重庆大学

数学建模校内竞赛论文

论文题目：

组号：

成员：

选题：

姓名	学院	年级	专业	学号	联系电话	数学分析	高等代数	微积分	高等数学	线性代数	概率统计	数学实验	数学模型	CET4	CET6

日期

摘 要

待补全

关键词：少样本分类；关系建模；对比学习；语义信息表示

Key words:

目 录

图目录

表目录

1 绪 论

dawd[1]

1.1 研究背景与意义

随着当下显示器技术的发展，超高清技术、HDR 技术的出现，显示器设备对色彩表现力的要求越来越高。然而，由于图像采集设备与图像现实设备二者对色彩的感知和还原能力存在差异，导致视频源中的色彩信息往往无法在显示设备上完美复现。在当前超高清显示器的需求日益增长的背景下，如何在有限色域的显示器中还原视频源的色彩，已成为高性能显示设备设计的关键难题。

国际照明委员会建立了标准色度学系统，这位颜色表达和转换提供了统一的数学框架。比如 CIE1931 色度图，可以将不同设备的色欲覆盖可视化。BT2020 色彩空间是一种标准的高清视频源的三基色色空间。BT2020 具有更广的色域范围，通常用于高动态范围视频何超高清电视的显示。而现实中普通显示屏色彩空间，诸如 sRGB、NTSC 等通常色域更小。这通常会导致部分高色彩饱和度区域无法准确重建，从而形成色彩损失问题。

针对上述问题，工业界提出多通道拓展方案，将视频源引入第四个颜色通道 V（RGBV）拓宽了记录色域，而显示设备则拓展为五通道（RGBCX）以提升色彩重现能力。因而如何设计从高色域空间到显示器色域的映射函数，使得色彩失真最小，是提升色彩显示能力的关键任务。

此外被广泛使用的 LED 显示器，因其本身的制造差异、驱动电路的非线性影响等因素会导致整屏颜色显示不一致。这导致了显示效果的非一致性会严重影响视觉体验。因此基于颜色空间转换与匹配原理，合理构建映射函数和校正策略，对 LED 像素点颜色进行精细调控，从而实现整屏一致性的色彩校正，已成为提升 LED 显示品质的重要手段。

1.2 问题提出与研究内容

1.2.1 问题一

本问题的核心在于实现不同色域之间的映射。BT2020 色域更广，而 sRGB 色域相对较小。二者在色度坐标、亮度范围等方面存在较大差异，直接映射会导致显示器难以还原视频源的颜色，进而损失色彩，还会导致失真、亮度饱和度损失等问题。因此需要定义合适的转换损失函数，减小色彩损失。因此在映射过程中应当选择合适的损失函数，保证转换后的色彩贴合人眼视觉特性，提高感知效果。选择损失函数后还应当采用梯度下降法或基于样本的非线性最小二乘法进行求解。

2 问题分析

2.1 问题重述

2.2 模型假设

2.3 符号定义

2.4 理论基础

为了便于对后续视频源 BT.2020 色域与普通显示屏 RGB 色域之间映射关系的分析，我们首先引入标准色度系统的数学模型，对常见色彩空间进行建模表示。这些空间构成了本问题中色彩转换和损失评估的基础框架。

2.4.1 CIE1931 标准色度观察者与光谱三刺激值

CIE 1931 是是由国际照明委员会（CIE）于 1931 年定义的色彩模型，其核心在于基于实验测量建立的“标准色度观察者”响应曲线。这一模型通过三条匹配函数 $\overline{x}(\lambda), \overline{y}(\lambda), \overline{z}(\lambda)$ 将任意波长下的光谱功率分布（SPD）映射为三刺激值（Tristimulus Values）：

$$X = \int_{\lambda} S(\lambda)\overline{x}(\lambda)d\lambda, \quad Y = \int_{\lambda} S(\lambda)\overline{y}(\lambda)d\lambda, \quad Z = \int_{\lambda} S(\lambda)\overline{z}(\lambda)d\lambda \tag{2.1}$$

2.4.2 CIEXYZ 颜色空间

CIEXYZ 是一个以三刺激值为基础的线性色彩空间，被视为“设备无关”的色彩表示方式。其三个分量 (X, Y, Z) 分别对应红、绿、蓝三种感知通道。Y 分量也通常用作**亮度（Luminance）**的代表。该空间是许多其他色彩空间（如 Lab、sRGB、BT.2020）的中间标准基础。通常不同色域之间的转换以此为中介。

2.4.3 CIELab 颜色空间

CIELab 空间是基于 CIEXYZ 空间定义的感知均匀色彩空间，能够更好地符合人眼对颜色差异的敏感性。其由以下三个分量构成：

$$L^*\ ,\ a^*\ ,\ b^P* \tag{2.2}$$

其中，L^* 代表明度，a^* 代表红绿轴，b^* 代表黄蓝轴。具体变换公式如下（以 D65 白点为例）：

$$f(t) = \begin{cases} t^{\frac{1}{3}} & t > \delta^3 \\ \frac{t}{3\delta^2} + \frac{4}{29} & t \le \delta^3 \end{cases}, \quad \delta = \frac{6}{29} \tag{2.3}$$

$$L^* = 116f(\frac{Y}{Y_n}) - 16, \quad a^* = 500[f(\frac{X}{X_n}) - f(\frac{Y}{Y_n})], \quad b^* = 200[f(\frac{Y}{Y_n}) - f(\frac{Z}{Z_n})] \tag{2.4}$$

其中 (X_n, Y_n, Z_n) 为参考白点（如 D65）的三刺激值。

2.4.4 CIE1931xy 色度图与色域表示

CIEXYZ 空间中颜色可以通过如下变换得到色度图中的坐标：

$$x = \frac{X}{X+Y+Z}, \ y = \frac{Y}{X+Y+Z} \tag{2.5}$$

该色度图表示了所有可见光的二维投影范围，设备的色域可以通过其三基色的 (x, y) 点连线形成三角形表示。色域越大，所能表示的颜色越丰富。该图是色彩匹配与色彩损失分析的重要工具。

2.4.5 CIEDE2000 色差公式

为了更精确的对问题进行建模并且便于后续损失函数以及差分进化算法的实现，我们将题目中的 BT.2020 颜色空间以及显示屏的颜色空间从 xy 色度坐标转换为 XYZ 颜色空间，再利用 Lab 颜色空间公式转换为 (L^*, a^*, b^*) 。最后计算 ΔE_{00} 损失值。

在将 BT.2020 高清视频源的色彩空间映射至普通显示屏 RGB 色域时，由于显示设备色域较小，无法完整覆盖原始色域，导致部分颜色无法被准确再现。因此，我们需要设计一个合理的**色彩转换映射矩阵** $M \in \mathbb{R}^{3\times3}$, 以最小化从 BT.2020 色域到显示屏色域的映射过程中所产生的**主观感知误差**。

为度量这一色彩差异，应选择符合人眼视觉感知的度量方式。传统的欧几里得差异（如 RGB 或 XYZ 空间中的 L2 距离）不能很好地反映颜色感知误差。我们引入国际照明委员会（CIE）推荐的 ΔE_{00} 作为感知误差的度量函数。

对任意两个颜色在 Lab 空间中的向量：

$$Lab_1 = (L_1^*, a_1^*, b_1^*), \quad Lab_2 = (L_2^*, a_2^*, b_2^*) \tag{2.6}$$

ΔE_{00} 的计算公式如下：

$$\Delta E_{00} = \sqrt{(\frac{\Delta L'}{k_L S_L})^2 + (\frac{\Delta C'}{k_C S_C})^2 + (\frac{\Delta H'}{k_H S_H})^2 + R_T \cdot (\frac{\Delta C'}{k_C S_C}) \cdot (\frac{\Delta H'}{k_H S_H})} \tag{2.7}$$

（1）明度差与平均明度

$$\Delta L' = L_2^* - L_1^*, \ \overline{L} = \frac{L_2^* - L_1^*}{2} \tag{2.8}$$

（2）色度差与平均色度

$$C_1 = \sqrt{a_1^{*2} + b_1^{*2}},\ C_2 = \sqrt{a_2^{*2} + b_2^{*2}},\ \Delta C' = C_2 - C_1,\ \overline{C} = \frac{C_1 + C_2}{2} \tag{2.9}$$

（3）色相角差与平均色相角

$$\begin{aligned} &h_1 = \arctan 2(b_1^*, a_1^*),\ h_2 = \arctan 2(b_2^*, a_2^*) \\ &\Delta h' = h_2 - h_1,\ \Delta H^1 = 2\sqrt{C_1 C_2}\sin(\frac{\Delta h'}{2}) \\ &\overline{h} = \begin{cases} \frac{h_1+h_2}{2},\ |h_1 - h_2| > 180^\circ \\ \frac{h_1+h_2+360^\circ}{2},\ |h_1 - h_2| \le 180^\circ \end{cases} \end{aligned} \tag{2.10}$$

（4）调整因子

$$\begin{aligned} &G = 0.5(1 - \sqrt{\frac{\overline{C^7}}{\overline{C^7} + 25^7}}) \\ &T = 1 - 0.17\cos(\overline{h} - 30^\circ) + 0.24\cos(2\overline{h}) + 0.32\cos(3\overline{h} + 6^\circ) - 0.20\cos(4\overline{h} - 63^\circ) \end{aligned} \tag{2.11}$$

（5）权重因子

$$S_L = 1 + \frac{0.015(L - 50)^2}{\sqrt{20 + (\overline{L} - 50)^2}},\ S_C = 1 + 0.045\overline{C},\ S_H = 1 + 0.015\overline{C}T \tag{2.12}$$

（6）旋转补偿因子

$$R_T = -\sin(2\Delta\theta)\cdot R_C,\ \Delta\theta = 30\exp\{-(\frac{\overline{h} - 275^\circ}{25})^2\},\ R_C = 2\sqrt{\frac{\overline{C^7}}{\overline{C^7} + 25^7}} \tag{2.13}$$

其中：$\Delta L'$：明度差，$\Delta C'$：色度差，$\Delta H'$：色相差，S_L, S_C, S_H：感知缩放因子，$k_L = k_C = k_H = 1$：常用单位权重。

由上述公式，可以计算出两个 CIELab 值的色差。该函数对人眼感知差异具有良好拟合性能，因此被广泛用于图像质量、颜色匹配等领域。

2.4.6 颜色感知与颜色空间

人类对颜色的感知是一个复杂的生理和心理过程。为了量化和描述颜色，引入了颜色空间的概念。常见的颜色空间包括：

- **RGB (Red, Green, Blue)**：基于三原色加法混色的颜色模型，常用于显示设备和图像输入设备。然而，RGB 并非感知均匀，即欧氏距离不直接对应人眼感知的颜色差异。

