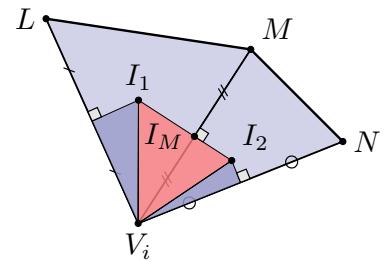


图 1.6: Voronoi 元胞面积计算

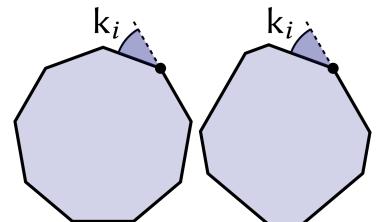


图 1.7: 逐顶点的测地曲率对
边界的约束

- 考虑“单位面积的角缺”，
- 考虑“单位长度的角度改变”。

以上两项都是“曲率密度”。

1.4.2.3 曲率密度

令 dA 和 ds 分别表示离散三维曲面上的面积微元和长度微元。对于三维曲面的离散高斯曲率和测地曲率，在共形映射下，映射前后的微元存在关系： $d\tilde{s} = e^u ds$ 和 $d\tilde{A} = e^{2u} dA$ ，新的曲率密度是旧的曲率密度乘曲率函数，即高斯曲率密度 $K dA$ 和测地曲率密度 κds 。

1.5 边界优先的平面参数化方法

BFF 方法依旧走的是确定平面参数化区域边界→从边界扩展到内部顶点的路线。基本上作业二和作业四都是这种路线。

1.5.1 泊松问题

我们的目标是找到黎曼度量下的共形映射 $f : M \rightarrow \mathbb{C}$ 。共形映射 f 的求解是一个泊松问题³。根据不同的边界条件，有迪利克雷一泊松问题：

$$\begin{cases} \Delta a = \phi, & \text{on } M, \\ a = g. & \text{on } \partial M. \end{cases} \quad (1.9)$$

和纽曼一泊松问题：

$$\begin{cases} \Delta a = \phi, & \text{on } M, \\ \frac{\partial a}{\partial \mathbf{n}} = h. & \text{on } \partial M. \end{cases} \quad (1.10)$$

其中 a 和 ϕ 是定义在 M 上的实值函数， $g : \partial M \rightarrow \mathbb{R}$ 和 $h : \partial M \rightarrow \mathbb{R}$ 分别表示沿边界的值和法线导数。问题 1.10 的解之间只差一个常数，在 BFF 中决定了全局缩放和全局平移。

在一个离散三角曲面上，沿对偶单元（即图 1.4 中的深色区域 C_i ）对方程 $\Delta a = \phi$ 进行积分，可以得到矩阵方程：

$$\mathbf{A}a = \mathbf{P}\phi, \quad (1.11)$$

其中矩阵 $\mathbf{A} \in \mathbb{R}^{|V| \times |V|}$ 叫做 cotan-Laplace 矩阵， $\mathbf{P} \in \mathbb{R}^{|V| \times |V|}$ 。矩阵 \mathbf{A} 的非零项是

$$A_{ij} = -\frac{1}{2}(\cot \beta_p^{ij} + \cot \beta_q^{ij}), \quad (1.12)$$

而 p 和 q 分别是与边 $V_i V_j \in E$ 相对的角，对于每个顶点 $V_i \in V$ ，有

$$\mathbf{A}_{ii} = -\sum_{V_i V_j \in E} \mathbf{A}_{ij}.$$

这里要说一下从 $\Delta a = \phi$ 到公式 1.11 和公式 1.12 的过程，分别对矩阵 \mathbf{A} 和矩阵 \mathbf{P} 的来源进行讨论。

³ 泊松问题，指求解满足泊松方程

$$\Delta \varphi = f \quad (\text{也常写作 } \nabla^2 \varphi = f)$$

的函数 φ ， Δ 为拉普拉斯算子。当 $f \equiv 0$ 时为拉普拉斯方程，详见 Poisson's Equation。

1.5.1.1 离散三维曲面上的 Laplacian

已知 f_M 是定义在离散三角区面上的实值函数。我们假设函数值是线性的（即三角面上的函数值可以由三角形顶点 V_i 、 V_j 和 V_k 处的函数值 $f_M(V_i)$ 、 $f_M(V_j)$ 和 $f_M(V_k)$ 线性插值得到），定义在顶点处的线性拉格朗日多项式（也称为“帽子函数”）基函数 $\phi_i(V_i)$ 如图 1.8 所示，满足：

$$\phi_i(V_j) = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{otherwise,} \end{cases} \quad (1.13)$$

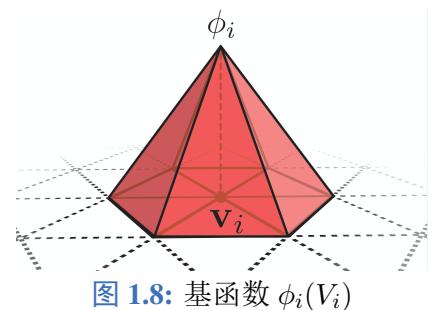


图 1.8：基函数 $\phi_i(V_i)$

于是沿曲面 M 对函数 a 进行插值，有

$$f_M(\mathbf{x}) = \sum_{V_i \in V} f(V_i) \phi_i(\mathbf{x}).$$

上式中的 \mathbf{x} 是离散顶点 V_i 到二维连续笛卡尔坐标的映射，在顶点 V_i 处有 $f(\mathbf{x}) = f(V_i)$ 。则函数 f_M 的散度为 $\Delta f = \nabla \cdot (\nabla f)$ ，计算依赖于梯度的计算：

$$\nabla f_M(\mathbf{x}) = \nabla \left(\sum_{V_i \in V} f(V_i) \phi_i(\mathbf{x}) \right) = \sum_{V_i \in V} f(V_i) \nabla \phi_i(\mathbf{x}).$$

从上式可以看出，梯度是基于每个顶点的函数值 $f(V_i)$ 加权的基函数梯度 $\nabla \phi_i(\mathbf{x})$ 的线性组合。观察图 1.8，基函数 $\phi_i(V_i)$ 定义在每个顶点 V_i 及其 1—邻域内，且是线性的，所以在每个 V_i 及其 1—邻域内，基函数的梯度为常数。

我们考虑三角面 $V_i V_j V_k$ ，基函数 ϕ_i 在三角面上的取值如图 1.9 所示，图中 \mathbf{e}_0 和 \mathbf{e}_0^\perp 是两个互相垂直、模长相等的向量，且 \mathbf{e}_0^\perp 指向三角形的内部。三角形颜色由深至浅反映了 ϕ_i 的数值变化。容易看出， ϕ_i 的值沿向量 \mathbf{e}_0 方向保持不变，所以基函数的梯度方向与 \mathbf{e}_0 垂直，满足 $\nabla \phi_i \cdot \mathbf{e}_0 = 0$ ，即 $\nabla \phi_i = \alpha \mathbf{e}_0^\perp$ 。梯度沿方向 \mathbf{e}_\perp 的大小为 $1/h$ ，所以方向梯度为

$$\nabla \phi_i|_T = \frac{1}{h} \frac{\mathbf{e}_0^\perp}{\|\mathbf{e}_0^\perp\|} = \frac{1}{2A_T/\|\mathbf{e}_0\|} \frac{\mathbf{e}_0^\perp}{\|\mathbf{e}_0^\perp\|} = \frac{\mathbf{e}_0^\perp}{2A_T},$$

其中 A_T 为 $T = \triangle V_i V_j V_k$ 的面积。令 \mathbf{n}_i 表示曲面 M 上的顶点 V_i 处的面法线，则有 $\mathbf{e}_0^\perp = \mathbf{n}_i \times \mathbf{e}_0$ ，此处的顶点法线 \mathbf{n}_i 非常重要。

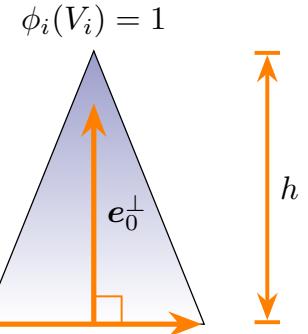


图 1.9：基函数的数值分布

对于 Δf_M ，有

$$\Delta f_M = \nabla \cdot \nabla f_M(\mathbf{x}) = \nabla \cdot \left(\sum_{V_i \in V} f(V_i) \nabla \phi_i(\mathbf{x}) \right),$$

在 V_i 的 1—邻域内，各个三角面上 $\nabla f_M(\mathbf{x})$ 是与 \mathbf{x} 无关的常向量。然而相邻的三角面内 $\nabla f_M(\mathbf{x})$ 不相同，是分片线性函数。所以 $\Delta f_M(\mathbf{x})$ 不能直接计算。我们考虑高斯散度定理⁴ 的二维形式——格林公式⁵：

$$\iint_R \nabla \cdot \mathbf{F} \, dA = \oint_{\partial R} \mathbf{F} \cdot \mathbf{n} \, ds,$$

⁴高斯散度定理

⁵Green's theorem

其中 R 定义了向量场 \mathbf{F} 的区域, $\nabla \cdot \mathbf{F}$ 是向量场 \mathbf{F} 的散度, 面积微元 dA , 区域边界 ∂R , \mathbf{n} 是曲线 C 上向外的法线。使用格林公式, 我们可以避免显式计算散度, 只需要沿着区域边界对梯度和边界法线的点积进行积分。

对于离散化的三角网格曲面 M , $\nabla f_M(\mathbf{x})$ 是定义在 M 上的向量场。考虑将曲面 M 划分为如图 1.4 和图 1.5 所示的 Voronoi 元胞 C_i , 则

$$\iint_M \nabla \cdot \nabla f_M(\mathbf{x}) dA = \sum_{V_i \in V} \iint_{C_i} \nabla \cdot \nabla f_M(\mathbf{x}) dA = \sum_{V_i \in V} \oint_{\partial C_i} \nabla f_M(\mathbf{x}) \cdot \mathbf{n} ds,$$

考虑 C_i 在三角形 $T_j = \triangle V_i V_j V_k$ 的区域 $C_i \cap T_j$, 有

$$\begin{aligned} \oint_{\partial C_i} \nabla f_M(\mathbf{x}) \cdot \mathbf{n} ds &= \sum_{V_j \in N(V_i)} \int_{\partial C_i \cap T_j} \nabla f_M(\mathbf{x}) \cdot \mathbf{n} ds \\ &= \sum_{V_j \in N(V_i)} \nabla f_M|_{T_j} \cdot \int_{\partial C_i \cap T_j} \mathbf{n} ds \\ &= \sum_{V_j \in N(V_i)} \nabla f_M|_{T_j} \cdot (l_j \mathbf{n}_j + l_k \mathbf{n}_k), \end{aligned}$$

其中 ∇f_{T_j} 是三角形 T_j 内的梯度, 与 $\mathbf{x} \in T_j$ 无关。

注 在查找到的某些资料中, 会有如下的“平均”定义:

$$\Delta f_M(V_i) = \frac{1}{|C_i|} \iint_{C_i} \nabla \cdot \nabla f_M(\mathbf{x}) dx,$$

相较于推导的结果, 此定义多了在分母上的面积项 $|C_i|$, 但是目前还不清楚为什么要定义“平均”项。

Voronoi 元胞 $C_i \cap T_j$ 的边界法向如图 1.11 所示, 对于 $T_j = V_i V_j V_k$ 中的彩色小三角形, 三角形边长 l_i 、 l_j 和 l_k 以及对应于边的外法向量 \mathbf{n}_i 、 \mathbf{n}_j 和 \mathbf{n}_k 满足⁶:

$$l_i \mathbf{n}_i + l_j \mathbf{n}_j + l_k \mathbf{n}_k = 0,$$

所以

$$l_j \mathbf{n}_j + l_k \mathbf{n}_k = -l_i \mathbf{n}_i = -\frac{\|V_j - V_k\|}{2} \frac{\mathbf{e}_i}{\|\mathbf{e}_i\|} = \frac{\mathbf{e}_i^\perp}{2},$$

于是点 V_i 处的散度 $\nabla f_M(\mathbf{x})$ 沿环路的积分为

$$\begin{aligned} \oint_{\partial C_i} \nabla f_M(\mathbf{x}) \cdot \mathbf{n} ds &= \sum_{V_j \in N(V_i)} \nabla f_M|_{T_j} \cdot \frac{\mathbf{e}_i^\perp}{2} \\ &= \sum_{V_j \in N(V_i)} [f_M(V_i) \nabla \phi_i|_{T_j} + f_M(V_j) \nabla \phi_j|_{T_j} + f_M(V_K) \nabla \phi_k|_{T_j}] \cdot \frac{\mathbf{e}_i^\perp}{2} \\ &= \sum_{V_j \in N(V_i)} \left[f_M(V_i) \frac{\mathbf{e}_i^\perp}{2 A_{T_j}} + f_M(V_j) \frac{\mathbf{e}_j^\perp}{2 A_{T_j}} + f_M(V_K) \frac{\mathbf{e}_k^\perp}{2 A_{T_j}} \right] \cdot \frac{\mathbf{e}_i^\perp}{2} \\ &\stackrel{\substack{\mathbf{e}_i^\perp + \mathbf{e}_j^\perp + \mathbf{e}_k^\perp = \mathbf{0} \\ \text{三边向量和为零} \\ \text{垂向量和也为零}}}{=} \sum_{V_j \in N(V_i)} \left[-f_M(V_i) \left(\frac{\mathbf{e}_j^\perp}{2 A_{T_j}} + \frac{\mathbf{e}_k^\perp}{2 A_{T_j}} \right) + f_M(V_j) \frac{\mathbf{e}_j^\perp}{2 A_{T_j}} + f_M(V_K) \frac{\mathbf{e}_k^\perp}{2 A_{T_j}} \right] \cdot \frac{\mathbf{e}_i^\perp}{2} \\ &= \sum_{V_j \in N(V_i)} \left[(f_M(V_j) - f_M(V_i)) \frac{\mathbf{e}_j^\perp}{2 A_{T_j}} + (f_M(V_K) - f_M(V_i)) \frac{\mathbf{e}_k^\perp}{2 A_{T_j}} \right] \cdot \frac{\mathbf{e}_i^\perp}{2} \end{aligned}$$

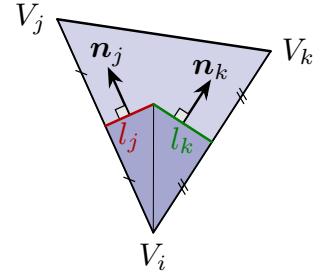


图 1.10: $C_i \cap T_j$

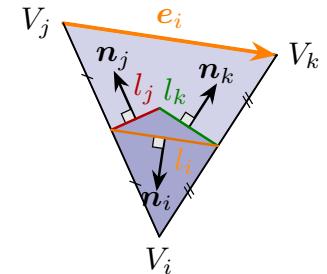


图 1.11: 法向关系

⁶ 三角形三边向量和为 $\mathbf{0}$, 所以垂直于三边的、对应模长相等的向量之和也为 $\mathbf{0}$

$$\begin{aligned}
&= \sum_{V_j \in N(V_i)} (f_M(V_j) - f_M(V_i)) \frac{\mathbf{e}_j^\perp \cdot \mathbf{e}_i^\perp}{4A_{T_j}} + (f_M(V_k) - f_M(V_i)) \frac{\mathbf{e}_k^\perp \cdot \mathbf{e}_i^\perp}{4A_{T_j}} \\
&= \frac{1}{2} \sum_{V_j \in N(V_i)} (f_M(V_j) - f_M(V_i)) \cot \beta_k^{ij} + (f_M(V_k) - f_M(V_i)) \cot \beta_j^{ik}.
\end{aligned}$$

倒数第二步用到了如下关系：

$$\left\{
\begin{array}{l}
\frac{\mathbf{e}_j^\perp \cdot \mathbf{e}_i^\perp}{4A_{T_j}} = \frac{\|\mathbf{e}_j^\perp\| \|\mathbf{e}_i^\perp\| \cos(\pi - \beta_k^{ij})}{2\|\mathbf{e}_j^\perp\| \|\mathbf{e}_i^\perp\| \sin \beta_k^{ij}} = \frac{1}{2} \cot \beta_k^{ij}, \\
\frac{\mathbf{e}_k^\perp \cdot \mathbf{e}_i^\perp}{4A_{T_j}} = \frac{\|\mathbf{e}_k^\perp\| \|\mathbf{e}_i^\perp\| \cos(\pi - \beta_j^{ik})}{2\|\mathbf{e}_k^\perp\| \|\mathbf{e}_i^\perp\| \sin \beta_j^{ik}} = \frac{1}{2} \cot \beta_j^{ik}.
\end{array}
\right.$$

重排求和项，得到

$$\oint_{\partial C_i} \nabla f_M(\mathbf{x}) \cdot \mathbf{n} \, ds = \sum_{V_j \in N(V_i)} \frac{1}{2} (\cot \beta_p^{ij} + \cot \beta_q^{ij}) (f_M(V_j) - f_M(V_i)), \quad (1.14)$$

其中 p, q 为边 $V_i V_j$ 的两个对角。上式左右分别对应了公式 1.12 所示的非零项 A_{ii} 以及 A_{ij} 。这里给出的是 $\Delta f_M(V_i)$ 的几何解释。

1.5.1.2 矩阵方程 $\mathbf{A}a = \mathbf{P}\phi$

公式 1.14 计算的是三维离散曲面的内点处的 Laplacian 在 Voronoi 元胞 C_i 区域上的积分值。但是这样的讨论没有涉及到方程 1.11 右端的质量矩阵 \mathbf{P} 。我们还需要从纽曼—泊松问题 1.10 出发，考虑其在曲面 M 上的积分弱形式⁷，注意这里 M 是三维无边界曲面，方程在边界情况需要单独讨论。引入试函数对方程 $\Delta a = \phi$ 在区域 M 上进行面积分（使用 $f_M(\mathbf{x})$ 代替 a ，用 φ 代替 ϕ ），有

$$\iint_M \psi \Delta f_M \, dA = \iint_M \psi \varphi \, dA,$$

其中

$$f_M(\mathbf{x}) = \sum_{V_j \in V} f_M(V_j) \phi_j(\mathbf{x}), \quad \psi(\mathbf{x}) = \sum_{V_i \in V} \psi(V_i) \phi_i(\mathbf{x}), \quad \varphi(\mathbf{x}) = \sum_{V_j \in V} \varphi(V_j) \phi_j(\mathbf{x}),$$

所以

$$\iint_M \left(\sum_{V_i \in V} \psi(V_i) \phi_i(\mathbf{x}) \right) \Delta \left(\sum_{V_j \in V} f_M(V_j) \phi_j(\mathbf{x}) \right) \, dA = \iint_M \left(\sum_{V_i \in V} \psi(V_i) \phi_i(\mathbf{x}) \right) \left(\sum_{V_j \in V} \varphi(V_j) \phi_j(\mathbf{x}) \right) \, dA,$$

即

$$\sum_{V_i, V_j \in V} \psi(V_i) f_M(V_j) \iint_M \phi_i \Delta \phi_j \, dA = \sum_{V_i, V_j \in V} \psi(V_i) \varphi(V_j) \iint_M \phi_i \phi_j \, dA.$$

⁷这个好像是有限元的方法，参考一个回答

What does "test function" mean?

假设你想要找到一个微分方程的解，比如 $f'' = gf$ ，我们通常要计算在给定点处的解的数值。然而一些函数（或者函数等价类）的空间（如 L^p 空间）内某点处的函数值不能很好地反映底层函数，甚至不具有任何意义。考虑加权平均值来评估函数更好，选择合适的函数 ψ 对函数进行平均，即

$$T_f(\psi) = \int_{\mathbb{R}} f(x) \phi(x) \, dx.$$

为了处理二阶导数 $\text{Delta}\phi_j$, 考虑如下恒等式:

$$\nabla \cdot (\phi_i \nabla \phi_j) = \phi_i \Delta \phi_j + \nabla \phi_i \cdot \nabla \phi_j \implies \phi_i \Delta \phi_j = \nabla \cdot (\phi_i \nabla \phi_j) - \nabla \phi_i \cdot \nabla \phi_j,$$

所以

$$\begin{aligned} \iint_M \phi_i \Delta \phi_j \, dA &= \iint_M \nabla \cdot (\phi_i \nabla \phi_j) - \nabla \phi_i \cdot \nabla \phi_j \, dA \\ &= \underbrace{\oint_{\partial M} \mathbf{n} \cdot (\phi_i \nabla \phi_j) \, ds}_{\text{曲面 } M \text{ 不存在边界}} + \iint_M \nabla \phi_i \cdot \nabla \phi_j \, dA \\ &= \iint_M \nabla \phi_i \cdot \nabla \phi_j \, dA. \end{aligned}$$

公式和矩阵项的对应关系为

$$\sum_{V_i, V_j \in V} \psi(V_i) f_M(V_j) \underbrace{\iint_M \phi_i \Delta \phi_j \, dA}_{\mathbf{A}_{ij}} = \sum_{V_i, V_j \in V} \psi(V_i) \varphi(V_j) \underbrace{\iint_M \phi_i \phi_j \, dA}_{\mathbf{P}_{ij}},$$

即可表示为矩阵形式

$$\boldsymbol{\psi}^\top \mathbf{A} f_M = \boldsymbol{\psi}^\top \mathbf{P} \boldsymbol{\varphi}, \forall \boldsymbol{\psi} \implies \mathbf{A} \mathbf{a} = \mathbf{P} \boldsymbol{\phi}.$$

并且

$$\begin{aligned} \mathbf{A}_{ij} &= \iint_M \nabla \phi_i \cdot \nabla \phi_j \, dA \\ &= \begin{cases} \sum_{V_i, V_j \in T} \frac{\mathbf{e}_{T,i}^\perp}{2A_T} \cdot \frac{\mathbf{e}_{T,j}^\perp}{2A_T} A_T = \frac{1}{2}(\cot \beta_p^{ij} + \cot \beta_q^{ij}), & i \neq j \\ \sum_{V_i \in T} \frac{\mathbf{e}_{T,i}^\perp \cdot (-\mathbf{e}_{T,j}^\perp - \mathbf{e}_{T,k}^\perp)}{4A_T} = -\frac{1}{2} \sum_{V_j \in N(V_i)} (\cot \beta_p^{ij} + \cot \beta_q^{ij}), & i = j. \end{cases} \\ \mathbf{P}_{ij} &= \iint_M \phi_i \phi_j \, dA \\ &= \begin{cases} \frac{1}{12}(A_{\triangle V_i V_j V_p} + A_{\triangle V_i V_j V_q}), & i \neq j, \\ \iint_M \phi_i^2 \, dA = \frac{1}{6} \sum_{V_j \in N(V_i)} A_{T_j}, & i = j. \end{cases} \end{aligned}$$

质量矩阵 \mathbf{P} 的计算比较麻烦, 就略过了 (其实是自己不会算 \mathbf{hh} , 如果有看到的大神可以帮我推一下)。于是我们就得到了离散曲面 M 的内点处的 \mathbf{A} 和 \mathbf{P} 的计算方法, 但是边界点 $V_i \in \partial M$ 的情况还需要进后续明确。BFF 没有用到质量矩阵 \mathbf{P} , 公式 1.11 的右端项全为积分量, 例如离散高斯曲率 K 等。

1.5.1.3 cotan-Laplacian 矩阵 \mathbf{A}

对于纽曼一泊松问题 1.10, 可以把矩阵方程 1.11 矩阵 \mathbf{A} 按内部顶点和边界顶点划分为子矩阵, 即

$$\begin{bmatrix} \mathbf{A}_{II} & \mathbf{A}_{IB} \\ \mathbf{A}_{IB}^\top & \mathbf{B}_{BB} \end{bmatrix} \begin{bmatrix} a_I \\ a_B \end{bmatrix} = \begin{bmatrix} \phi_I \\ \phi_B - h \end{bmatrix}, \quad (1.15)$$

其中 $h \in \mathbb{R}^{|B|}$, 是离散纽曼边界条件, 对应于边界条件 $\partial a / \partial \mathbf{n}$ 沿边界对偶边 e_i 的积分。对偶边即边界在点 V_i 的 Voronoi 元胞 C_i 内的部分 $\partial M \cap C_i$ 。