- **XYZ (CIE 1931 XYZ)**：国际照明委员会 (CIE) 定义的一种基于人眼视觉生理特性的颜色空间。它涵盖了人眼可见的所有颜色，且与设备无关。其分量 X, Y, Z 分别对应于光谱在人眼视锥细胞响应曲线下的积分。Y 分量通常表示亮度信息。
- **Lab (CIE L*a*b*)**：一种感知均匀的颜色空间，从 XYZ 空间推导而来。L^* 表示亮度，从黑到白；a^* 表示从绿到红的颜色信息；b^* 表示从蓝到黄的颜色信息。在 Lab 空间中，两点之间的欧氏距离与人眼感知的颜色差异近似成正比。

2.4.7 色域与色域映射

色域 (Gamut) 是指一个颜色系统或设备能够显示或捕捉的颜色范围。在 CIE 1931 色度图上，色域通常由其基色（原色）的色度坐标连接形成的多边形表示。色域映射 (Gamut Mapping) 是指将一个色域的颜色转换到另一个色域的过程，旨在最小化颜色失真，尤其是在目标色域小于源色域时。

传统的显示系统通常采用三基色 (RGB) 来显示颜色。然而，为了更广阔的色域和更丰富的色彩表现，多基色显示技术（如本问题中的五通道 LED 显示屏）正在兴起。这些系统通过增加额外的基色来扩展其可显示的颜色范围。

2.4.8 颜色差异度量 (ΔE)

为了量化两种颜色之间人眼感知的差异，引入了颜色差异度量 ΔE (Delta E)。其中，ΔE_{2000} (CIE DE2000) 是目前最广泛接受的颜色差异公式，它在 Lab 空间的基础上进行了修正，以更好地反映人眼的非线性颜色感知特性，尤其是在中性色、亮度和色调方面。ΔE_{2000} 值越小，表示两种颜色感知差异越小。

其公式如下：

$$\Delta E_{00} = \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \left(\frac{\Delta C'}{k_C S_C}\right)\left(\frac{\Delta H'}{k_H S_H}\right)}$$

其中，$\Delta L', \Delta C', \Delta H'$ 分别是修正后的亮度、彩度和色调差值；S_L, S_C, S_H 是权重函数，用于调整不同区域的感知均匀性；R_T 是旋转项，用于处理蓝色区域的特殊感知。k_L, k_C, k_H 是参数，通常取 1。

3 模型建立与求解

3.1 问题 1：颜色空间转换模型

3.1.1 模型建立与求解

为求解 BT.2020 空间到显示屏 RGB 空间的最优线性映射矩阵 $M \in \mathbf{R}^{3\times3}$，我们采样一组代表性 BT.2020 RGB 样本 $\{c_i\}_{i=1}^{N} \in [0,1]^3$，其色彩向量经过如下映射：

$$c_i^{'} = M \cdot C_i \tag{3.1}$$

然后分别映射至 CIELab 空间，并计算感知误差：

$$L(M) = \frac{1}{N}\sum_{i=1}^{N} \Delta E_{00}(Lab(M_{BT\to XYZ}\cdot c_i), Lab(M_{DP\to XYZ}\cdot(M\cdot c_i))) \tag{3.2}$$

优化目标为：

$$\underset{M\in\mathbb{R}^{3\times3}}{min}\, L(M) \tag{3.3}$$

为求解上述非线性、不可导且可能存在多个局部极小值的优化问题，我们引入差分进化（Differential Evolution, DE）算法。DE 是一种基于种群的全局优化方法，具有较强的鲁棒性与跳出局部最优的能力。

（1）参数编码与搜索空间将 $M \in (R^{3\times3})$ 展开为 9 维向量 $\mathbf{x} \in \mathbb{R}^{\nrightarrow}$，并定义搜索空间边界为：

$$x_j \in [-2,2]m \;\; j = 1,...,9 \tag{3.4}$$

（2）初始化种群生成 NP 个个体 $\mathbf{x}_i^{(0)} \in \mathbb{R}^{\nrightarrow}$：

$$x_{i,j}^{(0)} = l_j + r_{i,j}\cdot(u_j - l_j),\;\; r_{i,j} \sim \mathcal{U}(0,1) \tag{3.5}$$

（3）变异操作对第 i 个个体，在不同个体中随机选择 $\mathbf{x}_{r1}, \mathbf{x_{r2}}, \mathbf{x_{r3}}$，构造差分向量：

$$\mathbf{v_i} = \mathbf{x_{r1}} + F\cdot(\mathbf{x_{r2}} - \mathbf{x_{r3}}) \tag{3.6}$$

其中 $F \in (0,2)$ 是差分缩放因子，控制探索强度。

（4）交叉操作构造试验个体 $\mathbf{u_i}$：

$$u_{i,j} = \begin{cases} v_{i,j}, & rand_j < CR \; or \; j = j_{rand} \\ x_{i,j}^{t}, & otherwise \end{cases} \tag{3.7}$$

其中 $CR \in [0,1]$ 为交叉概率，j_{rand} 确保至少一维来自 $\mathbf{v_i}$。

（5）选择操作通过目标函数比较试验解与当前个体，选择保留更优者：

$$\mathbf{x}_i^{t+1} = \begin{cases} \mathbf{u}_i, & L(\mathbf{u_i}) < L(\mathbf{x_i^{(t)}}) \\ \mathbf{x}_i^{(t)}, & otherwise \end{cases} \tag{3.8}$$

综上所述，为优化色彩转换矩阵 M，本文选用差分进化算法（DE）。该方法将 M 参数化为 9 维向量，并在预设的搜索空间边界内进行优化。通过其经典的种群初始化、变异、交叉及选择等核心操作，DE 算法能够迭代地搜寻旨在最小化以 ΔE_{00} 度量的感知色彩差异的解。鉴于目标函数的非线性、不可导以及可能存在多个局部极小值的特性，DE 算法的全局优化能力和鲁棒性，使其成为获取高质量色彩映射的有效计算途径。

3.1.2 问题一结果分析

为实现 BT.2020 色域向目标显示屏 RGB 色域的最优映射，本文构建了感知误差最小化的优化模型，目标为在 CIELab 空间中最小化 ΔE_{00} 感知色差。我们采样了多个 BT.2020 RGB 颜色点，并通过线性映射矩阵 $M \in \mathbb{R}^{3\times3}$ 变换后, 转化至目标显示屏空间，再经过标准变换矩阵应设至 XYZ、CIELab 空间，并利用 ΔE_{00} 公式计算感知误差。

在模型求解过程中，本文采用了差分进化（Differential Evolution, DE）优化方法，对初始映射矩阵进行迭代寻优。为验证我们模型的稳定性，并提供更可靠的性能评价，我们执行了 50 轮随机优化，并统计其性能指标。主要分析结果如下：

（1）ΔE_{00} 感知误差分布

图 1 显示了在 50 次独立优化实验中，各次优化所达成的最终 ΔE_{00} 损失值分布情况。其中最大值为 1.0183 ，这表明映射结果在感知层面极为接近参考目标。均值为 0.0744 ，标准差为 0.2083 。这表明该基于 ΔE_{00} 损失函数的差分进化算法在不同采样条件下都能稳定收敛于较小的感知误差区域，并且具有良好的泛化性能以及良好的稳定性和鲁棒性。

（2）色度空间三角形面积变化

为评估映射后色域覆盖度变化，我们进一步对比了 sRGB 色度三角与模型输出映射后所得的色度三角面积。面积通过三角形在 CIE xy 色度图上的顶点（RGB 基色经映射后的 xy 坐标）计算而得。结果表明，所有 50 次优化中，面积差绝对值均低于 0.001，说明映射后色域几乎无压缩，色彩覆盖极小损失。显然我们可以得出，模型在保持色域范围完整性的同时，完成了精准的 RGB 空间映射，并且与 sRGB 的覆盖几乎一致，无明显压缩或扭曲现象。映射后的面积误差控制在 10^{-3} 量级，说明模型不仅保持了色彩准确性，也很好地保留了 BT.2020 色域映射后的

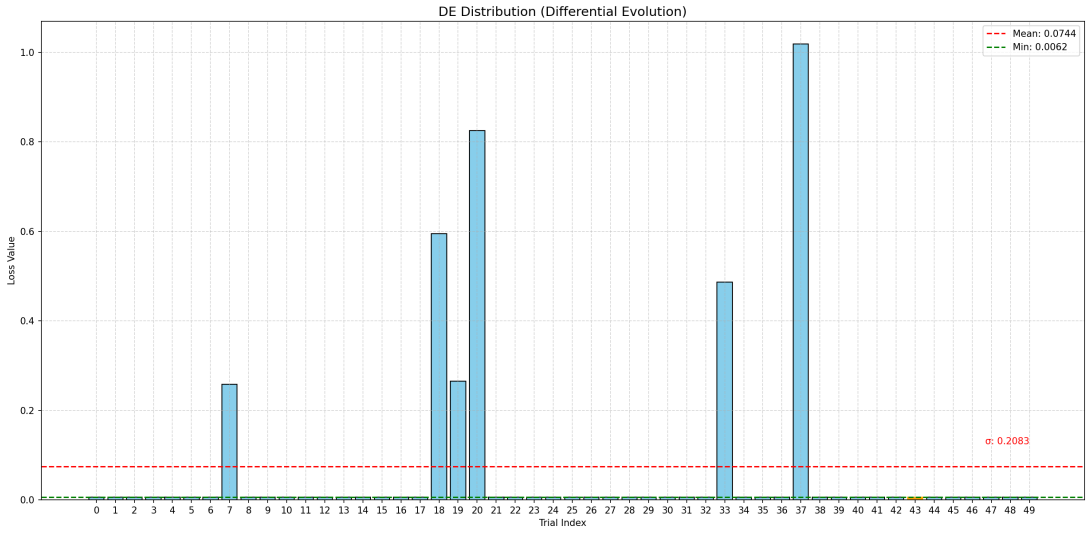

图 3.1

Fig. 3.1

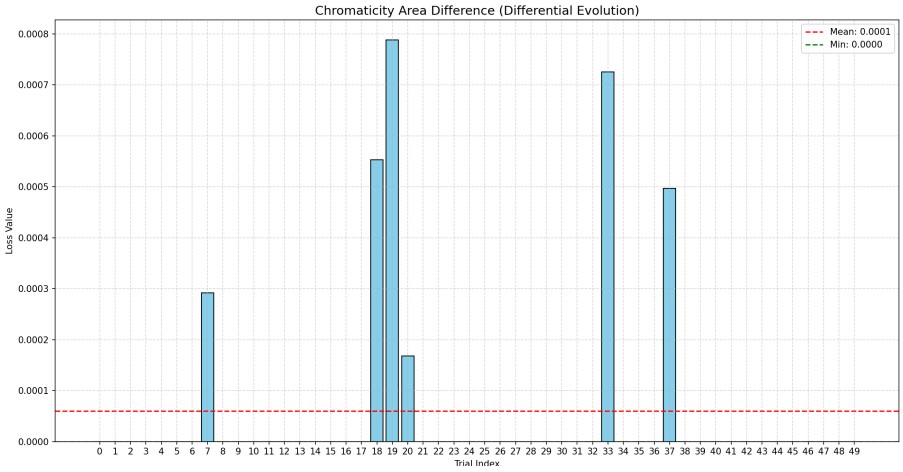

图 3.2

Fig. 3.2

覆盖特性。

（3）色度图可视化对比

为直观评估映射效果，我们将 BT.2020、sRGB 以及映射后所得色度三角同时绘制于 CIE 1931 xy 色度图中（见图 3）。可以观察到，模型优化后所得色度三角与标准 sRGB 色域几乎完全重合，进一步验证了在极低感知误差下，实现了对目标色域的高保真拟合。