对于迪利克雷一泊松问题 1.9, 边界条件 $a_B = g \in \mathbb{R}^{|B|}$ 可由内点值 a_I 和公式 1.15 求解得到:

$$\mathbf{A}_{II}a_I = \phi_I - \mathbf{A}_{IB}g. \quad (1.16)$$

为了快速求解矩阵方程 1.11, 使用稀疏矩阵的 Cholesky 分解 $\mathbf{A} = \mathbf{L}\mathbf{L}^\top$:

$$\begin{bmatrix} \mathbf{A}_{II} & \mathbf{A}_{IB} \\ \mathbf{A}_{IB}^\top & \mathbf{B}_{BB} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{II} & \mathbf{0} \\ \mathbf{L}_{BI} & \mathbf{L}_{BB} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{II}^\top & \mathbf{L}_{BI}^\top \\ \mathbf{0} & \mathbf{L}_{BB}^\top \end{bmatrix},$$

于是迪利克雷问题的 Cholesky 分解可由左上矩阵给出: $\mathbf{A}_{II} = \mathbf{L}_{II}\mathbf{L}_{II}^\top$ 。

注 \mathbf{A} 的 Cholesky 分解贯穿了 BFF 方法, 因为 BFF 只涉及到泊松问题。但是对于 LSCM/SCP 问题就没有类似的方法了。

1.5.2 Cherrier 公式

共形映射下的曲率变化与缩放因子 u 有很紧密的关系。对于具有边界条件的共形映射问题, 在共形映射 $f: M \rightarrow \tilde{M}$ 下的任意两个曲面, 有如下关系成立:

$$\begin{aligned} \Delta u &= K - e^{2u}\tilde{K} && \text{on } M \\ \frac{\partial u}{\partial n} &= \kappa_g - e^u\tilde{\kappa}_g && \text{on } \partial M, \end{aligned} \quad (1.17)$$

其中 Δ 是曲面上的 Laplacian。 \tilde{K} 和 $\tilde{\kappa}_g$ 是曲面 \tilde{M} 的高斯曲率和测地曲率。但是公式 ?? 极大程度上忽略了边界信息, BFF 方法在公式两端乘了面积微元, 将曲率改为了曲率密度:

$$\Delta u \, dA = K \, dA - \tilde{K} \, d\tilde{A} \quad \text{on } M \quad (1.18)$$

$$\frac{\partial u}{\partial n} \, ds = \kappa_g \, ds - \tilde{\kappa}_g \, d\tilde{s} \quad \text{on } \partial M, \quad (1.19)$$

在顶点 $V_i \in V$ 的 Voronoi 元胞内积分公式 1.18, 可得

$$\mathbf{A}u = \Omega - \tilde{\Omega}, \quad (1.20)$$

其中 Ω 和 $\tilde{\Omega}$ 分别表示原象和映像的角缺。为了达到平面参数化的目的, 有 $\tilde{\Omega} = 0$ 。类似地, 沿对偶边 $e_i = C_i \cap \partial M$ 积分公式 1.19, 可得

$$h = \kappa_g - \tilde{\kappa}_g, \quad (1.21)$$

其中 h 是离散纽曼边界条件数据, κ_g 和 $\tilde{\kappa}_g$ 分别是原象和映像的边界曲率。

注 BFF 中如果使用缩放因子 u 调整边长, 就无法获得精确的目标曲率。BFF 直接从离散 Cherrier 公式中获得角度, 并精确满足边界闭合条件 (命题 1) 和目标锥角 (命题 2) 的必要条件。

1.5.3 Poincaré-Steklov 算子

类似泊松问题等椭圆方程问题的边界值可以由多种边界条件表示。Poincaré-Steklov 算子将一个解的边界值映射到另一个边界值上, 使得解相同。我们需要两个这样的算子, 分别是对于泊松公式的 Dirichlet—Neumann 映射, 和柯西—黎曼方程的 Hilbert 变换。

1.5.3.1 Dirichlet—Neumann

给定一个如公式 1.9 所示的泊松问题的迪利克雷边界值 g , 目标是找到一个纽曼值 h , 并得到与之相同的解。当然可以通过计算迪利克雷边界条件的方程解, 然后计算边界上的法向导数, 但是这种方

法的表现不佳。更本质的方法是求解纽曼数据 h , 以精确再现离散迪利克雷边界值的特解。对于拉普拉斯问题 (即 $\phi = 0$), 求解方程 1.15 和 1.16 中的 h 就得计算 \mathbf{A} 的舒尔补 (章节 A.2):

$$h = (\mathbf{A}_{IB}^\top \mathbf{A}_{II}^{-1} \mathbf{A}_{IB} - \mathbf{A}_{BB})g. \quad (1.22)$$

对于具有非零源项 ϕ 的泊松问题, Dirichlet—Neumann 映射变为仿射算子:

$$\Lambda_\phi g = \phi_B - \mathbf{A}_{IB}^\top \mathbf{A}_{II}(\phi_I - \mathbf{A}_{IB}g) - \mathbf{A}_{BB}g. \quad (1.23)$$

公式 1.23 可以通过线性解法器对 \mathbf{A}_{II} 进行预分解求解。实际上 Λ_ϕ 求解的是迪利克雷—泊松方程, 并取 ϕ 与泊松解在边界点处的 Laplacian 值的差值。

1.5.3.2 Neumann—Dirichlet

相较于上一节, 反过来的计算过程更加直接, 只需要求解纽曼—泊松问题 (公式 1.10) 得到 a 并截取边界 $g = a_B$, 解之间只差一个常数, 用伪逆 Λ_ϕ^\dagger 表示纽曼—迪利克雷映射。

1.5.4 Hilbert 变换

在类圆盘域, Hilbert 变换 \mathcal{H} 将调和函数 a 的切导数映射到它的共轭调和函数 b 的法导数⁸, 为全纯函数 $f = a + ib$ 提供边界数据。我们要找一对在顶点上具有自由度的标准分片线性函数, 基本观点是固定 a 并求解 b 使得最小二乘共形能量 E_C 最小, 能量用于度量 f 不满足柯西—黎曼方程的情况。此能量可以表示为 Dirichlet 能量减去映射像的面积。 E_C 的形式如下:

$$E_C(\mathbf{a}, \mathbf{b}) = \begin{bmatrix} \mathbf{a}^\top & \mathbf{b}^\top \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{U} \\ \mathbf{U}^\top & \mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}, \quad (1.24)$$

其中 \mathbf{U} 表示边界多边形的定向面积:

$$\mathbf{a}^\perp \mathbf{U} \mathbf{b} = \frac{1}{2} \sum_{V_i V_j \in \partial M} \mathbf{a}_j \mathbf{b}_i - \mathbf{a}_i \mathbf{b}_j. \quad (1.25)$$

要固定 \mathbf{a} 最小化 E_C , 需要求解纽曼—拉普拉斯方程 $\mathbf{A}\mathbf{b} = -\mathbf{U}^\top \mathbf{a}$ 。纽曼边界条件可以将公式 1.25 对 \mathbf{b} 求导得到:

$$\mathbf{h}_j = \frac{1}{2}(\mathbf{a}_k - \mathbf{a}_i). \quad (1.26)$$

其中 i 、 j 和 k 是沿着边界的三个连续的点。

1.5.5 插值

当我们想要把给定的表示边界的函数 $\tilde{\gamma} : \partial \rightarrow \mathbb{C}$ 扩展到区域内部, 即找到映射 $f : M \mapsto \mathbb{C}$ 使得 $f|_{\partial M} = \tilde{\gamma}$ 时, 简单的方法是通过调和函数对每个 $\tilde{\gamma}$ 分别插值, 即求解拉普拉斯方程:

$$\begin{cases} \Delta a = 0, & \text{使得 } a|_{\partial M} = \Re(\tilde{\gamma}), \\ \Delta b = 0, & \text{使得 } b|_{\partial M} = \Im(\tilde{\gamma}). \end{cases}$$

⁸法导数是沿空间中某个表面的法线方向 (即正交方向) 的方向导数, 或更一般地沿某个超曲面正交的法向量场的方向导数, 可以参考纽曼边界条件 (Neumann boundary condition) 如果法线方向用 \mathbf{n} 表示, 那么函数 f 的法线导数为

$$\frac{\partial f}{\partial \mathbf{n}} = \nabla f(\mathbf{x}) \cdot \mathbf{n} = \nabla_{\mathbf{n}} f(\mathbf{x}) = \frac{\partial f}{\partial \mathbf{x}} \cdot \mathbf{n} = Df(\mathbf{x})[\mathbf{n}].$$

如果 $\tilde{\gamma}$ 已经兼容了全纯映射 f , Hilbert 变换 \mathcal{H} 提供了一个不同的策略:

(1) (调和扩展) 求解 $\Delta a = 0$ 使得 $a|_{\partial M} = \Re(\tilde{\gamma})$ 。

(2) (调和共轭) 求解 $\Delta b = 0$ 使得 $\partial b / \partial n = \mathcal{H}a$,

其中问题(2)的离散纽曼数据需要通过公式 1.26 计算。这两种策略面向不同的应用场景和需求。前者的 f 可以对 $\tilde{\gamma}$ 进行精确插值, 但是不保证是全纯的。后者的 f 是全纯的, 但是可能无法精确插值 $\tilde{\gamma}$ 的两个分量。当 $\tilde{\gamma}$ 来自全形映射时, 两个策略得到的结果相同。这两种策略最适合不同的应用, 详情可查看章节 1.6 中对不同策略的简单讨论。

1.5.6 曲线积分

BFF 的关键步骤是从给定的曲率和长度数据得到闭合边界曲线 $\tilde{\gamma}$ 。对于连续的情况, 边界可以直接对数据积分得到。但是离散情况下离散误差会导致边界不闭合, 所以需要找到逼近给定数据的边界。直接优化顶点位置是一个困难的非线性问题, BFF 方法考虑了通过最小化调整长度来封闭曲线的凸问题。BFF 方法通过对曲率进行积分来解决这个问题:

$$\varphi(t) = \int_0^t \kappa_g(s) \, ds.$$

然后沿着边界构造单位切向量 $\mathbf{T}(s) = e^{i\varphi(s)}$, 并求解

$$\min_{r: \partial M \rightarrow \mathbb{R}} \frac{1}{2} \int_0^{2\pi} [r(s) - 1]^2 \, ds \quad \text{使得} \quad \int_0^{2\pi} r(s) \mathbf{T}(s) \, ds = 0. \quad (1.27)$$

如果 κ_g 已经描述了一个闭合环路, 那么有 $r(s) \equiv 1$; 否则 r 会最小化地调整曲线的速度使之闭合。其他情况下, 最终的边界曲线通过对缩放过地切向量积分得到:

$$\tilde{\gamma}(t) = \int_0^t r(s) \mathbf{T}(s) \, ds.$$

1.5.6.1 离散化

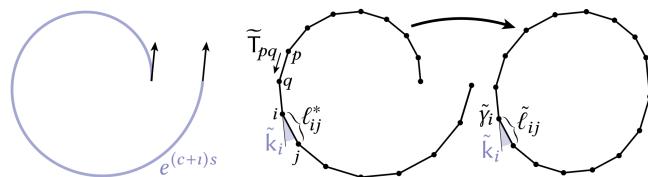


图 1.12: 逼近目标闭合环路

为了把公式 1.27 所示的问题离散化, 令 $\tilde{\mathbf{k}}$ 和 ℓ^* 分别表示目标外角和目标边界长度, 如图 1.12 所示。目标是找到一个以顶点 $\tilde{\gamma}_i \in \mathbb{C}$ 为顶点的多边形, 精确匹配目标边界外角, 接近和匹配目标边长。首先计算累计外角

$$\varphi_p = \sum_{i=1}^{p-1} \tilde{\kappa}_g, i$$

和目标切向量 $\tilde{\mathbf{T}}_{ij} = (\cos \varphi_i, \sin \varphi_i)$, 于是公式 1.27 变为

$$\min_{\tilde{\ell}: B \rightarrow \mathbb{R}} \frac{1}{2} \sum_{V_i V_j \in \partial M} \ell_{ij}^{-1} |\tilde{\ell}_{ij} - \ell_{ij}^*|^2 \quad \text{使得} \quad \sum_{V_i V_j \in \partial M} \tilde{\ell}_{ij} \tilde{\mathbf{T}}_{ij} = 0. \quad (1.28)$$

令 $\mathbf{N} \in \mathbb{R}^{|B| \times |B|}$ 是对角质量矩阵, 对角项为 $\mathbf{N}_{ii} = 1/\ell_{ii}$ 。将每一个单位切向量的两个分量打包到

矩阵 $\mathbf{T} \in \mathbb{R}^{2 \times |B|}$, 那么最优的边长由如下公式给出 (ℓ 表示边长 ℓ 对应的向量):

$$\tilde{\ell} = \ell^* - \mathbf{N}^{-1} \tilde{\mathbf{T}}^\top (\tilde{\mathbf{T}} \mathbf{N}^{-1} \tilde{\mathbf{T}}^\top)^{-1} \tilde{\mathbf{T}} \ell^*. \quad (1.29)$$

注意 $\tilde{\mathbf{T}} \mathbf{N}^{-1} \tilde{\mathbf{T}}^\top$ 是 2×2 的矩阵。最后的顶点位置可以直接通过如下公式获得:

$$\tilde{\gamma}_p = \sum_{i=1}^{p-1} \tilde{\ell}_{ij} \tilde{\mathbf{T}}_{ij}. \quad (1.30)$$

原理上 $\tilde{\ell}_{ij}$ 可能为负, 但是 BFF 的算例中没有出现这个情况, 比值 $\tilde{\ell}_i / \ell_i^*$ 一般在 $1 \pm .001$ 左右。

1.6 BFF 算法流程

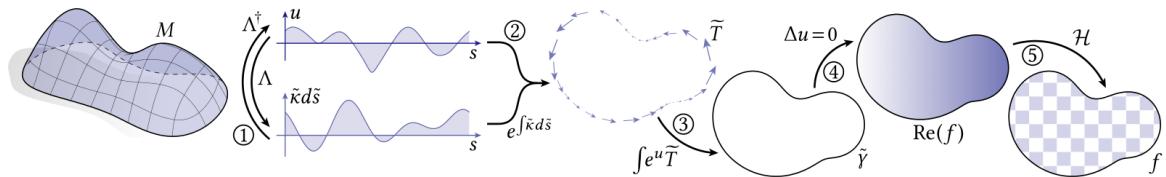


图 1.13: BFF 算法流程。① 给定曲面 M , 以及目标缩放因子 u 和或者沿目标边界的目标外角 \tilde{k} , 互补量由 Dirichlet—Neumann 映射 Λ 得到。② 对曲率密度积分得到单位切向量 \tilde{T} 。③ 将缩放后的切向量积分得到目标边界 $\tilde{\gamma}$ 。④ 将 $\tilde{\gamma}$ 的实部调和扩展。⑤ Hilbert 变换 \mathcal{H} 提供了坐标虚部, 最终得到最终的平面参数化映射 $f : M \mapsto \mathbb{C}$ 。

BFF 的方法流程如图 1.13 所示。总算法流程如算法 1 所示, 具体的算法步骤可查看附录章节 A.1。

Algorithm 1: BFF 算法流程

Input: 具有圆盘拓扑结构的三角化曲面 M 以及下列两个条件之一:

- ① 期望取到的缩放因子 u , 或者
- ② 沿边界顶点处的目标外角 \tilde{k} , 外角之和为 2π 。

Output: 分片线性映射 $f : V \mapsto \mathbb{C}$, 映射接近给定边界数据 (u 或者 \tilde{k}) 的共形映射。

1 计算边界数据的补:

- 如果给的是缩放因子 u , 计算与之兼容的外角:

► 章节 1.5.2

► 章节 1.5.3.1

$$\tilde{k} \leftarrow k - \Lambda_\Omega u.$$

- 如果给的是外角 \tilde{k} , 计算与之兼容的缩放因子:

► 章节 1.5.3.2

$$u \leftarrow \Lambda_\Omega^\dagger(k - \tilde{k}).$$

/*

*/

2 构造具有外角 \tilde{k} 和近似边长 $\ell_{ij}^* = e^{(u_i+u_j)/2} \ell_{ij}$ 的闭合曲线 $\tilde{\gamma}$;

► 章节 1.5.6

3 计算曲线 $\tilde{\gamma}$ 的全纯扩张 f ;

► 章节 1.5.5

对于 BFF 方法, 三维曲面 M 的平面参数化分为三种情况:

- (1) 将 M 进行自由边界参数化
- (2) 将 M 映射到单位圆盘区域
- (3) 将 M 映射到固定边界形状区域

一般来说, 一次基于公式 1.29 的闭合边界求解往往是不精确的, 所以 BFF 采取了多次 (10 次) 迭代, 得到最终的逼近结果 $\tilde{\ell}$ 。除了第一种情况, 对于单位圆盘和任意边界曲线都采取了这个策略。BFF 给出的例子中, 实际测试收敛得很快。

对于第一种情况，BFF 采用了“令映射边界的缩放因子为 0 ($u|_{\partial M} = 0$)”的策略。

对于第二种情况，均值化定理 (uniformization theorem) 保证了存在从曲面到单位圆盘的共形映射，映射具有常边界曲率 $\kappa_g \equiv 1$ 。然而，规定曲率密度恒定，得到的形状只是凸的，并非圆形。所以 BFF 我们采用了一种简单的定点方案：如果目标曲率 $\tilde{\kappa}_g$ 等于 1，那么目标曲率密度应该是 $\tilde{\kappa}_g d\tilde{s} = d\tilde{s} = e^u ds$ ，即新的长度密度。 n 次迭代后目标外角与最邻近的对偶边成正比：

$$\tilde{k}_i^n \leftarrow 2\pi \tilde{\ell}_i^{n-1} / \sum_{V_i \in B} \tilde{\ell}_i^{n-1}.$$

并且在迭代过程中还进行了平均：

$$\tilde{k}^n \leftarrow \frac{1}{2} (\tilde{k}^n + \tilde{k}^{n-1}).$$

对于第三种情况，采用第二种情况的策略，需要指定边界顶点对应的目标外角。平面曲面由曲线曲率 $\tilde{\kappa}_g$ 决定，BFF 方法通过迭代达到曲率密度 $\tilde{\kappa}_g e^{u^{n-1}} ds$ ，其中 u^{n-1} 是上一次对缩放因子的猜测。离散情况下，令 $\gamma^*(s) : [0, L] \mapsto \mathbb{C}$ 表示期望的闭曲线，弧长参数为 s 。令 $s_i = \sum_{k=1}^i \tilde{\ell}_{k,k+1}^n$ 表示当前迭代步下的累计边界长度，令 $S = \sum_i s_i$ 表示总长度。首先需要对 γ^* 进行采样，得到多边形顶点 $z_i = \gamma^*((L/S)s_i)$ ，与我们上一次迭代得到的边长区间成正比。然后计算采样后曲线的外角 $\tilde{k}_i = \arg((z_{i+1} - z_i)/(z_i - z_{i-1}))$ 。采样点 s_i 是根据最新的长度密度确定的，因此这些角度可以提供所需的曲率密度的近似值。

1.7 实验结果

我们使用六个模型测试本文方法的有效性（表 1.2），从自由边界参数化、单位圆盘参数化和固定边界形状（正多边形）三个方面测试方法的有效性和效率。

名称	格式	面类型	V	E	F	边界数	最长边界长度	存储
cathead	OBJ	三角面	131	378	248	1	12	8KB
Balls	OBJ	三角面	547	1578	1032	1	60	26KB
bunnyhead	OBJ	三角面	741	2188	1448	1	32	55KB
hand	OFF	三角面	1558	4653	3096	1	18	97KB
cow	OBJ	三角面	3195	8998	5804	1	584	194KB
face	OBJ	三角面	17157	51300	34149	1	168	1.13MB

表 1.2：测试数据

六组模型的平面着色渲染结果如图 1.14。其中如 1.14(f) 是 BFF 案例用到的模型。

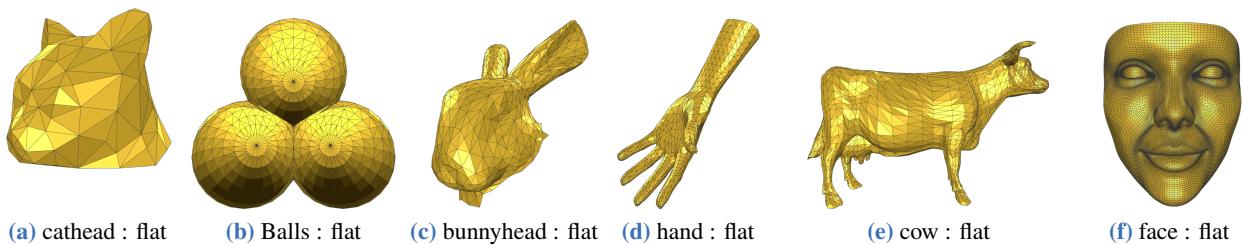


图 1.14：模型的平面着色渲染结果

各模型的自由边界参数化结果如图 1.15 所示。其中图 1.15(e) 的结果和 LSCM 十分接近。图 1.15(f)

的结果和 BFF 方法的结果相同。对于自由边界，BFF 方法可以得到很好的平面参数化结果。但是，BFF 方法存在天然的不足，在如图 1.15(i) 和 1.15(j) 中，存在大扭曲区域。

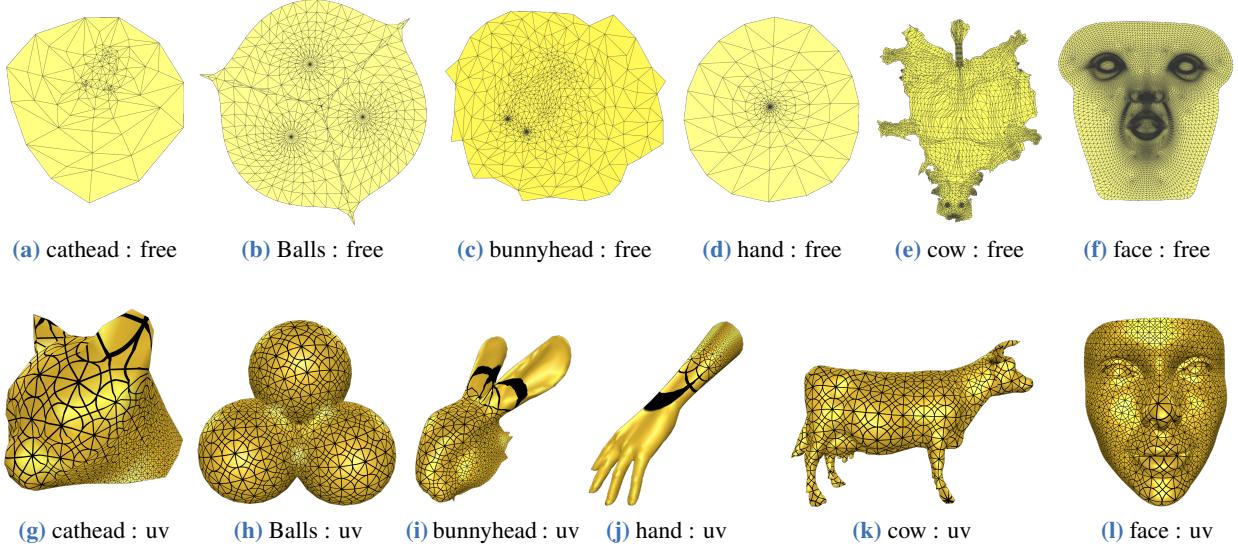


图 1.15：自由边界参数化结果

对于单位圆盘区域，BFF 方法的平面参数化结果如图 1.16 所示。大部分模型的参数化结果和自由边界接近，但图 1.16(k) 相较于图 1.15(k) 存在分布不均匀、局部较大扭曲的现象。一般这些区域表示模型存在较大扭曲的区域，如牛角、牛尾等区域。在这些区域，可能用 Cone 更好。