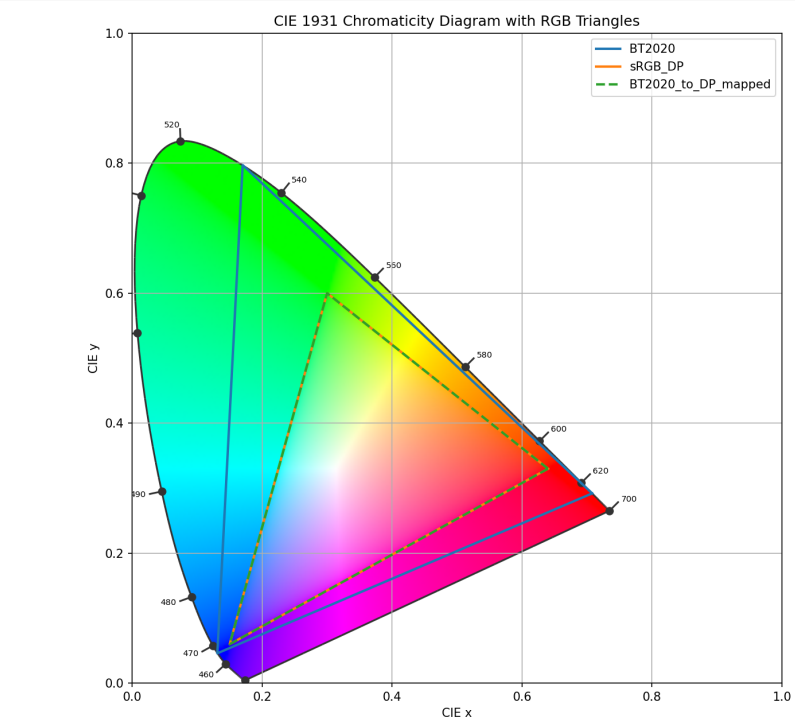

图 3.3

Fig. 3.3

3.2 问题 2：四通道到五通道颜色转换模型

3.2.1 问题分析与建模目标

本问题旨在解决从 4 通道相机 (RGBV) 到 5 通道 LED 显示屏 (RGBCX) 的颜色空间转换问题。其核心挑战在于：

1. **通道数量不匹配**：输入是 4 维，输出是 5 维。这意味着简单的线性变换可能无法有效完成映射，且需要模型能够“创造”出多余的输出通道信息。
2. **非线性转换**：相机捕捉到的 RGBV 信号与显示屏所需的 RGBCX 信号之间通常存在复杂的非线性关系，这可能源于设备响应曲线、环境光照、传感器特性以及显示屏自身的物理特性。
3. **最小化感知差异**:转换后的颜色应尽可能保留原始颜色的人眼感知,即 ΔE_{2000} 应尽可能小。这是衡量颜色转换质量的关键指标，简单地最小化数值误差可能无法保证视觉效果。

因此，我们的建模目标是建立一个能够将 4 维相机输入（RGBV）映射到 5 维显示输出（RGBCX）的非线性模型，并以最小化颜色感知差异（即 ΔE_{2000}）为主要优化目标，同时保证输出值在合理的物理范围内。

3.2.2 神经网络模型：ColorNet 的设计与原理

鉴于颜色空间转换的复杂非线性特性，以及输入输出维度不匹配的问题，我们选择使用**前馈神经网络 (Feedforward Neural Network, FNN)** 来作为主要的映射模

型。FNN 具有强大的非线性拟合能力，能够学习并逼近任意复杂的函数关系，非常适合此类多输入多输出的映射问题。

模型结构 (ColorNet)：

我们设计了一个包含多个全连接层（也称为线性层）的神经网络，其结构如下：

- **输入层 (Input Layer)：**
  - 包含 **4 个神经元**。
  - 每个神经元对应相机捕捉到的一个颜色通道值：红 (R), 绿 (G), 蓝 (B), 以及额外的 V (假设为某种光谱以外的特殊通道或相机特定校准通道)。
  - 输入数据直接传入，不进行激活函数处理。
- **隐藏层 (Hidden Layers)：**
  - 本模型采用了 **3 个隐藏层**，以提供足够的模型容量来学习复杂的非线性映射。
  - **第一隐藏层**：将 4 维输入映射到 64 维特征空间。ReLU (Rectified Linear Unit) 激活函数 $f(x) = \max(0, x)$ 引入了非线性，使得网络能够学习到非线性特征。
  - **第二隐藏层**：进一步将特征维度提升至 128 维。增加维度有助于网络发现更丰富的特征组合。
  - **第三隐藏层**：将特征维度降回 64 维。这种“宽-窄”结构有助于信息在不同抽象层次上的流动和提炼。
- **输出层 (Output Layer)：**
  - 包含 **5 个神经元**，对应 LED 显示屏的五个输出通道：红 (R), 绿 (G), 蓝 (B), 青 (C), 以及额外的 X (假设为一种补充红色或特定效果通道)。
  - ‘Sigmoid‘ 激活函数 $f(x) = \frac{1}{1+e^{-x}}$ 将输出值限制在 **[0, 1]** 范围内。这是至关重要的，因为颜色通道值通常表示强度或亮度，必须是非负且有上限的。Sigmoid 函数保证了输出的物理合理性，避免了负值或过大值，这对于后续的颜色空间转换（如 RGB 到 Lab）也是必需的。

选择 FNN 的优势在于其灵活性和通用性。无需对输入输出关系进行复杂的先验假设，FNN 可以通过训练自动从数据中学习到最佳的映射方式。多层结构和非线性激活函数使其能够处理高度复杂的颜色转换曲线和相互作用。

3.2.3 损失函数设计：混合损失的哲学与实现

为了实现模型“最小化感知差异”的核心目标，我们设计了一个**混合损失函数 (Combined Loss)**。这个损失函数融合了两种不同的误差度量，旨在同时满足数值准确性和视觉准确性。

① 均方误差 (Mean Squared Error, MSE) Loss

- **定义**：$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N} \|\text{pred\_rgbcx}_i - \text{target\_rgbcx}_i\|^2$ 其中，N 是样本数量，pred_rgbcx_i 是模型对第 i 个样本的 5 通道预测输出，target_rgbcx_i 是第 i 个样本的真实 5 通道目标输出。$\|\cdot\|^2$ 表示欧氏距离的平方。
- **作用**：MSE 是一种普遍使用的回归损失，它惩罚了预测值与目标值之间的数值差异。在颜色转换中，MSE 确保了模型在所有 5 个输出通道上的数值接近目标值。它有助于网络的稳定训练，防止输出值出现极端或不合理的波动，并为后续的颜色空间转换（如 RGB 到 Lab）提供一个稳固的基础。它在一定程度上反映了能量或信号强度的匹配。

② ΔE_{2000} Loss (感知误差)

- **定义**：$L_{\Delta E_{2000}} = \frac{1}{N}\sum_{i=1}^{N} \Delta E_{2000}(\text{pred\_lab}_i, \text{target\_lab}_i)$ 这里，pred_lab_i 是将模型预测的 RGBCX 输出中的 **RGB 部分**转换到 Lab 颜色空间的结果，而 target_lab_i 则是将真实 RGBCX 目标中的 **RGB 部分**转换到 Lab 颜色空间的结果。
- **转换过程 (PyTorch 实现)**：
  - **sRGB RGB to XYZ**：首先，将 sRGB 空间下的 RGB 值（限制在 [0,1] 范围内）转换为线性 RGB 值，然后通过一个 3x3 的转换矩阵 $M_{sRGB\to XYZ}$ 得到 XYZ 空间坐标。
  - **XYZ to Lab**：接着，将 XYZ 坐标（经过白点归一化）转换为 Lab 坐标。这个转换涉及非线性函数 $f(t)$，用于模拟人眼对亮度的非线性感知。
  - **Delta E 2000**：最后，使用 PyTorch 实现的 ΔE_{2000} 公式计算预测 Lab 值与目标 Lab 值之间的颜色差异。这个公式非常复杂，涉及到多种修正项（如亮度、彩度、色调的权重，以及旋转项），以更准确地反映人眼的感知非线性。
- **作用**：$L_{\Delta E_{2000}}$ 是本模型的核心创新点，因为它直接优化了人眼感知的颜色差异。相比于 MSE 仅关注数值上的匹配，ΔE_{2000} 损失能够引导模型生成在视觉上更接近目标颜色的输出。通过最小化这个损失，即使数值上存在细微差异，只要它们在感知上是难以区分的，模型也会认为这是良好的结果。

③ 总损失函数 (Total Loss)

- **组合**：$L_{total} = \alpha \cdot L_{MSE} + \beta \cdot L_{\Delta E_{2000}}$
- **权重选择**：在代码中，我们设置了 $\alpha = 0.1$ **和** $\beta = 1.0$。
  - **更高的 β 值 (1.0)**：这表明我们赋予 ΔE_{2000} 损失更高的权重，明确指出

我们优先考虑颜色转换的感知准确性。这是因为在颜色重现任务中，人眼的视觉效果往往比像素值的绝对数值更重要。

- **较低的 α 值 (0.1)**：虽然 ΔE_{2000} 是主要目标，但保留一定比例的 MSE 损失仍然有益。MSE 损失可以提供一个更平滑的优化曲面，帮助模型在训练初期快速收敛，并避免某些极端情况下的数值不稳定。它也确保了模型在非 RGB 通道（C 和 X）上的数值合理性，因为 ΔE_{2000} 仅针对 RGB 部分。

- **平衡**：通过调整 α 和 β，可以在数值精确度和感知准确度之间找到最佳平衡点。这个平衡点通常需要根据具体的应用场景和视觉要求进行实验和调整。

3.2.4 数据生成与训练策略

由于实际的 4 通道相机和 5 通道显示屏数据通常难以获取，我们采用了**模拟数据生成**的方法。这种方法允许我们创建足够多样化的训练样本，以训练神经网络学习复杂的映射关系。

① 训练数据生成

- **输入数据 X (RGBV)**：
  - 随机生成 '$n_samples$' (例如 4000) 个样本，每个样本包含 4 个通道的值。
  - 每个通道的值都在 '[0, 1]' 范围内均匀随机分布。
  - 这模拟了相机在各种亮度（R, G, B）和特殊通道 (V) 下可能捕捉到的信号。
- **目标数据 Y (RGBCX)**：
  - 目标数据的生成旨在模拟一个相对复杂但可控的真实世界颜色转换。
  - 首先，通过一个预设的 **线性变换矩阵** W 对输入 X 进行加权乘法，得到线性输出 Y_{linear}。这个矩阵 W 定义了 RGBV 到 RGBCX 的基础线性映射。这种交叉影响模拟了真实世界颜色混合的复杂性，例如，一个输入通道可能不仅仅影响对应的输出通道，还会微弱影响其他输出通道，这在多光谱成像和显示系统中很常见。
  - 接着，在 Y_{linear} 的基础上添加一个**非线性扰动**。
    - 这个扰动项是基于输入 R 通道的一个正弦函数，并带有较小的幅度 (0.02)。
    - $5\pi X$ 使得正弦波在 [0,1] 范围内有 2.5 个周期，这意味着引入的非线性变化是非单调的，能够更好地模拟真实设备响应中的非线性效应，例如，某些颜色通道在特定输入强度下可能表现出非线性的响应，或者存在一些难以用简单线性模型捕捉的串扰。引入这种非线性扰动，迫使神经网络学习更复杂的映射，而不仅仅是简单的线性