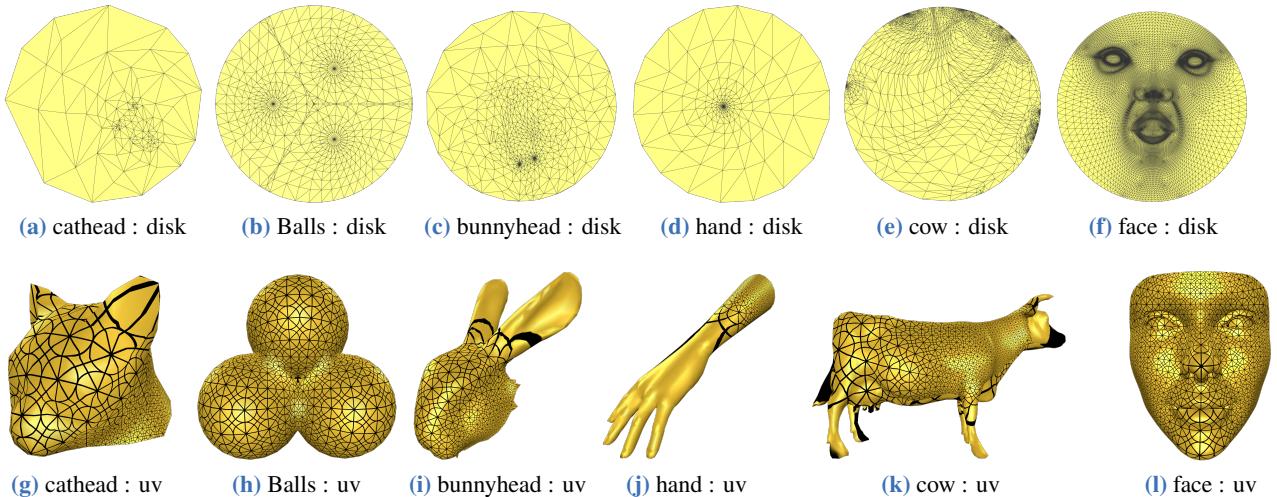


图 1.16：单位圆盘参数化结果

此外，本文还测试了正多边形的参数化结果（其实应该测一些任意形状多边形边界），如图 1.17、图 1.18 和图 1.19 所示。

本文实现的 BFF 方法用时如表 1.3 所示。用时与数据规模呈现的关系应当不只是线性关系，实际上矩阵的 Cholesky 分解与顶点的规模的关系是 $O(|V|^3)$ ，总体复杂度应当为

$$O(\underbrace{|V|}_{\text{顶点排序}} + \underbrace{|F|}_{\text{cotan-Laplace 矩阵}} + \underbrace{|V|^3}_{\text{Cholesky 分解 + 线性方程组求解}}) = O(|F| + |V|^3).$$

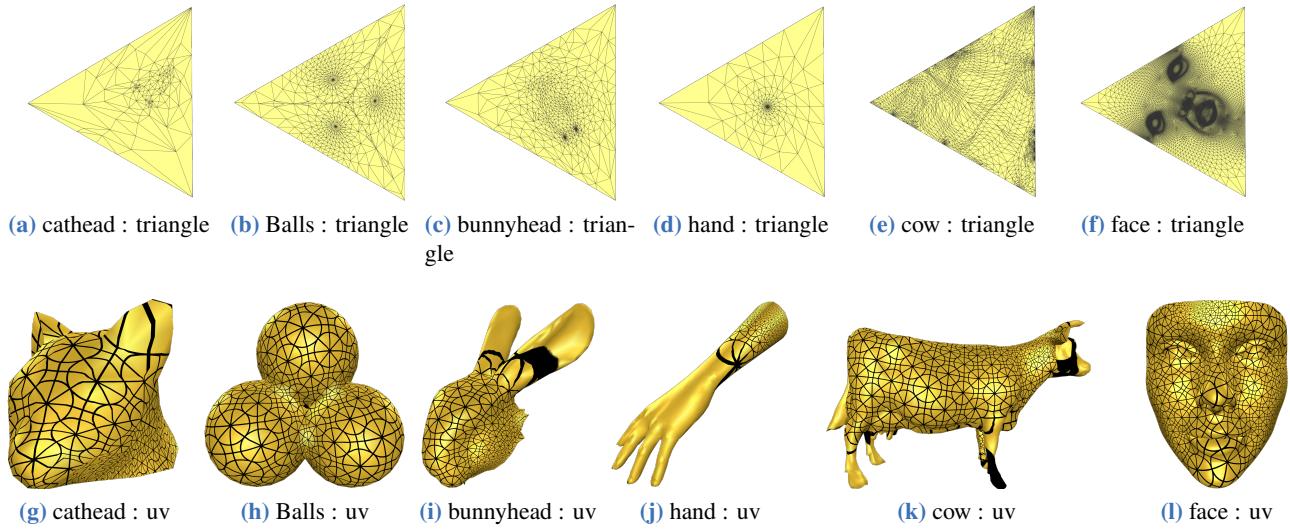


图 1.17: 三角形边界参数化

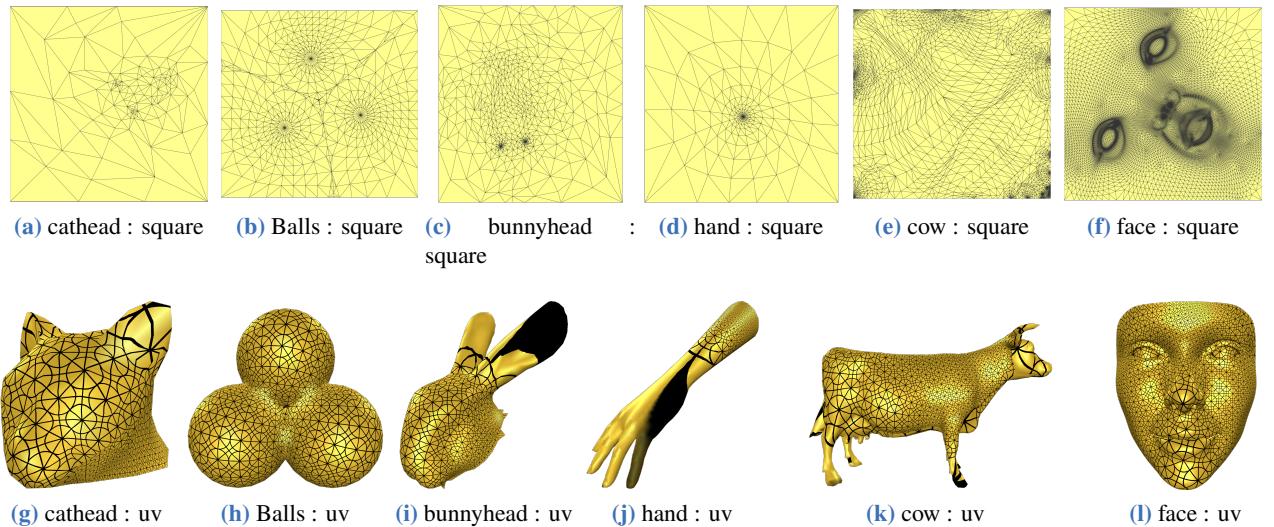


图 1.18: 正方形边界参数化

名称	V	E	F	总用时 (ms)
cathead	131	378	248	0.878
Balls	547	1578	1032	2.553
bunnyhead	741	2188	1448	3.479
hand	1558	4653	3096	8.199
cow	3195	8998	5804	110.963
face	17157	51300	34149	133.171
hilbert	79129	226896	147168	13min

表 1.3: 方法用时

表 1.3 中额外测试了大模型 Hilbert 的用时。相较于 BFF，自己的实现版本用时很长（13 分钟），是 BFF 方法的几百倍。考虑到自己基于 Eigen 实现，并没有采用第三方库做计算加速，可能采用一些高性能库会有更好的性能表现。

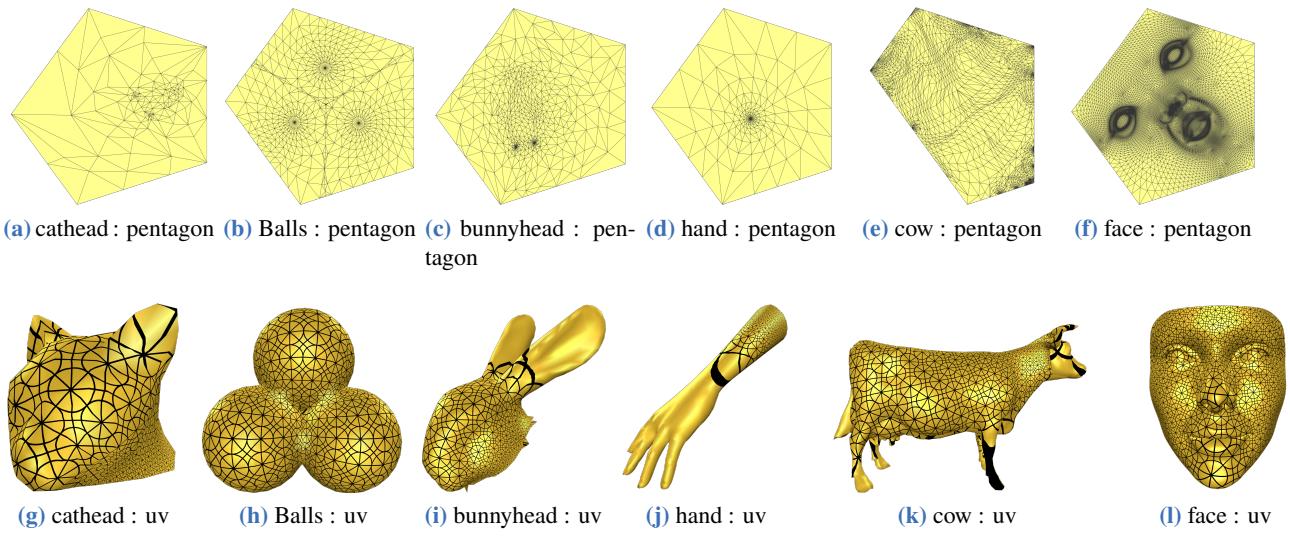


图 1.19: 五边形边界参数化

1.8 总结

本次作业实现了 [3] 提出的边界优先平面参数化方法 Boundary First Flattening。BFF 方法十分优秀，可以在极短的时间内得到（近似）满足任意边界的共形平面参数化结果。BFF 方法将“限制边界外角”改为“限制边界曲率密度”的思路也十分新颖。如果再结合曲面切割和二维空间的网格片排布，就可以得到一套优秀的曲面参数化工具。

附录 A 补充说明

A.1 算法流程

Algorithm 2: 边界优先的平面参数化算法: BoundaryFirstFlattening($M, \ell, [\mathbf{u}|\tilde{\mathbf{k}}]$)

Input: 具有圆盘拓扑的离散三角流形 $M = B \subseteq V, E, F$, 边长 $\ell : E \mapsto \mathbb{R}_{>0}$ 在面上满足三角不等式, 或者缩放因子 $\mathbf{u} : B \mapsto \mathbb{R}$ 或和为 2π 的外角 $\tilde{\mathbf{k}} : B \mapsto \mathbb{R}$

Output: 平面参数化映射 $f : V \mapsto \mathbb{C}$

```

1  $\beta \leftarrow \text{InteriorAngles}(M, \ell)$ ;                                ▶ 计算每个三角面的内角
2  $\Omega, \mathbf{k} \leftarrow \text{DiscreteCurvatures}(M, \beta)$ ;                      ▶ 计算离散曲率, 章节 1.4.2.2
3  $\mathbf{A} \leftarrow \text{BuildLaplace}(M, \beta)$ ;                                     ▶ 算法 3
4  $\mathbf{L} \leftarrow \text{CholeskyFactor}(\mathbf{A})$ ;                               ▶ 稀疏矩阵的 Cholesky 分解, 章节 1.5.1.3
5 if 给的是  $\mathbf{u}$  then
6    $\tilde{\mathbf{k}} \leftarrow \mathbf{k} - \text{DirichletToNeumann}(\mathbf{A}, \mathbf{L}, \Omega, \mathbf{u})$ ;      ▶ 算法 2
7 else
8    $\mathbf{u} \leftarrow \text{NeumannToDirichlet}(\mathbf{L}, -\Omega, \mathbf{k} - \tilde{\mathbf{k}})$ ;          ▶ 算法 5
9 end
10 foreach  $V_i V_j \in \partial M$  do  $\ell_{ij}^* \leftarrow e^{(\mathbf{u}_i + \mathbf{u}_j)/2} \ell_{ij}$ ;
11  $\tilde{\gamma} \leftarrow \text{BestFitCurve}(M, \ell, \ell^*, \tilde{\mathbf{k}})$ ;                         ▶ 算法 6
12 return  $\text{ExtendCurve}(M, \mathbf{L}, \tilde{\gamma})$ ;                                     ▶ 算法 7

```

Algorithm 3: 构造 Laplace 矩阵: BuildLaplace(M, ℓ)