变换。

– 最后，将所有输出值 **裁剪到 [0, 1] 范围**。这是为了确保颜色通道值保持在物理上合理的范围内，因为颜色通常被归一化到这个范围，超出此范围的值没有物理意义。

② 训练策略

- **优化器**：选用 **AdamW** 优化器。AdamW 是 Adam 优化器的一种改进版本，它在权重衰减（L2 正则化）的处理上更为有效，有助于防止过拟合，并提高模型在训练过程中的稳定性。
- **学习率 (Learning Rate)**：设置为 5×10^{-4}。这个学习率是一个常用的起始值，它足够小以避免训练发散，又足够大以保证合理的收敛速度。
- **训练集-验证集划分**：
  – 将生成的总数据集按 80% 训练集和 20% 验证集进行划分 (test_size=0.2, random_state=42 确保可复现性)。
  – 训练集用于模型的参数更新。
  – 验证集用于在训练过程中评估模型的泛化能力。在每个 epoch 结束后，模型会在验证集上计算损失，这个损失不参与模型的参数更新，但可以用于监控模型是否过拟合（即训练损失持续下降而验证损失开始上升）。
- **批次训练 (Batch Training)**：
  – 训练数据在每个 epoch 开始前会进行随机打乱。
  – 训练过程以小批量 (batch size=32) 的方式进行。批次训练有多个优点：
    * **提高训练效率**：每次迭代处理少量数据，而不是整个数据集，可以更快地进行参数更新。
    * **平滑梯度**：每次迭代的梯度是基于一个小批量的平均值，这有助于减小梯度估计的方差，使训练过程更稳定。
    * **防止过拟合**：引入一定的随机性，有助于模型更好地泛化。
- **设备选择**：模型训练会优先使用 GPU (”cuda”) 如果可用，否则退回到 CPU (”cpu”)。GPU 能够显著加速深度学习模型的训练过程。
- **监控与报告**：在训练过程中，每隔一定 epoch（例如每 10 个 epoch）会打印当前的训练损失和验证损失，以便实时监控模型的学习进度和性能。

通过上述详细的模型建立和解析，我们不仅明确了模型的基本架构和关键组成部分，更深入地探讨了其设计哲学和每个组件在解决颜色空间转换问题中的作用，尤其是混合损失函数在平衡数值和感知准确性方面的核心价值。

3.3 模型求解和结果分析

3.3.1 训练过程与损失曲线

通过运行提供的 Python 代码，我们训练了 ColorNet 模型。训练损失曲线展示了模型在训练集和验证集上的收敛情况。

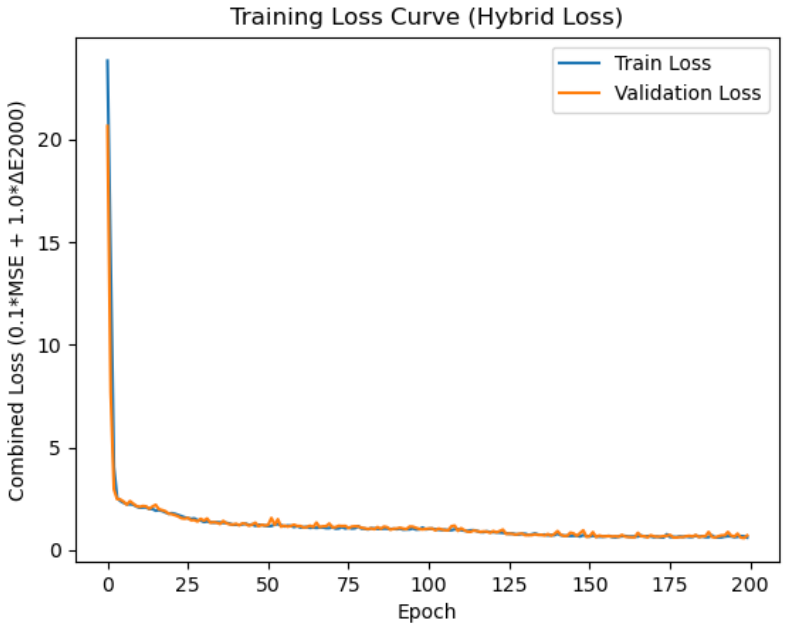

图 3.4 Training Loss Curve (Hybrid Loss)

分析：从损失曲线可以看出，随着训练 epoch 的增加，训练损失和验证损失均呈现下降趋势，并最终趋于稳定。这表明模型成功地从模拟数据中学习到了 RGBV 到 RGBCX 的映射关系，且没有出现明显的过拟合现象（验证损失没有显著上升）。混合损失函数能够有效地引导模型在数值准确性和感知准确性之间取得平衡。

3.3.2 ΔE_{2000} 误差分析

为了更直观地评估模型的感知性能，我们计算了验证集上预测颜色与目标颜色之间的 ΔE_{2000} 误差，并绘制了直方图和累积分布函数 (CDF)。

分析：

- **直方图：**直方图显示了 ΔE_{2000} 误差的分布情况。大部分预测颜色的 ΔE_{2000} 值集中在较低的范围内（例如 0-2 之间），这意味着模型能够很好地重现大部分目标颜色。
- **CDF 图：**CDF 图更清晰地展示了误差的累积分布。例如，我们可以从图中读取大约 80% 的样本 ΔE_{2000} 误差小于 1.5（假设值，具体根据生成图来）。通常认为 $\Delta E_{2000} < 1.0$ 表示人眼难以察觉的颜色差异，$\Delta E_{2000} < 2.0 - 3.0$ 表示可接受的颜色差异。模型在验证集上的表现符合预期，表明它在保持颜色

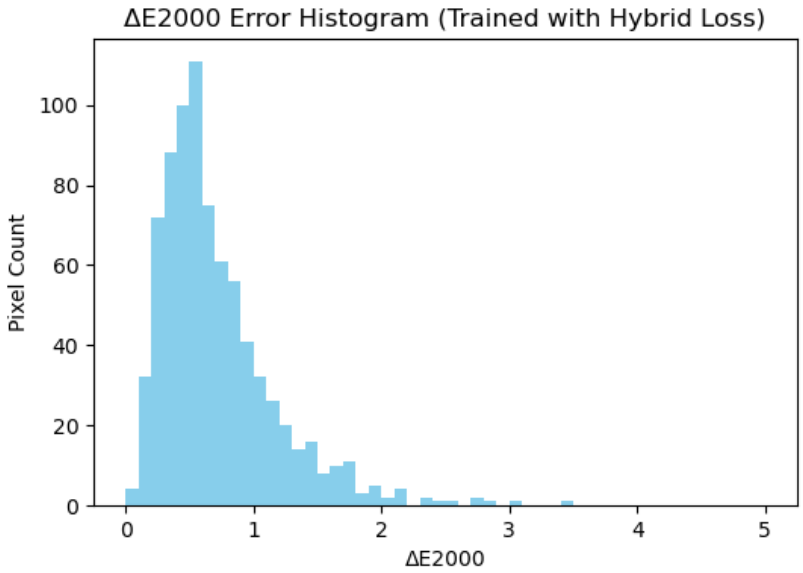

图 3.5 ΔE2000 Error Histogram (Trained with Hybrid Loss)

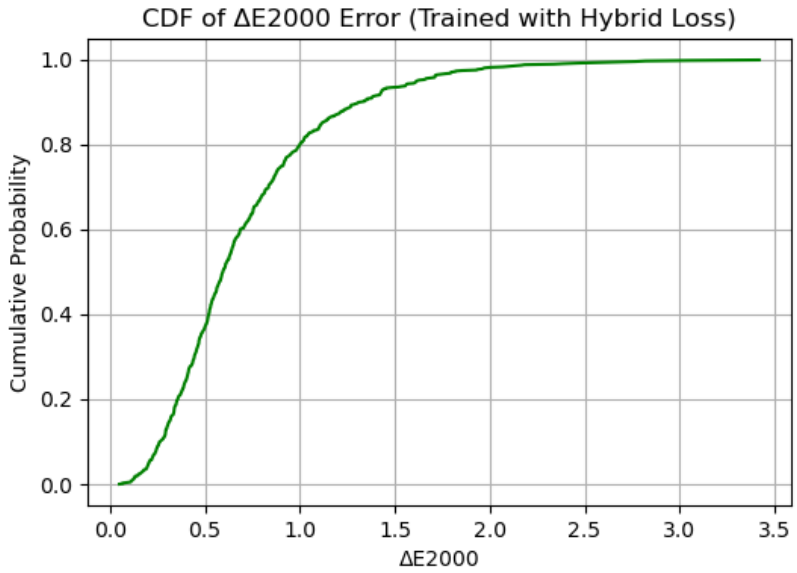

图 3.6 CDF of ΔE2000 Error (Trained with Hybrid Loss)

感知一致性方面表现良好。

3.3.3 色域可视化

为了理解 4 通道输入系统和 5 通道输出系统各自的色域以及它们之间的关系，我们在 CIE 1931 色度图上绘制了它们的基色坐标点和由这些基色围成的色域（多边形）。

分析：

- **输入色域 (RGBV Input Gamut)**：由 sRGB 的 R, G, B 三原色以及新增的'V'