Input: 边长为 ℓ 的曲面 M

Output: 一个 zero-Neumann 拉普拉斯矩阵 $\mathbf{A} \in \mathbb{R}^{|V| \times |V|}$

```

1  $\mathbf{A} \leftarrow \mathbf{0} \in \mathbb{R}^{|V| \times |V|}$ ;                                         ▶ 初始化一个空的稀疏矩阵
2 foreach 三角面  $V_p V_q V_r \in F$  do
3   foreach 循环遍历顶点  $ijk \in \mathcal{C}(V_p V_q V_r)$  do                                ▶  $\mathcal{C}$  指三角面的顶点序循环偏移
4      $\mathbf{A}_{ii}, \mathbf{A}_{jj}+ = \frac{1}{2} \cot \beta_k^{ij}$ ;
5      $\mathbf{A}_{ij}, \mathbf{A}_{ji}- = \frac{1}{2} \cot \beta_k^{ij}$ ;
6   end
7 end
8 return  $\mathbf{A}$ ;

```

Algorithm 4: 从 Dirichlet 到 Neumann: DirichletToNeumann($\mathbf{A}, \mathbf{L}, \phi, \mathbf{g}$)

Input: Zero-Neumann 拉普拉斯矩阵 \mathbf{A} 及其分解 \mathbf{L} , 源项 ϕ 和 Dirichlet 边界数据 $\mathbf{g} : B \mapsto \mathbb{R}$

Output: Neumann 数据 $\mathbf{h} : B \mapsto \mathbb{R}$

```

1 求解  $\mathbf{A}_{II}\mathbf{a} = \phi_I - \mathbf{A}_{IB}\mathbf{g}$ ;
2 return  $\phi_B - \mathbf{A}_{IB}^\top \mathbf{a} - \mathbf{A}_{BB}\mathbf{g}$ ;

```

Algorithm 5: 从 Neumann 到 Dirichlet: $\text{NeumannToDirichlet}(\mathbf{L}, \phi, \mathbf{h})$

Input: Zero-Neumann 拉普拉斯矩阵的 Cholesky 分解 \mathbf{A} , 源项 $\phi : V \mapsto \mathbb{R}$ 和 Neumann 数据

$$\mathbf{h} : B \mapsto \mathbb{R}$$

Output: Dirichlet 数据 $\mathbf{g} : B \mapsto \mathbb{R}$

1 求解 $\mathbf{A}\mathbf{a} = \phi - [\mathbf{0}; \mathbf{h}]$;

2 **return** \mathbf{a}_B ;

Algorithm 6: 计算最优匹配的边界曲线: $\text{BestFitCurve}(M, \ell, \ell^*, \tilde{\mathbf{k}})$

Input: 圆盘 (M, ℓ) , 目标边长 $\ell^* : B \mapsto \mathbb{R}_{>0}$, 以及和为 2π 的目标外角 $\tilde{\mathbf{k}} : B \mapsto \mathbb{R}$

Output: 顶点位置 $\tilde{\gamma} : B \mapsto \mathbb{C}$

```

1  $\tilde{\mathbf{T}} \leftarrow \mathbf{0} \in \mathbb{R}^{2 \times |B|}$ ;                                ▶ 存储切向量的稠密矩阵
2  $\varphi_{0,1} \leftarrow 0$ ;                                         ▶ 第一个切向量
3 for  $i = 1, \dots, |B| - 1$  do                                ▶ 沿边界顺序
4    $\varphi_{i,i+1} \leftarrow \varphi_{i-1,i} + \tilde{\mathbf{k}}_i$ ;          ▶ 累积外角
5    $\tilde{\mathbf{T}}_{i,i+1} \leftarrow e^{i\varphi_{i,i+1}}$ ;                ▶ 将切向量填入矩阵列
6 end
7  $\mathbf{N} \leftarrow \mathbf{0} \in \mathbb{R}^{|B| \times |B|}$ ;                      ▶ 边界质量矩阵
8 for  $V_i \in B$  do  $\mathbf{N}_{ii} \leftarrow 1/\ell_i$ ;           ▶  $\ell_i$  是对偶边
9  $\tilde{\ell} \leftarrow \ell^* - \mathbf{N}^{-1} \tilde{\mathbf{T}}^\top (\tilde{\mathbf{T}} \mathbf{N}^{-1} \tilde{\mathbf{T}}^\top)^{-1} \tilde{\mathbf{T}} \ell^*$ ;    ▶ 调整长度逼近边界
10  $\tilde{\gamma}_1 \leftarrow \mathbf{0} \in \mathbb{C}$ ;                           ▶ 将第一个顶点放在原点
11 for  $i = 2, \dots, |B|$  do                                ▶ 遍历其余顶点
12    $\tilde{\gamma}_i \leftarrow \tilde{\gamma}_{i-1} + \tilde{\ell}_{i-1,i} \tilde{\mathbf{T}}_{i-1,i}$ ;    ▶ 累计缩放后的切向量
13 end
14 return  $\tilde{\gamma}$ ;

```

Algorithm 7: 扩展曲线到区域内部: $\text{ExtendCurve}(M, \mathbf{L}, \tilde{\gamma})$

Input: 拓扑圆盘曲面 M , zero-Neumann 拉普拉斯矩阵的 Cholesky 因子 \mathbf{L} 和闭曲线 $\tilde{\gamma} : B \mapsto \mathbb{C}$

Output: 平面参数化映射 $f : V \mapsto \mathbb{C}$

```

1 求解  $\mathbf{A}_{II}\mathbf{a} = -\mathbf{A}_B \Re(\tilde{\gamma})$ ;                                ▶ 调和扩展
2 foreach  $V_i \in B$  do  $\mathbf{h}_i \leftarrow \frac{1}{2}(\mathbf{a}_{i+1} - \mathbf{a}_{i-1})$ ;          ▶ 希尔伯特变换
3 求解  $\mathbf{Ab} = \mathbf{h}$ ;                                         ▶ 调和共轭
4 return  $\mathbf{a} + i\mathbf{b}$ ;

```

A.2 舒尔补 (Schur complement)

舒尔补来自于求解线性方程组问题。对于非负整数 p 和 q , 由四个矩阵 $\mathbf{A}_{p \times p}$, $\mathbf{B}_{p \times q}$, $\mathbf{C}_{q \times p}$ 和 $\mathbf{D}_{q \times q}$ 组成的线性方程组:

$$\begin{cases} \mathbf{A}_{p \times p}\mathbf{x} + \mathbf{B}_{p \times q}\mathbf{y} = \mathbf{u}, \\ \mathbf{C}_{q \times p}\mathbf{x} + \mathbf{D}_{q \times q}\mathbf{y} = \mathbf{v}, \end{cases} \implies \begin{bmatrix} \mathbf{A}_{p \times p} & \mathbf{B}_{p \times q} \\ \mathbf{C}_{q \times p} & \mathbf{D}_{q \times q} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \implies \mathbf{M}_{(p+q) \times (p+q)}\mathbf{a} = \mathbf{b}. \quad (\text{A.1})$$

根据公式 A.1 可得

$$\mathbf{x} = \mathbf{A}^{-1}(\mathbf{u} - \mathbf{B}\mathbf{y})$$

代入公式 A.1 的第二个方程得到

$$(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})\mathbf{y} = \mathbf{v} - \mathbf{C}\mathbf{A}^{-1}\mathbf{u}.$$

则如果 \mathbf{A} 可逆, 定义子矩阵 \mathbf{A} 的舒尔补是 $q \times q$ 矩阵 $\mathbf{S}_{q \times q}$:

$$\mathbf{S}_{q \times q} := \mathbf{M}/\mathbf{A} := \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}.$$

使用 \mathbf{S} 可求得 \mathbf{y} :

$$\mathbf{y} = \mathbf{S}^{-1}(\mathbf{v} - \mathbf{C}\mathbf{A}^{-1}\mathbf{u}) = -\mathbf{S}^{-1}\mathbf{C}\mathbf{A}^{-1}\mathbf{u} + \mathbf{S}^{-1}\mathbf{v}, \quad \textcircled{1}$$

从公式 A.1 的第一个方程得到 $\mathbf{x} = \mathbf{A}^{-1}(\mathbf{u} - \mathbf{B}\mathbf{y})$, 并将上述公式代入, 得到

$$\mathbf{x} = (\mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1}\mathbf{C}\mathbf{A}^{-1})\mathbf{u} - \mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1}\mathbf{v}. \quad \textcircled{2}$$

结合公式 \textcircled{1} 和公式 \textcircled{2}, 可得

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1} \\ -\mathbf{S}^{-1}\mathbf{C}\mathbf{A}^{-1} & \mathbf{S}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix},$$

于是可以计算矩阵 $\mathbf{M}_{(p+q) \times (p+q)}$ 的逆:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1} \\ -\mathbf{S}^{-1}\mathbf{C}\mathbf{A}^{-1} & \mathbf{S}^{-1} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_p & -\mathbf{A}^{-1}\mathbf{B} \\ & \mathbf{I}_q \end{bmatrix} \begin{bmatrix} \mathbf{A}^{-1} & \\ & \mathbf{S}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I}_p & \\ -\mathbf{C}\mathbf{A}^{-1} & \mathbf{I}_q \end{bmatrix}.$$

类似地, 则如果 \mathbf{D} 可逆, 定义子矩阵 \mathbf{D} 的舒尔补是 $p \times p$ 矩阵 $\mathbf{T}_{p \times p}$:

$$\mathbf{T} := \mathbf{M}/\mathbf{D} := \mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C},$$

并且有矩阵 $\mathbf{M}_{(p+q) \times (p+q)}$ 的逆:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{T}^{-1} & -\mathbf{T}^{-1}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}\mathbf{T}^{-1} & \mathbf{D}^{-1} + \mathbf{D}^{-1}\mathbf{C}\mathbf{T}^{-1}\mathbf{B}\mathbf{D}^{-1} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_p & \\ -\mathbf{D}^{-1}\mathbf{C} & \mathbf{I}_q \end{bmatrix} \begin{bmatrix} \mathbf{T}^{-1} & \\ & \mathbf{D}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I}_p & -\mathbf{B}\mathbf{D}^{-1} \\ & \mathbf{I}_q \end{bmatrix}.$$

附录 B 代码修改说明

B.1 对原有文件的修改

Listing B.1: surfacemeshprocessing.*

```
1 // ===== surfacemeshprocessing.h =====
2 class SurfaceMeshProcessing : public QMainWindow
3 {
4     // ===== hw4: Actions =====
5     QAction* actBFFSolver;
6     // ...
7 };
8
9 // ===== surfacemeshprocessing.cpp =====
10 void SurfaceMeshProcessing::CreateActions(void)
11 {
12     // ...
13     // ===== hw3: Actions =====
14     actBFFSolver = new QAction("BFF Solver", this);
15     actBFFSolver->setStatusTip(tr("Boundary First Flattening of various free-boundaries
16     "));
17     connect(actBFFSolver, SIGNAL(triggered()), viewer, SLOT(BFFSolver()));
```

B.2 新增文件

文件 在相关代码存放在目录 Surface_Framework_Cmake/src/homeworks/BoundaryFirstFlattening/:

- (a) Util_BoundaryFirstFlattening.h
- (b) Util_BoundaryFirstFlattening.cpp

Listing B.2: Util_BoundaryFirstFlattening.h

```
1 #pragma once
2
3 #include <unordered_map>
4
5 #include <Eigen\.Dense>
6 #include <Eigen\Sparse>
7
8 #include " ../PolyMesh/include/PolyMesh/PolyMesh.h"
9
10 namespace tutte
11 {
12     enum class UVBoundaryType;
13 }
```

```

14
15 namespace bff
16 {
17     enum class FlattenType { FREE, DISK, FIXED };
18     class BFFSolver
19     {
20     public:
21         void Solve(acamcad::polymesh::PolyMesh* mesh, FlattenType type);
22         void SetExteriorAngle(size_t numBV, tutte::UVBoundaryType shape);
23
24     private:
25         // procedural methods
26         Eigen::VectorXd CalculateBoundaryLengths(const std::vector<acamcad::polymesh::MVert*>& bVertices) const;
27         std::tuple<Eigen::VectorXd, Eigen::VectorXd> CalculateDiscreteCurvatures() const;
28         Eigen::SparseMatrix<double> BuildLaplacian();
29
30         void InitParam();
31         void Flatten(const Eigen::VectorXd& givenTarget, bool givenScaleFactor = false)
32 ;
33         void Flatten(const Eigen::VectorXd& scaleFactor,
34                     const Eigen::VectorXd& exteriorAngle,
35                     bool conjugate = true);
36         void FlattenToDisk();
37         Eigen::MatrixXd ConstructBestFitCurve(const Eigen::VectorXd& l,
38                                              const Eigen::VectorXd& lstar,
39                                              const Eigen::VectorXd& ktilde) const;
40         std::tuple<Eigen::VectorXd, Eigen::VectorXd> ExtendCurve(const Eigen::MatrixXd&
41 gamma_tilde, bool freeBoundary);
42
43         // utility methods
44         void Util_RearrangeVertIndex();
45         size_t Util_VertIndex(acamcad::polymesh::MVert* vert) const { return
46 m_VertIndexMap.find(vert)->second; }
47         double Util_VertAngleDefect(acamcad::polymesh::MVert* vert) const;
48         double Util_VertExteriorAngle(acamcad::polymesh::MVert* vert) const;
49         double Util_HalfEdgeAngleCotan(acamcad::polymesh::MHalfedge* he) const;
50         Eigen::VectorXd Util_CvtDirichletToNeumann(const Eigen::VectorXd& phi,
51                                               const Eigen::VectorXd& g) const;
52         Eigen::VectorXd Util_CvtNeumannToDirichlet(const Eigen::VectorXd& phi,
53                                               const Eigen::VectorXd& h);
54
55         Eigen::VectorXd Util_VecVertConcat(const Eigen::VectorXd& A, const Eigen::VectorXd& B) const;
56         double Util_TargetBoundaryLengths(Eigen::VectorXd& lstar) const;
57         double Util_TargetDualBoundaryLengths(Eigen::VectorXd& lstar, Eigen::VectorXd&
58 ldual) const;
59         void Util_ConstrainUV();