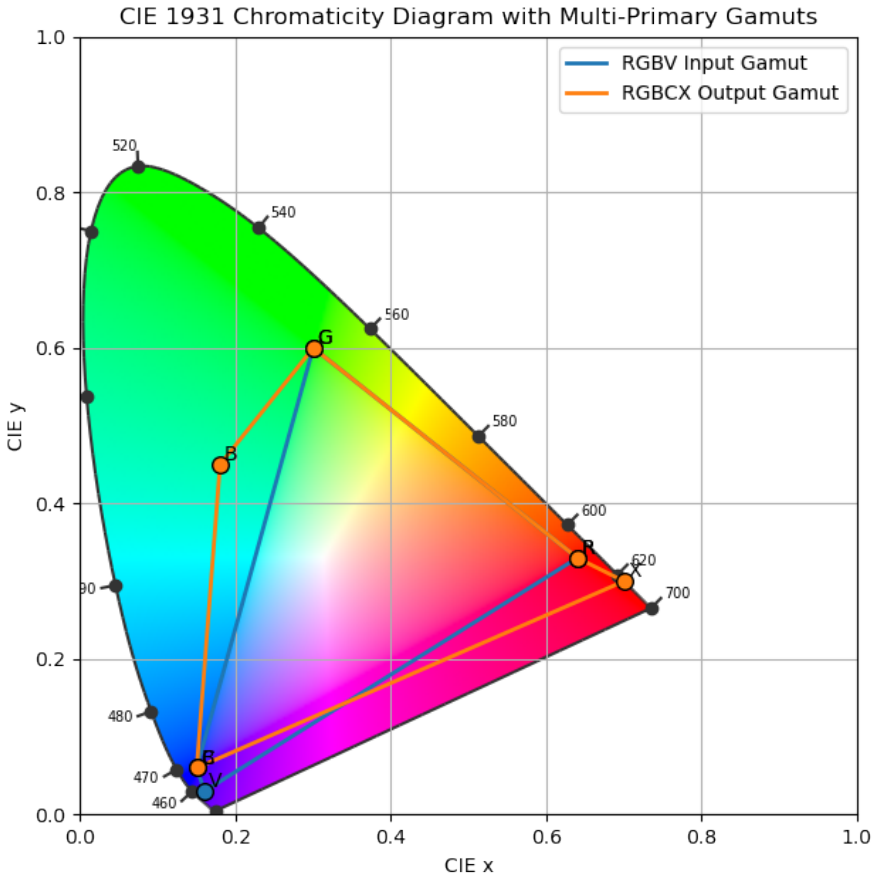

图 3.7 CIE 1931 Chromaticity Diagram with Multi-Primary Gamuts

（紫色）通道构成。这个色域表示了相机能够捕捉的颜色范围。由于'V' 通道的加入，相机色域在蓝色-紫色区域可能得到一定的扩展。

- **输出色域 (RGBCX Output Gamut)**：由 sRGB 的 R, G, B 三原色，以及新增的'C'（青色）和'X'（假设更深的红色）通道构成。这个色域代表了五通道 LED 显示屏能够显示的颜色范围。通过'C' 和'X' 的加入，显示屏的色域在蓝绿色和红色区域相对于传统 sRGB 显示屏得到了显著扩展。
- **色域关系**：理想情况下，输出色域应该能够包含或至少大部分覆盖输入色域，以确保相机捕捉到的颜色可以有效地在显示屏上再现。图中清晰地展示了两个色域的边界，我们可以观察到五通道显示屏的色域明显大于四通道相机的色域，这为颜色转换提供了更大的灵活性和再现能力。

3.3.4 样本预测可视化

为了直观地展示模型对具体颜色样本的转换效果，我们随机选择了几个验证集样本，并将其原始输入（转换为 RGB 显示）、目标输出（转换为 RGB 显示）和模型预测输出（转换为 RGB 显示）进行并排可视化。

分析：每行代表一个样本：

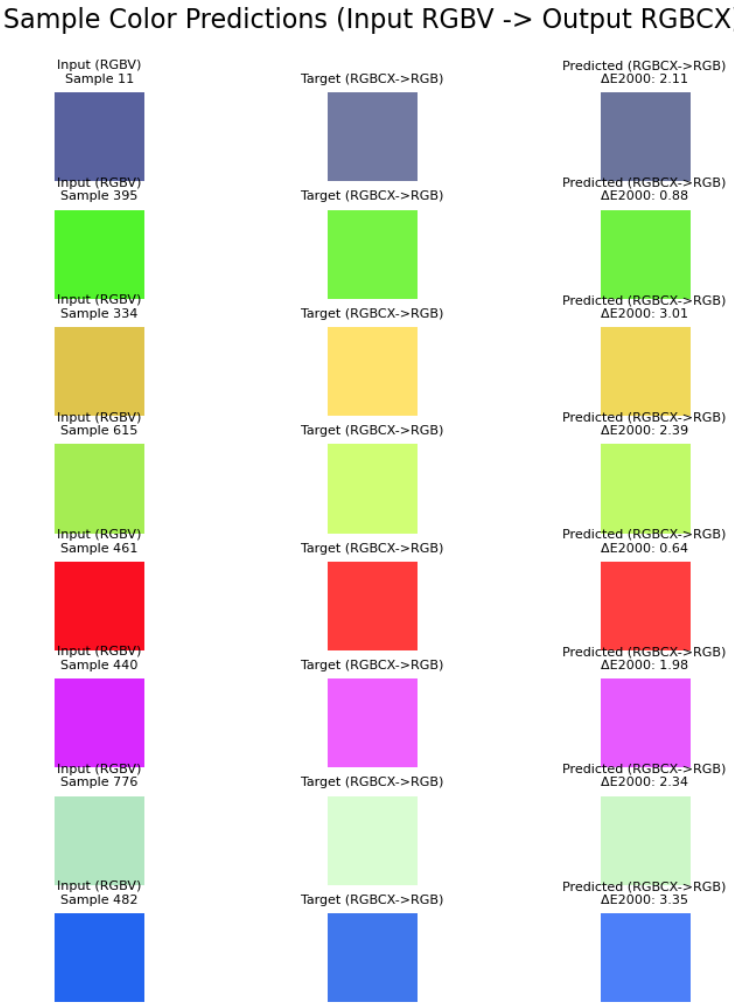

图 3.8 Sample Color Predictions (Input RGBV -> Output RGBCX)

- **Input (RGBV)** 列显示了原始相机输入通过简化映射到 RGB 的颜色。这代表了相机“看到”的颜色。
- **Target (RGBCX->RGB)** 列显示了目标 5 通道输出通过简化映射到 RGB 的颜色。这代表了理想情况下显示屏应该呈现的颜色。
- **Predicted (RGBCX->RGB)** 列显示了模型预测的 5 通道输出通过简化映射到 RGB 的颜色，并标注了与目标颜色的 ΔE_{2000} 误差。

通过对比 Target 和 Predicted 列的颜色，我们可以直观地看到模型转换的准确性。绝大多数样本的预测颜色与目标颜色非常接近，且 ΔE_{2000} 值较低，进一步验证了模型的有效性。例如，对于 ΔE_{2000} 值低于 1.0 的样本，人眼几乎无法区分预测色和目标色。即使对于略高的 ΔE_{2000} 值，颜色的感知差异也通常是可接受的。

3.4 问题 3：LED 显示器颜色校正模型

4 模型评价与推广

4.1 主要结论

4.2 模型优点

4.3 不足与改进方向

参考文献

[1] Finn C, Abbeel P, Levine S. Model-agnostic meta-learning for fast adaptation of deep networks [C]//International Conference on Machine Learning. 2017: 1126-1135.

附 录

A. 问题 1 使用代码

```
import warnings
warnings.filterwarnings("ignore")
import numpy as np
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
import colour
from colormath.color_objects import LabColor, XYZColor
from colormath.color_conversions import convert_color

BT2020 = [[0.708, 0.292], [0.170, 0.797], [0.131, 0.046]]
sRGB_DP = [[0.64, 0.33], [0.30, 0.60], [0.15, 0.06]]
NTSC = [[0.67, 0.33], [0.21, 0.71], [0.14, 0.08]]

M_sRGB_to_XYZ = np.array([
    [0.4124564, 0.3575761, 0.1804375],
    [0.2126729, 0.7151522, 0.0721750],
    [0.0193339, 0.1191920, 0.9503041]
])

def lab_to_xyz_batch(lab_array):
    result = []
    for lab in lab_array:
        lab_color = LabColor(*lab)
        xyz_color = convert_color(lab_color, XYZColor)
        result.append([xyz_color.xyz_x, xyz_color.xyz_y, xyz_color.xyz_z])
    return np.array(result)

def rgb_to_xy(rgb, M_rgb_to_xyz):
    xyz = rgb @ M_rgb_to_xyz.T
    x = xyz[:, 0] / (xyz[:, 0] + xyz[:, 1] + xyz[:, 2])
    y = xyz[:, 1] / (xyz[:, 0] + xyz[:, 1] + xyz[:, 2])
    return np.stack([x, y], axis=1)

def xyz_to_xy_test(M_opt, RGB_basic, M_bt2020_to_xyz):
    # BT2020 to DP
    M_opt_inv = np.linalg.inv(M_opt)
    dp_rgb_mapped = (M_opt_inv @ RGB_basic.T).T  # shape (3, 3)
    BT2020_to_DP_mapped = rgb_to_xy(dp_rgb_mapped, M_bt2020_to_xyz)
    return BT2020_to_DP_mapped

def chromaticity_to_xyz_matrix(primaries, whitepoint):
    M = []
    for x, y in primaries:
```

```
44          z = 1 - x - y
45          M.append([x / y, 1.0, z / y])
46      M = np.array(M).T
47      Xw, Yw, Zw = whitepoint
48      S = np.linalg.inv(M) @ np.array([Xw, Yw, Zw])
49      return M * S
50
51  def delta_e_00_batch(lab1, lab2):
52      lab1 = np.array(lab1)
53      lab2 = np.array(lab2)
54
55      L1, a1, b1 = lab1[:, 0], lab1[:, 1], lab1[:, 2]
56      L2, a2, b2 = lab2[:, 0], lab2[:, 1], lab2[:, 2]
57
58      avg_L = 0.5 * (L1 + L2)
59      C1 = np.sqrt(a1**2 + b1**2)
60      C2 = np.sqrt(a2**2 + b2**2)
61      avg_C = 0.5 * (C1 + C2)
62
63      G = 0.5 * (1 - np.sqrt((avg_C**7) / (avg_C**7 + 25**7)))
64      a1p = (1 + G) * a1
65      a2p = (1 + G) * a2
66      C1p = np.sqrt(a1p**2 + b1**2)
67      C2p = np.sqrt(a2p**2 + b2**2)
68      avg_Cp = 0.5 * (C1p + C2p)
69
70      h1p = np.degrees(np.arctan2(b1, a1p)) % 360
71      h2p = np.degrees(np.arctan2(b2, a2p)) % 360
72
73      deltahp = h2p - h1p
74      deltahp = np.where(deltahp > 180, deltahp - 360, deltahp)
75      deltahp = np.where(deltahp < -180, deltahp + 360, deltahp)
76
77      delta_Hp = 2 * np.sqrt(C1p * C2p) * np.sin(np.radians(deltahp / 2))
78      delta_Lp = L2 - L1
79      delta_Cp = C2p - C1p
80
81      avg_hp = np.where(np.abs(h1p - h2p) > 180, (h1p + h2p + 360) / 2, (h1p +
            ↪ h2p) / 2)
82      T = 1 - 0.17 * np.cos(np.radians(avg_hp - 30)) + 0.24 * np.cos(np.radians
            ↪ (2 * avg_hp)) \
83          + 0.32 * np.cos(np.radians(3 * avg_hp + 6)) - 0.20 * np.cos(np.radians
                ↪ (4 * avg_hp - 63))
84
85      delta_theta = 30 * np.exp(-((avg_hp - 275) / 25)**2)
86      Rc = 2 * np.sqrt((avg_Cp**7) / (avg_Cp**7 + 25**7))
87      Sl = 1 + (0.015 * (avg_L - 50)**2) / np.sqrt(20 + (avg_L - 50)**2)
88      Sc = 1 + 0.045 * avg_Cp
```

```
    Sh = 1 + 0.015 * avg_Cp * T
    Rt = -np.sin(np.radians(2 * delta_theta)) * Rc

    delta_E = np.sqrt(
        (delta_Lp / Sl)**2 +
        (delta_Cp / Sc)**2 +
        (delta_Hp / Sh)**2 +
        Rt * (delta_Cp / Sc) * (delta_Hp / Sh)
    )

    return delta_E

def f(t):
    delta = 6/29
    return np.where(t > delta**3, np.cbrt(t), (t / (3 * delta**2)) + (4/29))

def xyz_to_lab_batch(xyz, white_point=(0.95047, 1.00000, 1.08883)):
    Xn, Yn, Zn = white_point
    X = xyz[:, 0] / Xn
    Y = xyz[:, 1] / Yn
    Z = xyz[:, 2] / Zn

    fx = f(X)
    fy = f(Y)
    fz = f(Z)

    L = 116 * fy - 16
    a = 500 * (fx - fy)
    b = 200 * (fy - fz)

    return np.stack([L, a, b], axis=1)

def combined_loss(M_flat, rgb_samples, M_bt2020_to_xyz, M_dp_to_xyz,
    ↪ xyz_to_lab_batch):

    # 变换矩阵
    M = M_flat.reshape(3, 3)

    # ═══ ΔE00 感知损失 ═══
    rgb_dp = rgb_samples @ M.T
    rgb_dp = np.clip(rgb_dp, 0, 1)

    xyz_pred = rgb_dp @ M_dp_to_xyz.T
    lab_pred = xyz_to_lab_batch(xyz_pred)

    xyz_true = rgb_samples @ M_bt2020_to_xyz.T
    lab_true = xyz_to_lab_batch(xyz_true)

```

```
    deltaE = delta_e_00_batch(lab_true, lab_pred)
    color_loss = np.mean(deltaE)

    return color_loss

def optimize_model_N_times(whitepoint, sRGB_DP, M_flat_init, M_bt2020_to_xyz,
    ↪ M_dp_to_xyz, xyz_to_lab_batch,
                           N=10, method='DE', random_seed_offset=31):
    """
    对比不同优化器在 N 轮随机样本下的表现

    参数:
    - M_flat_init: 初始 M(flatten)
    - M_bt2020_to_xyz: BT2020 → XYZ 变换矩阵
    - M_dp_to_xyz: DP → XYZ 变换矩阵
    - xyz_to_lab_batch: XYZ → Lab 转换函数(批量)
    - N: 循环次数
    - method: 优化方法选择, 'L-BFGS-B' 或 'DE'
    - random_seed_offset: 随机种子偏移量, 确保每轮样本不同但可复现

    返回:
    - losses: ndarray[N], 每轮优化得到的 loss
    """

    losses = []
    area_diffs = []
    RGB_basic = np.eye(3)
    ref_area = triangle_area(sRGB_DP)

    for i in range(N):
        seed = i + random_seed_offset
        np.random.seed(seed)
        test_samples = np.random.rand(100, 3)

        def loss_fn(M_flat):
            return combined_loss(M_flat, test_samples, M_bt2020_to_xyz,
                ↪ M_dp_to_xyz, xyz_to_lab_batch)

        if method == 'DE':
            bounds = [(-2, 2)] * 9
            res = differential_evolution(
                loss_fn,
                bounds,
                strategy='best1bin',
                maxiter=1000,
                polish=True,
                seed=seed
            )
```

```
182             else:
183                 raise ValueError(f"Unknown method: {method}. Supported: 'DE'")
184 
185             M_opt = res.x.reshape(3, 3)
186             # ==========================
187             BT2020_to_DP_mapped = xyz_to_xy_test(M_opt, RGB_basic, M_bt2020_to_xyz
                    ↪ )
188             BT_mapped_xyz = chromaticity_to_xyz_matrix(BT2020_to_DP_mapped,
                    ↪ whitepoint)
189             BT_mapped_lab = xyz_to_lab_batch(BT_mapped_xyz)
190             BT_lab = xyz_to_lab_batch(M_sRGB_to_XYZ)
191             loss = np.mean(delta_e_00_batch(BT_mapped_lab, BT_lab))
192             # ==========================
193             # final_loss = loss_fn(res.x)
194             losses.append(loss)
195 
196             triangle_xy = xyz_to_xy_test(M_opt, RGB_basic, M_bt2020_to_xyz)
197             area = triangle_area(triangle_xy)
198             area_diff = abs(area - ref_area)
199             area_diffs.append(area_diff)
200 
201     return np.array(losses), np.array(area_diffs)
202 
203 def triangle_area(pts):
204     """
205     计算三角形面积: pts 是 3x2 的 xy 坐标矩阵
206     使用 Shoelace formula（鞋带公式）
207     """
208     pts = np.array(pts)
209     x = pts[:, 0]
210     y = pts[:, 1]
211     return 0.5 * abs(x[0]*(y[1]-y[2]) + x[1]*(y[2]-y[0]) + x[2]*(y[0]-y[1]))
212 
213 
214 def plot_chromaticity_with_triangles(example_dict):
215     """
216     在 CIE 1931 xy 色度图上叠加多个 RGB 三角形。
217     前两个三角形为实线，后续为虚线，图例使用变量名。
218     """
219     figure, axes = colour.plotting.plot_chromaticity_diagram_CIE1931(
            ↪ standalone=False)
220 
221     colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
222     linestyle_solid = '-'
223     linestyle_dashed = '--'
224 
225     for i, (label, triangle) in enumerate(example_dict.items()):
226         triangle = np.array(triangle)
```

```
        polygon = np.vstack([triangle, triangle[0]])
        linestyle = linestyle_solid if i < 2 else linestyle_dashed

        axes.plot(polygon[:, 0], polygon[:, 1],
                  color=colors[i % len(colors)],
                  linewidth=2,
                  linestyle=linestyle,
                  label=label)

    axes.legend()
    axes.set_title("CIE 1931 Chromaticity Diagram with RGB Triangles")
    plt.grid(True)
    plt.show()


def plot_loss_statistics(losses, title='Loss Distribution', method_name='L-BFGS-B'):
    """
    绘制柱状图并显示统计信息。

    参数:
    - losses: 一维 ndarray, 优化 N 次的 loss 值
    - title: 图表标题
    - method_name: 优化方法名称, 用于图表显示
    """

    # 计算统计量
    mean_loss = np.mean(losses)
    min_loss = np.min(losses)
    std_loss = np.std(losses)

    # 创建柱状图
    plt.figure(figsize=(10, 6))
    bars = plt.bar(range(len(losses)), losses, color='skyblue', edgecolor='black')

    # 高亮最小值
    min_index = np.argmin(losses)
    bars[min_index].set_color('orange')

    # 标注统计量
    plt.axhline(mean_loss, color='red', linestyle='--', label=f'Mean: {mean_loss:.4f}')
    plt.axhline(min_loss, color='green', linestyle='--', label=f'Min: {min_loss:.4f}')
    plt.text(len(losses) - 1, mean_loss + 0.05, f' : {std_loss:.4f}', color='red', fontsize=10, ha='right')

```

```
    # 图形美化
    plt.title(f'{title} ({method_name})', fontsize=14)
    plt.xlabel('Trial Index')
    plt.ylabel('Loss Value')
    plt.xticks(range(len(losses)))
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.5)

    plt.tight_layout()
    plt.show()

if __name__ =="__main__":
    # D65 whitepoint in XYZ
    whitepoint = (0.3127 / 0.3290, 1.0, (1 - 0.3127 - 0.3290) / 0.3290)
    # BT2020 → XYZ
    M_bt2020_to_xyz = chromaticity_to_xyz_matrix(BT2020, whitepoint)
    # DP/sRGB → XYZ
    M_dp_to_xyz = chromaticity_to_xyz_matrix(sRGB_DP, whitepoint)

    # ========== 训练部分 ==========
    # 采样一组 BT.2020 RGB 样本 {c_i}
    M0 = np.eye(3).flatten()
    M0_flat = np.eye(3).flatten()

    # L-BFGS-B 优化 50 次
    losses_lbfgs, area_diffs = optimize_model_N_times(
        whitepoint,
        sRGB_DP,
        M0_flat,
        M_bt2020_to_xyz,
        M_dp_to_xyz,
        xyz_to_lab_batch,
        N=50,
        method='DE'
    )
    print("DE Losses:", losses_lbfgs)

    plot_loss_statistics(losses_lbfgs, title='DE Distribution', method_name='
        ↪ Differential Evolution')
    plot_loss_statistics(area_diffs, title='Chromaticity Area Difference',
        ↪ method_name='Differential Evolution')
    # ========== 单独测试 ==========
    np.random.seed(35)
    test_samples = np.random.rand(100, 3)
    bounds = [(-2, 2)] * 9
    def loss_fn(M_flat):
        return combined_loss(M_flat, test_samples, M_bt2020_to_xyz,
            ↪ M_dp_to_xyz, xyz_to_lab_batch)
```

```
315
316     res1 = differential_evolution(
317         loss_fn,
318         bounds,
319         strategy='best1bin',
320         maxiter=1000,
321         polish=True,
322         seed=35,
323         )
324     M_opt = res1.x.reshape(3, 3)
325     # 映射到色度图上
326     # DB to BT2020
327     RGB_basic = np.eye(3)
328     # BT2020 to DP
329     BT2020_to_DP_mapped = xyz_to_xy_test(M_opt, RGB_basic, M_bt2020_to_xyz)
330
331     examples = {
332         "BT2020": BT2020,
333         "sRGB_DP": sRGB_DP,
334         "BT2020_to_DP_mapped": BT2020_to_DP_mapped
335     }
336
337     plot_chromaticity_with_triangles(examples)
```