```

```

56
57     private:
58         // data member
59         acamcad::polymesh::PolyMesh* m_Mesh = nullptr;
60         std::unordered_map<acamcad::polymesh::MVert*, size_t> m_VertIndexMap;
61         bool m_FlattenToDisk = false;
62
63         size_t m_NumV{ 0 };
64         size_t m_NumBV{ 0 };
65         size_t m_NumIV{ 0 };
66
67         std::vector<acamcad::polymesh::MVert*> m_BoundaryVertices;
68         Eigen::VectorXd m_BoundaryEdgeLengths;
69         Eigen::VectorXd m_TargetEdgeLengths;
70         Eigen::VectorXd m_BoundaryScaleFactor;
71         Eigen::VectorXd m_TargetExteriorAngle;
72
73         Eigen::VectorXd m_GaussianCurv, m_GeodesicCurv;
74         Eigen::SparseMatrix<double> m_LapMat, m_LapII, m_LapIB, m_LapBB;
75         Eigen::MatrixXd m_UVList;
76
77         Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>> m_LUSolver;
78     };
79 }
```

Listing B.3: Util_BoundaryFirstFlattening.cpp

```

1 #include "Util_BoundaryFirstFlattening.h"
2
3 #include "TutteEmbedding/Util_TutteEmbedding.h"
4
5 #include <Eigen/Sparse>
6 #include <Eigen/SparseCholesky>
7
8 namespace bff
9 {
10     void BFFSolver::Solve(acamcad::polymesh::PolyMesh* mesh, FlattenType type)
11     {
12         assert(!mesh->isEmpty(), "mesh is empty!");
13         m_Mesh = mesh;
14
15         InitParam();
16
17         switch (type)
18         {
19             case bff::FlattenType::FREE:
20             {
21                 m_BoundaryScaleFactor = Eigen::VectorXd::Zero(m_NumBV);
22                 Flatten(m_BoundaryScaleFactor, true);
23             }
24         }
25     }
26 }
```

```

23     }
24     break;
25   case bff::FlattenType::DISK:
26   {
27     m_FlattenToDisk = true;
28     FlattenToDisk();
29   }
30   break;
31   case bff::FlattenType::FIXED:
32   {
33     assert(m_TargetExteriorAngle.rows() == m_NumBV, "Require Input Exterior
Angle!");
34     // rearrange vert index
35     Eigen::VectorXd tmpBoundaryExteriorAngle = m_TargetExteriorAngle;
36     m_TargetExteriorAngle = Eigen::VectorXd::Zero(m_NumBV);
37     for (size_t bVertID = 0; bVertID < m_NumBV; ++bVertID)
38     {
39       size_t bID = Util_VertIndex(m_BoundaryVertices[bVertID]) - m_NumIV;
40       m_TargetExteriorAngle(bID) = tmpBoundaryExteriorAngle(bVertID);
41     }
42     Flatten(m_TargetExteriorAngle, false);
43   }
44   break;
45 }
46 }

47 void BFFSolver::SetExteriorAngle(size_t numBV, tutte::UVBoundaryType shape)
48 {
49   m_TargetExteriorAngle = Eigen::VectorXd::Zero(numBV);
50   auto targetExteriorBoundaryUVs = tutte::GetBoundaryUVs(numBV, shape);
51   for (size_t j = 0; j < numBV; ++j)
52   {
53     size_t i = (j + numBV - 1) % numBV;
54     size_t k = (j + 1) % numBV;

55     auto vecij = targetExteriorBoundaryUVs[j] - targetExteriorBoundaryUVs[i];
56     auto vecjk = targetExteriorBoundaryUVs[k] - targetExteriorBoundaryUVs[j];

57     m_TargetExteriorAngle(j) = acamcad::vectorAngle(acamcad::MVector3(vecij.x()
58 , vecij.y(), 0.0),
59                                         acamcad::MVector3(vecjk.x()
60 , vecjk.y(), 0.0));
61   }
62 }

63 Eigen::VectorXd BFFSolver::CalculateBoundaryLengths(const std::vector<acamcad::
64 polymesh::MVert*>& bVertices) const
65 {
66   size_t numBV = bVertices.size();

```

```

68     Eigen::VectorXd edges = Eigen::VectorXd(numBV);
69     edges.setZero();
70
71     for (size_t bVertID = 0; bVertID < numBV; ++bVertID)
72     {
73         auto* pBVert0 = bVertices[bVertID];
74         auto* pBVert1 = bVertices[(bVertID + 1) % m_NumBV];
75         size_t bID = Util_VertIndex(pBVert0) - m_NumIV;
76         edges(bID) = acamcad::distance(pBVert0->position(), pBVert1->position());
77     }
78     return edges;
79 }
80
81 std::tuple<Eigen::VectorXd, Eigen::VectorXd> BFFSolver::CalculateDiscreteCurvatures()
82 const
83 {
84     Eigen::VectorXd gaussianCurv = Eigen::VectorXd::Zero(m_NumIV);
85     Eigen::VectorXd geodesicCurv = Eigen::VectorXd::Zero(m_NumBV);
86
87     for (auto* pVert : m_Mesh->vertices())
88     {
89         size_t iVertID = Util_VertIndex(pVert);
90         if (!m_Mesh->isBoundary(pVert))
91             gaussianCurv(iVertID) = Util_VertAngleDefect(pVert);
92     }
93
94     for (auto* pBVert : m_BoundaryVertices)
95     {
96         size_t bID = Util_VertIndex(pBVert) - m_NumIV;
97         geodesicCurv(bID) = Util_VertExteriorAngle(pBVert);
98     }
99
100    return std::make_tuple(gaussianCurv, geodesicCurv);
101 }
102
103 Eigen::SparseMatrix<double> BFFSolver::BuildLaplacian()
104 {
105     std::vector<Eigen::Triplet<double>> ATriplets;
106     ATriplets.reserve(m_Mesh->halfEdges().size());
107
108     for (auto* pFace : m_Mesh->polyfaces())
109     {
110         acamcad::polymesh::MHalfedge* he = pFace->halfEdge();
111         do
112         {
113             size_t i = Util_VertIndex(he->fromVertex());
114             size_t j = Util_VertIndex(he->toVertex());
115             double w = 0.5 * Util_HalfEdgeAngleCotan(he);

```

```

116     ATriplets.emplace_back(i, i, w);
117     ATriplets.emplace_back(j, j, w);
118     ATriplets.emplace_back(i, j, -w);
119     ATriplets.emplace_back(j, i, -w);
120
121     he = he->next();
122 } while (he != pFace->halfEdge());
123 }
124
125 Eigen::SparseMatrix<double> A = Eigen::SparseMatrix<double>(m_NumV, m_NumV);
126 A.setFromTriplets(ATriplets.begin(), ATriplets.end());
127 A.makeCompressed();
128
129 auto AShift = Eigen::SparseMatrix<double>(m_NumV, m_NumV);
130 AShift.setIdentity();
131 A += AShift * std::numeric_limits<float>::epsilon();
132
133 return A;
134 }
135
136 Eigen::MatrixXd BFFSolver::ConstructBestFitCurve(const Eigen::VectorXd& l,
137                                                 const Eigen::VectorXd& lstar,
138                                                 const Eigen::VectorXd& ktilde)
139 {
140     Eigen::MatrixXd T_tilde = Eigen::MatrixXd::Zero(2, m_NumBV);
141     Eigen::VectorXd Ndiag = Eigen::VectorXd::Zero(m_NumBV);
142     double angle = 0.0;
143
144     for (size_t bVertID = 0; bVertID < m_NumBV; ++bVertID)
145     {
146         size_t bIDI = Util_VertIndex(m_BoundaryVertices[bVertID]) - m_NumIV;
147
148         angle += ktilde(bIDI);
149         T_tilde.col(bIDI) = Eigen::Vector2d(std::cos(angle), std::sin(angle));
150         Ndiag(bIDI) = 1.0 / l(bIDI);
151     }
152
153     Eigen::MatrixXd N = Ndiag.asDiagonal();
154     Eigen::VectorXd ltilde = lstar - N.inverse() * T_tilde.transpose() * (T_tilde *
155     N.inverse() * T_tilde.transpose()).inverse() * T_tilde * lstar;
156
157     Eigen::MatrixXd gamma_tilde = Eigen::MatrixXd::Zero(2, m_NumBV);
158
159     for (size_t bVertID = 1; bVertID < m_NumBV; ++bVertID)
160     {
161         size_t ID1 = Util_VertIndex(m_BoundaryVertices[bVertID]) - m_NumIV;
162         size_t ID0 = Util_VertIndex(m_BoundaryVertices[bVertID - 1]) - m_NumIV;

```

```

163         gamma_tilde.col(ID1) = gamma_tilde.col(ID0) + ltilde(ID0) * T_tilde.col(ID0
164     );
165
166     return gamma_tilde;
167 }
168
169 void BFFSolver::InitParam()
170 {
171     m_BoundaryVertices = m_Mesh->boundaryVertices();
172
173     m_NumV = m_Mesh->vertices().size();
174     m_NumBV = m_BoundaryVertices.size();
175     m_NumIV = m_NumV - m_NumBV;
176
177     Util_RearrangeVertIndex();
178
179     m_BoundaryEdgeLengths = CalculateBoundaryLengths(m_BoundaryVertices);
180
181     {
182         auto& [Gk, Ck] = CalculateDiscreteCurvatures();
183         m_GaussianCurv = Gk;
184         m_GeodesicCurv = Ck;
185     }
186
187     m_LapMat = BuildLaplacian();
188     m_LapII = m_LapMat.block(0, 0, m_NumIV, m_NumIV);
189     m_LapIB = m_LapMat.block(0, m_NumIV, m_NumIV, m_NumBV);
190     m_LapBB = m_LapMat.block(m_NumIV, m_NumIV, m_NumBV, m_NumBV);
191
192     m_LUSolver.compute(m_LapMat);
193 }
194
195 void BFFSolver::Flatten(const Eigen::VectorXd& givenTarget,
196                         bool givenScaleFactor)
197 {
198     if (givenScaleFactor)
199     {
200         Eigen::VectorXd dudn = Util_CvtDirichletToNeumann(-m_GaussianCurv,
201 givenTarget);
202         Eigen::VectorXd tExteriorAngle = m_GeodesicCurv - dudn;
203         double EPS = std::abs(tExteriorAngle.sum() - M_PI * 2);
204         std::cout << "Exterior Angle EPS = " << EPS << "\n";
205
206         Flatten(givenTarget, tExteriorAngle, true);
207     }
208     else
209     {
210         m_BoundaryScaleFactor = Util_CvtNeumannToDirichlet(-m_GaussianCurv,