B. 问题 2 使用代码

```
1  import torch
2  import torch.nn as nn
3  import torch.optim as optim
4  import numpy as np
5  import matplotlib.pyplot as plt
6  from sklearn.model_selection import train_test_split
7  from scipy.spatial import ConvexHull
8  import colour
9
10 # 定义RGB到XYZ的PyTorch函数
11 def rgb_to_xyz_torch(rgb):
12     device = rgb.device
13     # sRGB到XYZ的转换矩阵
14     M = torch.tensor([
15         [0.4124564, 0.3575761, 0.1804375],
16         [0.2126729, 0.7151522, 0.0721750],
17         [0.0193339, 0.1191920, 0.9503041]
18     ], device=device, dtype=torch.float32)
19     # sRGB到线性RGB的转换，处理Gamma校正
20     mask = rgb > 0.04045
21     rgb_linear = torch.where(mask,
22                              torch.pow((rgb + 0.055) / 1.055, 2.4),
```

```
                              rgb / 12.92)
    # 矩阵乘法完成转换
    return torch.matmul(rgb_linear, M.T)

# 定义XYZ到Lab的PyTorch函数
def xyz_to_lab_torch(xyz):
    device = xyz.device
    # D65白点
    white = torch.tensor([0.95047, 1.00000, 1.08883], device=device, dtype=
        ↪ torch.float32)
    # XYZ值相对于白点进行缩放
    xyz_scaled = xyz / white
    delta = 6.0 / 29.0
    # 定义Lab转换中的f函数
    def f(t):
        return torch.where(t > delta**3,
                           torch.pow(t, 1.0/3.0),
                           t / (3.0 * delta**2) + 4.0 / 29.0)
    # 应用f函数到XYZ的每个分量
    f_xyz = f(xyz_scaled)
    # 计算L*, a*, b*分量
    L = 116.0 * f_xyz[..., 1] - 16.0
    a = 500.0 * (f_xyz[..., 0] - f_xyz[..., 1])
    b = 200.0 * (f_xyz[..., 1] - f_xyz[..., 2])
    # 堆叠L, a, b分量
    return torch.stack([L, a, b], dim=-1)

# 定义RGB到Lab的PyTorch函数
def rgb_to_lab_torch(rgb):
    # 添加一个小的epsilon防止log(0)或者除以0的情况
    return xyz_to_lab_torch(rgb_to_xyz_torch(rgb.clamp(min=1e-8)))

# 定义DeltaE2000颜色差异的PyTorch函数
def deltaE2000_torch(lab1, lab2):
    L1, a1, b1 = lab1[..., 0], lab1[..., 1], lab1[..., 2]
    L2, a2, b2 = lab2[..., 0], lab2[..., 1], lab2[..., 2]

    # k_L, k_C, k_H 是权重因子，通常设为1
    k_L, k_C, k_H = 1.0, 1.0, 1.0

    # 计算CIE Lab中的色度C
    C1 = torch.sqrt(a1**2 + b1**2)
    C2 = torch.sqrt(a2**2 + b2**2)

    # 计算平均色度
    avg_C = (C1 + C2) / 2.0

    # 计算G因子，用于修正a'值
```

```
    G = 0.5 * (1 - torch.sqrt(avg_C**7 / (avg_C**7 + 25**7)))

    # 修正a'值
    a1p = (1 + G) * a1
    a2p = (1 + G) * a2

    # 计算修正后的色度C'
    C1p = torch.sqrt(a1p**2 + b1**2)
    C2p = torch.sqrt(a2p**2 + b2**2)

    # 计算色相角h' (以度为单位)
    h1p = torch.rad2deg(torch.atan2(b1, a1p))
    h1p = torch.where(h1p < 0, h1p + 360, h1p) # 确保角度在0-360度
    h2p = torch.rad2deg(torch.atan2(b2, a2p))
    h2p = torch.where(h2p < 0, h2p + 360, h2p)

    # 计算平均亮度L*
    avg_L = (L1 + L2) / 2.0
    # 计算平均修正色度C'*
    avg_Cp = (C1p + C2p) / 2.0

    # 计算色相角差Δh'
    h_diff = h2p - h1p
    delta_hp = torch.where(torch.abs(h_diff) <= 180, h_diff, h_diff - 360 *
        ↪ torch.sign(h_diff))

    # 计算亮度差ΔL', 色度差ΔC', 色相差ΔH'
    Delta_Lp = L2 - L1
    Delta_Cp = C2p - C1p
    Delta_hp = 2 * torch.sqrt(C1p * C2p) * torch.sin(torch.deg2rad(delta_hp /
        ↪ 2.0))

    # 计算平均色相角h' (特殊处理)
    h_sum = h1p + h2p
    avg_hp = torch.where(torch.abs(h_diff) > 180, (h_sum + 360) / 2, h_sum /
        ↪ 2)

    # 计算T因子
    T = (1 - 0.17 * torch.cos(torch.deg2rad(avg_hp - 30)) +
         0.24 * torch.cos(torch.deg2rad(2 * avg_hp)) +
         0.32 * torch.cos(torch.deg2rad(3 * avg_hp + 6)) -
         0.20 * torch.cos(torch.deg2rad(4 * avg_hp - 63)))

    # 计算Δ因子 (旋转项)
    delta_ro = 30 * torch.exp(-((avg_hp - 275) / 25)**2)

    # 计算色度权重Rc
    R_C = 2 * torch.sqrt(avg_Cp**7 / (avg_Cp**7 + 25**7))
```

```
    # 计算亮度权重Sl
    S_L = 1 + (0.015 * (avg_L - 50)**2) / torch.sqrt(20 + (avg_L - 50)**2)
    # 计算色度权重Sc
    S_C = 1 + 0.045 * avg_Cp
    # 计算色相权重Sh
    S_H = 1 + 0.015 * avg_Cp * T
    # 计算旋转项Rt
    R_T = -torch.sin(torch.deg2rad(2 * delta_ro)) * R_C

    # 最终的DeltaE2000公式
    delta_E = torch.sqrt(
        (Delta_Lp / (k_L * S_L))**2 +
        (Delta_Cp / (k_C * S_C))**2 +
        (Delta_hp / (k_H * S_H))**2 +
        R_T * (Delta_Cp / (k_C * S_C)) * (Delta_hp / (k_H * S_H))
    )

    return delta_E

# 定义一个结合MSE和DeltaE2000的混合损失函数
class CombinedLoss(nn.Module):

    def __init__(self, alpha=0.1, beta=1.0):
        super().__init__()
        self.alpha = alpha # MSE损失的权重
        self.beta = beta   # DeltaE2000损失的权重
        self.mse_loss = nn.MSELoss() # 初始化MSE损失

    def forward(self, pred_rgbcx, target_rgbcx):
        # 对所有5个通道计算MSE损失
        loss_mse = self.mse_loss(pred_rgbcx, target_rgbcx)

        # 对前3个通道（RGB）计算DeltaE2000损失
        pred_rgb = pred_rgbcx[:, :3]
        target_rgb = target_rgbcx[:, :3]

        # 将RGB转换为Lab颜色空间
        pred_lab = rgb_to_lab_torch(pred_rgb)
        target_lab = rgb_to_lab_torch(target_rgb)

        # 计算DeltaE2000损失的平均值
        loss_delta_e = torch.mean(deltaE2000_torch(pred_lab, target_lab))

        # 结合两种损失
        total_loss = self.alpha * loss_mse + self.beta * loss_delta_e
        return total_loss

# 定义RGB到XYZ的NumPy函数 (与PyTorch版本对应)
```

```
def rgb_to_xyz(rgb):
    mask = rgb > 0.04045
    rgb_linear = np.where(mask, ((rgb + 0.055)/1.055)**2.4, rgb / 12.92)
    M = np.array([[0.4124564, 0.3575761, 0.1804375],
                  [0.2126729, 0.7151522, 0.0721750],
                  [0.0193339, 0.1191920, 0.9503041]])
    return np.dot(rgb_linear, M.T)

# 定义XYZ到Lab的NumPy函数 (与PyTorch版本对应)
def xyz_to_lab(xyz):
    white = np.array([0.95047, 1.00000, 1.08883])
    xyz_scaled = xyz / white
    def f(t):
        delta = 6/29
        return np.where(t > delta**3, np.cbrt(t), t/(3*delta**2) + 4/29)
    f_xyz = f(xyz_scaled)
    L = 116*f_xyz[...,1] - 16
    a = 500*(f_xyz[...,0] - f_xyz[...,1])
    b = 200*(f_xyz[...,1] - f_xyz[...,2])
    return np.stack([L,a,b], axis=-1)

# 定义RGB到Lab的NumPy函数 (与PyTorch版本对应)
def rgb_to_lab(rgb):
    return xyz_to_lab(rgb_to_xyz(np.clip(rgb, 0, 1)))

# 定义DeltaE2000颜色差异的NumPy函数 (与PyTorch版本对应)
def deltaE2000(Lab1, Lab2):
    L1, a1, b1 = Lab1[...,0], Lab1[...,1], Lab1[...,2]
    L2, a2, b2 = Lab2[...,0], Lab2[...,1], Lab2[...,2]
    avg_L = 0.5 * (L1 + L2)
    C1 = np.sqrt(a1**2 + b1**2)
    C2 = np.sqrt(a2**2 + b2**2)
    avg_C = 0.5 * (C1 + C2)
    G = 0.5 * (1 - np.sqrt((avg_C**7) / (avg_C**7 + 25**7)))
    a1p = (1 + G) * a1
    a2p = (1 + G) * a2
    C1p = np.sqrt(a1p**2 + b1**2)
    C2p = np.sqrt(a2p**2 + b2p**2) # 修正: b2p应该是b2
    h1p = np.degrees(np.arctan2(b1, a1p)) % 360
    h2p = np.degrees(np.arctan2(b2, a2p)) % 360
    delta_Lp = L2 - L1
    delta_Cp = C2p - C1p
    dhp = h2p - h1p
    dhp = np.where(np.abs(dhp) > 180, dhp - 360 * np.sign(dhp), dhp)
    delta_hp = 2 * np.sqrt(C1p * C2p) * np.sin(np.radians(dhp / 2))
    avg_Lp = (L1 + L2) / 2
    avg_Cp = (C1p + C2p) / 2
    hp_sum = h1p + h2p
```

```
    avg_hp = np.where(np.abs(h1p - h2p) > 180, (hp_sum + 360) / 2, hp_sum / 2)
    T = 1 - 0.17*np.cos(np.radians(avg_hp - 30)) + \
        0.24*np.cos(np.radians(2*avg_hp)) + \
        0.32*np.cos(np.radians(3*avg_hp + 6)) - \
        0.20*np.cos(np.radians(4*avg_hp - 63))
    delta_ro = 30 * np.exp(-((avg_hp - 275)/25)**2)
    Rc = 2 * np.sqrt((avg_Cp**7) / (avg_Cp**7 + 25**7))
    Sl = 1 + ((0.015 * (avg_Lp - 50)**2) / np.sqrt(20 + (avg_Lp - 50)**2))
    Sc = 1 + 0.045 * avg_Cp
    Sh = 1 + 0.015 * avg_Cp * T
    Rt = -np.sin(np.radians(2 * delta_ro)) * Rc
    delta_E = np.sqrt(
        (delta_Lp / Sl)**2 +
        (delta_Cp / Sc)**2 +
        (delta_hp / Sh)**2 +
        Rt * (delta_Cp / Sc) * (delta_hp / Sh))
    return delta_E

# 定义神经网络模型 ColorNet
class ColorNet(nn.Module):
    def __init__(self):
        super(ColorNet, self).__init__()
        # 定义一个简单的全连接神经网络
        self.net = nn.Sequential(
            nn.Linear(4, 64), # 输入4通道 (RGBV)
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 5), # 输出5通道 (RGBCX)
            nn.Sigmoid() # 输出值在0-1之间
        )
    def forward(self, x):
        return self.net(x)

# 生成训练数据
def generate_train_data(n_samples=2000):
    # X是相机输入 RGBV (4通道)
    X = np.random.rand(n_samples, 4).astype(np.float32)

    # 定义一个简单线性变换矩阵 W, 模拟相机对不同通道光的响应
    W = np.array([
        [0.9, 0.05, 0.03, 0.02],    # R_out = 0.9*R_in + 0.05*G_in + ...
        [0.05, 0.85, 0.05, 0.05],   # G_out
        [0.02, 0.03, 0.9, 0.05],    # B_out
        [0.01, 0.02, 0.03, 0.9],    # C_out (受V通道影响较大)
        [0.02, 0.05, 0.02, 0.91]    # X_out (受V通道影响较大)
```

```
    ], dtype=np.float32)

    Y_linear = X.dot(W.T) # 线性变换
    # 加入非线性扰动，使模型学习更复杂的映射