```

```

    m_GeodesicCurv - givenTarget);
    Flatten(m_BoundaryScaleFactor, givenTarget, false);
}

Util_ConstrainUV();
}

void BFFSolver::Flatten(const Eigen::VectorXd& scaleFactor,
                       const Eigen::VectorXd& exteriorAngle,
                       bool freeBoundary)
{
    Eigen::VectorXd tmpBoundaryLengths = Eigen::VectorXd::Zero(m_NumBV);
    Util_TargetBoundaryLengths(tmpBoundaryLengths);

    Eigen::MatrixXd gamma_tilde = ConstructBestFitCurve(m_BoundaryEdgeLengths,
tmpBoundaryLengths, exteriorAngle);

    auto& [a, b] = ExtendCurve(gamma_tilde, freeBoundary);

    m_UVList = Eigen::MatrixXd(m_NumV, 2);
    const auto& vertices = m_Mesh->vertices();
    for (size_t vertID = 0; vertID < m_NumV; ++vertID)
    {
        size_t orderedVertID = Util_VertIndex(m_Mesh->vert(vertID));
        m_UVList(vertID, 0) = a(orderedVertID);
        m_UVList(vertID, 1) = -b(orderedVertID);
    }
}

void BFFSolver::FlattenToDisk()
{
    m_BoundaryScaleFactor = Eigen::VectorXd::Zero(m_NumBV);
    m_TargetExteriorAngle = Eigen::VectorXd::Zero(m_NumBV);

    m_TargetEdgeLengths = Eigen::VectorXd::Zero(m_NumBV);

    Eigen::VectorXd tDualLength = Eigen::VectorXd::Zero(m_NumBV);
    for (size_t repeat = 0; repeat < 20; ++repeat)
    {
        Util_TargetBoundaryLengths(m_TargetEdgeLengths);
        double L = Util_TargetDualBoundaryLengths(m_TargetEdgeLengths, tDualLength);
;

        for (auto* pBVert : m_BoundaryVertices)
        {
            size_t bID = Util_VertIndex(pBVert) - m_NumIV;
            m_TargetExteriorAngle(bID) = 2 * M_PI * tDualLength(bID) / L;
        }
        m_BoundaryScaleFactor = Util_CvtNeumannToDirichlet(-m_GaussianCurv,
}
}

```

```

256     m_GeodesicCurv - m_TargetExteriorAngle);
257 }
258
259     Flatten(m_TargetExteriorAngle, false);
260 }
261
262 void BFFSolver::Util_RearrangeVertIndex()
263 {
264     m_VertIndexMap.clear();
265     m_VertIndexMap.reserve(m_NumV);
266
267     size_t ivertCount = 0;
268
269     for (auto* pVert : m_Mesh->vertices())
270     {
271         if (m_Mesh->isBoundary(pVert)) continue;
272         m_VertIndexMap.insert({ pVert, ivertCount++ });
273     }
274
275     std::for_each(m_BoundaryVertices.begin(), m_BoundaryVertices.end(),
276 [ &](acamcad::polymesh::MVert* v) { m_VertIndexMap.insert({ v, ivertCount++ });
277 });
278 }
279
280 // Integrated gaussian curvature for inner vertices
281 double BFFSolver::Util_VertAngleDefect(acamcad::polymesh::MVert* vert) const
282 {
283     const auto& neighbors = m_Mesh->vertAdjacentVertices(vert);
284     size_t numNV = neighbors.size();
285
286     double angleSum{ 0.0 };
287     for (size_t vertID = 0; vertID < numNV; ++vertID)
288     {
289         size_t i = vertID;
290         size_t j = (i + 1) % numNV;
291         angleSum += std::abs(acamcad::vectorAngle(
292             neighbors[i]->position() - vert->position(),
293             neighbors[j]->position() - vert->position()));
294     }
295     return 2 * M_PI - angleSum;
296 }
297
298 double BFFSolver::Util_VertExteriorAngle(acamcad::polymesh::MVert* vert) const
299 {
300     const auto& neighbors = m_Mesh->vertAdjacentVertices(vert);
301     size_t numNV = neighbors.size();
302
303     double angleSum{ 0.0 };
304     for (size_t vertID = 0; vertID < numNV - 1; ++vertID)

```

```

303     {
304         size_t i = vertID;
305         size_t j = i + 1;
306         angleSum += std::abs(acamcad::vectorAngle(
307             neighbors[i]->position() - vert->position(),
308             neighbors[j]->position() - vert->position())));
309     }
310     return M_PI - angleSum;
311 }
312
313 double BFFSolver::Util_HalfEdgeAngleCotan(acamcad::polymesh::MHalfedge* he) const
314 {
315     auto* pVertA = he->fromVertex();
316     auto* pVertB = he->next()->fromVertex();
317     auto* pVertC = he->prev()->fromVertex();
318
319     auto vecU = pVertA->position() - pVertC->position();
320     auto vecV = pVertB->position() - pVertC->position();
321
322     return acamcad::dot(vecU, vecV) / acamcad::cross(vecU, vecV).norm();
323 }
324
325 Eigen::VectorXd BFFSolver::Util_CvtDirichletToNeumann(const Eigen::VectorXd& phi,
326                                         const Eigen::VectorXd& g)
327 const
328 {
329     Eigen::VectorXd b = phi - m_LapIB * g;
330     Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>> ldltSolver;
331     ldltSolver.compute(m_LapII);
332     Eigen::VectorXd a = ldltSolver.solve(b);
333
334     return - (m_LapIB.transpose() * a - m_LapBB * g);
335 }
336
337 Eigen::VectorXd BFFSolver::Util_CvtNeumannToDirichlet(const Eigen::VectorXd& phi,
338                                         const Eigen::VectorXd& h)
339 {
340     Eigen::VectorXd b = Util_VecVertConcat(phi, -h);
341     Eigen::VectorXd a = m_LUSolver.solve(b);
342
343     return a.segment(m_NumIV, m_NumBV);
344 }
345
346 Eigen::VectorXd BFFSolver::Util_VecVertConcat(const Eigen::VectorXd& A, const Eigen
347 ::VectorXd& B) const
348 {
349     size_t mA = A.size();
350     size_t mB = B.size();

```

```

350     Eigen::VectorXd concatVec = Eigen::VectorXd(mA + mB);
351     concatVec << A, B;
352
353     return concatVec;
354 }
355
356 double BFFSolver::Util_TargetBoundaryLengths(Eigen::VectorXd& tBoundaryLength)
357 const
358 {
359     tBoundaryLength = Eigen::VectorXd::Zero(m_NumBV);
360     for (size_t bVertID = 0; bVertID < m_NumBV; ++bVertID)
361     {
362         size_t bID0 = Util_VertIndex(m_BoundaryVertices[bVertID]) - m_NumIV;
363         size_t bID1 = Util_VertIndex(m_BoundaryVertices[(bVertID + 1) % m_NumBV]) -
364 m_NumIV;
365
366         double oBoundaryLength = m_BoundaryEdgeLengths[bID0];
367
368         double u0 = m_BoundaryScaleFactor(bID0);
369         double u1 = m_BoundaryScaleFactor(bID1);
370
371         tBoundaryLength(bID0) = std::exp(0.5 * (u0 + u1)) * oBoundaryLength;
372     }
373     return tBoundaryLength.sum();
374 }
375
376 double BFFSolver::Util_TargetDualBoundaryLengths(Eigen::VectorXd& tBoundaryLength,
377 Eigen::VectorXd& tDualLength) const
378 {
379     tDualLength = Eigen::VectorXd::Zero(m_NumBV);
380     for (size_t bVertID = 0; bVertID < m_NumBV; ++bVertID)
381     {
382         size_t bID0 = Util_VertIndex(m_BoundaryVertices[bVertID]) - m_NumIV;
383         size_t bID1 = Util_VertIndex(m_BoundaryVertices[(bVertID + 1) % m_NumBV]) -
384 m_NumIV;
385
386         tDualLength(bID1) = 0.5 * (tBoundaryLength(bID0) + tBoundaryLength(bID1));
387     }
388     return tDualLength.sum();
389 }
390
391 std::tuple<Eigen::VectorXd, Eigen::VectorXd> BFFSolver::ExtendCurve(const Eigen::MatrixXd& gamma_tilde, bool freeBoundary)
392 {
393     Eigen::VectorXd Re_gamma = gamma_tilde.row(0).transpose();
394     Eigen::VectorXd rightB = -m_LapIB * Re_gamma;
395
396     Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>> ldltSolver;
397     ldltSolver.compute(m_LapII);

```

```

394     Eigen::VectorXd aI = ldltSolver.solve(rightB);
395     Eigen::VectorXd a = Util_VecVertConcat(aI, Re_gamma);
396     Eigen::VectorXd b = Eigen::VectorXd::Zero(m_NumIV);
397
398     if (freeBoundary)
399     {
400         Eigen::VectorXd h = Eigen::VectorXd::Zero(m_NumV);
401
402         for (size_t bVertID = 0; bVertID < m_NumBV; ++bVertID)
403         {
404             size_t beforeID = (bVertID + m_NumBV - 1) % m_NumBV;
405             size_t afterID = (bVertID + 1) % m_NumBV;
406
407             auto* pBVert = m_BoundaryVertices[bVertID];
408             auto* pBeforeBVert = m_BoundaryVertices[beforeID];
409             auto* pAfterBVert = m_BoundaryVertices[afterID];
410
411             h(Util_VertIndex(pBVert)) = 0.5 * (a(Util_VertIndex(pBeforeBVert)) - a(
412               Util_VertIndex(pAfterBVert)));
413         }
414
415         ldltSolver.compute(m_LapMat);
416         b = ldltSolver.solve(h);
417     }
418     else
419     {
420         Eigen::VectorXd Im_gamma = gamma_tilde.row(1).transpose();
421         rightB = -m_LapIB * Im_gamma;
422         Eigen::VectorXd bI = ldltSolver.solve(rightB);
423         b = Util_VecVertConcat(bI, Im_gamma);
424     }
425
426     return std::make_tuple(a, b);
427 }
428
429 // constrain UV in range [0, 1]x[0, 1]
430 void BFFSolver::Util_ConstrainUV()
431 {
432     double xmin = std::numeric_limits<double>::max();
433     double xmax = -std::numeric_limits<double>::max();
434     double ymin = std::numeric_limits<double>::max();
435     double ymax = -std::numeric_limits<double>::max();
436
437     for (size_t vertID = 0; vertID < m_NumV; ++vertID)
438     {
439         if (!m_Mesh->isBoundary(m_Mesh->vert(vertID))) continue;
440         double x = m_UVList(vertID, 0);
441         double y = m_UVList(vertID, 1);

```

```
442     xmin = std::min(xmin, x);
443     xmax = std::max(xmax, x);
444     ymin = std::min(ymin, y);
445     ymax = std::max(ymax, y);
446 }
447
448 double midx = (xmin + xmax) * 0.5;
449 double midy = (ymin + ymax) * 0.5;
450
451 double maxScale = std::max(xmax - xmin, ymax - ymin);
452
453 for (size_t vertID = 0; vertID < m_NumV; ++vertID)
454 {
455     auto* vert = m_Mesh->vert(vertID);
456     double UVx = (m_UVList(vertID, 0) - midx) / maxScale;
457     double UVy = (m_UVList(vertID, 1) - midy) / maxScale;
458
459     vert->setTexture(UVy * 2.0f, -UVx * 2.0f, 0.0);
460 }
461 }
462 }
```

Bibliography

- [1] Breannan Smith, Fernando De Goes, and Theodore Kim. “Analytic Eigensystems for Isotropic Distortion Energies”. In: *ACM Trans. Graph.* 38.1 (Feb. 2019). issn: 0730-0301. doi: [10.1145/3241041](https://doi.org/10.1145/3241041). URL: <https://doi.org/10.1145/3241041>.
- [2] Bruno Lévy et al. “Least Squares Conformal Maps for Automatic Texture Atlas Generation”. In: *ACM Trans. Graph.* 21.3 (July 2002), pp. 362–371. issn: 0730-0301. doi: [10.1145/566654.566590](https://doi.org/10.1145/566654.566590). URL: <https://doi.org/10.1145/566654.566590>.
- [3] Rohan Sawhney and Keenan Crane. “Boundary First Flattening”. In: *ACM Trans. Graph.* 37.1 (Dec. 2017). issn: 0730-0301. doi: [10.1145/3132705](https://doi.org/10.1145/3132705). URL: <https://doi.org/10.1145/3132705>.
- [4] 钟玉泉. 复变函数论. Ed. by 丁鹤龄. 高等教育出版社, 2004.
- [5] Miao Jin et al. *Conformal Geometry*. en. Cham: Springer International Publishing, 2018. isbn: 978-3-319-75330-0 978-3-319-75332-4. doi: [10.1007/978-3-319-75332-4](https://doi.org/10.1007/978-3-319-75332-4). URL: <http://link.springer.com/10.1007/978-3-319-75332-4> (visited on 11/18/2022).