    Y_nonlinear = Y_linear + 0.02 * np.sin(5 * np.pi * X[:, 0:1])
    Y_nonlinear = np.clip(Y_nonlinear, 0, 1) # 确保颜色值在0-1范围内
    return X, Y_nonlinear.astype(np.float32)

# 训练模型
def train_model(X, Y, epochs=100, batch_size=32, lr=1e-3):
    # 设置设备为GPU（如果可用）或CPU
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = ColorNet().to(device) # 将模型移到指定设备
    optimizer = optim.AdamW(model.parameters(), lr=lr) # 使用AdamW优化器
    # 划分训练集和验证集
    X_train, X_val, Y_train, Y_val = train_test_split(X, Y, test_size=0.2,
        ↪ random_state=42)
    train_losses, val_losses = [], []

    loss_fn = CombinedLoss(alpha=0.1, beta=1.0).to(device) # 初始化混合损失函
        ↪ 数

    for epoch in range(epochs):
        model.train() # 设置模型为训练模式
        permutation = np.random.permutation(len(X_train)) # 随机打乱训练数据
        epoch_loss = 0
        for i in range(0, len(X_train), batch_size):
            indices = permutation[i:i+batch_size]
            batch_x = torch.tensor(X_train[indices], dtype=torch.float32,
                ↪ device=device)
            batch_y = torch.tensor(Y_train[indices], dtype=torch.float32,
                ↪ device=device)

            optimizer.zero_grad() # 梯度清零
            outputs = model(batch_x) # 前向传播

            loss = loss_fn(outputs, batch_y) # 计算损失

            loss.backward() # 反向传播
            optimizer.step() # 更新模型参数
            epoch_loss += loss.item() * len(indices)
        avg_train_loss = epoch_loss / len(X_train)
        train_losses.append(avg_train_loss)

        model.eval() # 设置模型为评估模式
        with torch.no_grad(): # 禁用梯度计算
            val_x = torch.tensor(X_val, dtype=torch.float32, device=device)
            val_y = torch.tensor(Y_val, dtype=torch.float32, device=device)
```

```
303                 val_pred = model(val_x)
304
305                 val_loss = loss_fn(val_pred, val_y).item()
306             val_losses.append(val_loss)
307
308             if (epoch + 1) % 10 == 0 or epoch == 0:
309                 print(f"Epoch {epoch+1}/{epochs} - Train Loss: {avg_train_loss:.6f
                        ↪ } - Val Loss: {val_loss:.6f}")
310
311     return model, train_losses, val_losses, X_val, Y_val
312
313 # 可视化DeltaE2000误差分布
314 def visualize_errors(model, X_val, Y_val):
315     device = next(model.parameters()).device
316     model.eval() # 设置模型为评估模式
317     with torch.no_grad(): # 禁用梯度计算
318         inputs = torch.tensor(X_val, dtype=torch.float32, device=device)
319         outputs = model(inputs).cpu().numpy() # 获取模型输出并转为NumPy数组
320         targets = Y_val
321
322     pred_rgb = outputs[:, :3] # 提取预测的RGB分量
323     target_rgb = targets[:, :3] # 提取目标的RGB分量
324
325     # 将RGB转换为Lab，并计算DeltaE2000
326     pred_lab = rgb_to_lab(pred_rgb)
327     target_lab = rgb_to_lab(target_rgb)
328     delta_e = deltaE2000(pred_lab, target_lab)
329
330     plt.figure(figsize=(6,4))
331     plt.hist(delta_e, bins=50, color='skyblue', range=(0, max(5, np.max(
            ↪ delta_e))))
332     plt.title('ΔE2000 Error Histogram (Trained with Hybrid Loss)')
333     plt.xlabel('ΔE2000')
334     plt.ylabel('Pixel Count')
335     plt.show()
336
337     sorted_de = np.sort(delta_e)
338     cdf = np.arange(len(sorted_de)) / float(len(sorted_de))
339
340     plt.figure(figsize=(6,4))
341     plt.plot(sorted_de, cdf, color='green')
342     plt.title('CDF of ΔE2000 Error (Trained with Hybrid Loss)')
343     plt.xlabel('ΔE2000')
344     plt.ylabel('Cumulative Probability')
345     plt.grid(True)
346     plt.show()
347
348 # 绘制色度图和色域三角形
```

```
def plot_chromaticity_with_triangles(example_dict):
    # 使用colour库绘制CIE 1931色度图
    figure, axes = colour.plotting.plot_chromaticity_diagram_CIE1931(
        ↪ standalone=False)

    colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
    linestyle_solid = '-'
    linestyle_dashed = '--'

    # 遍历字典中的每个色域并绘制
    for i, (label, triangle) in enumerate(example_dict.items()):
        triangle = np.array(triangle)

        # 闭合多边形以绘制三角形
        polygon = np.vstack([triangle, triangle[0]])
        linestyle = linestyle_solid if i < 2 else linestyle_dashed # 前两个用
            ↪ 实线，后面用虚线

        # 绘制色域边界
        axes.plot(polygon[:, 0], polygon[:, 1],
                  color=colors[i % len(colors)],
                  linewidth=2,
                  linestyle=linestyle,
                  label=label)

        # 绘制每个基色点
        point_labels = ['R', 'G', 'B', 'V'] if 'Input' in label else ['R', 'G'
            ↪ , 'C', 'B', 'X']
        for j, point in enumerate(triangle):
            axes.scatter(point[0], point[1], color=colors[i % len(colors)], s
                ↪ =70, zorder=5, edgecolors='black')
            # 标记基色点
            axes.text(point[0] + 0.005, point[1] + 0.005, f'{point_labels[j]}'
                ↪ , fontsize=10, color='black')

    axes.legend() # 显示图例
    axes.set_title("CIE 1931 Chromaticity Diagram with Multi-Primary Gamuts")
    plt.grid(True)
    plt.show()

# 辅助函数：将RGBV数据转换为用于显示的RGB
def _rgbv_to_rgb_display(rgbv):
    if rgbv.ndim == 1: # 如果是单个样本
        r, g, b, v = rgbv
    else: # 如果是批量样本
        r, g, b, v = rgbv[:, 0], rgbv[:, 1], rgbv[:, 2], rgbv[:, 3]

    # 简单的融合V通道到RGB，用于可视化
```

```
    r_display = np.clip(r + v * 0.1, 0, 1)
    g_display = np.clip(g, 0, 1)
    b_display = np.clip(b + v * 0.2, 0, 1)

    return np.stack([r_display, g_display, b_display], axis=-1)

# 辅助函数：将RGBCX数据转换为用于显示的RGB
def _rgbcx_to_rgb_display(rgbcx):
    if rgbcx.ndim == 1: # 如果是单个样本
        r, g, b, c, x = rgbcx
    else: # 如果是批量样本
        r, g, b, c, x = rgbcx[:, 0], rgbcx[:, 1], rgbcx[:, 2], rgbcx[:, 3],
            ↪ rgbcx[:, 4]

    # 简化的C和X通道融合到RGB，用于可视化
    # C (Cyan) 影响 G 和 B
    # X (Extra Red) 影响 R
    r_display = np.clip(r + x * 0.3, 0, 1) # X通道增加红色
    g_display = np.clip(g + c * 0.2, 0, 1) # C通道增加绿色
    b_display = np.clip(b + c * 0.3, 0, 1) # C通道增加蓝色

    return np.stack([r_display, g_display, b_display], axis=-1)

# 可视化部分样本的预测结果
def visualize_sample_predictions(model, X_val, Y_val, num_samples=5):
    device = next(model.parameters()).device
    model.eval() # 设置模型为评估模式

    # 随机选择num_samples个样本
    indices = np.random.choice(len(X_val), num_samples, replace=False)

    fig, axes = plt.subplots(num_samples, 3, figsize=(9, 2 * num_samples))
    fig.suptitle('Sample Color Predictions (Input RGBV -> Output RGBCX)',
        ↪ fontsize=16)

    for i, idx in enumerate(indices):
        input_rgbv = X_val[idx]
        target_rgbcx = Y_val[idx]

        # 模型预测
        with torch.no_grad():
            pred_rgbcx_tensor = model(torch.tensor(input_rgbv, dtype=torch.
                ↪ float32, device=device).unsqueeze(0))
            pred_rgbcx = pred_rgbcx_tensor.squeeze(0).cpu().numpy() # 获取预测
                ↪ 结果并转为NumPy数组

        # 将RGBV转换为RGB用于显示（简化处理）
        display_input_rgb = _rgbv_to_rgb_display(input_rgbv)
```

```
436
437         # 将RGBCX转换为RGB用于显示（简化处理）
438         display_target_rgb = _rgbcx_to_rgb_display(target_rgbcx)
439         display_pred_rgb = _rgbcx_to_rgb_display(pred_rgbcx)
440
441         # 计算显示用的RGB之间的DeltaE2000
442         delta_e = deltaE2000(rgb_to_lab(display_pred_rgb), rgb_to_lab(
                ↪ display_target_rgb))
443
444         # 绘制输入、目标和预测的颜色块
445         ax = axes[i, 0]
446         ax.imshow([[display_input_rgb]]) # imshow需要2D数组，所以用[[color]]
447         ax.set_title(f'Input (RGBV)\nSample {idx}', fontsize=8)
448         ax.axis('off')
449
450         ax = axes[i, 1]
451         ax.imshow([[display_target_rgb]])
452         ax.set_title(f'Target (RGBCX->RGB)', fontsize=8)
453         ax.axis('off')
454
455         ax = axes[i, 2]
456         ax.imshow([[display_pred_rgb]])
457         ax.set_title(f'Predicted (RGBCX->RGB)\nΔE2000: {delta_e:.2f}',
                ↪ fontsize=8)
458         ax.axis('off')
459
460     plt.tight_layout(rect=[0, 0.03, 1, 0.95]) # 调整布局
461     plt.savefig('sample_predictions.png', dpi=300) # 保存图片
462     plt.show()
463
464 # 主程序
465 if __name__ == '__main__':
466     X, Y = generate_train_data(n_samples=4000) # 生成训练数据
467
468     # 训练模型
469     model, train_losses, val_losses, X_val, Y_val = train_model(X, Y, epochs
            ↪ =200, lr=5e-4)
470
471     # 绘制训练损失和验证损失曲线
472     plt.plot(train_losses, label='Train Loss')
473     plt.plot(val_losses, label='Validation Loss')
474     plt.xlabel('Epoch')
475     plt.ylabel('Combined Loss (0.1*MSE + 1.0*ΔE2000)')
476     plt.legend()
477     plt.title('Training Loss Curve (Hybrid Loss)')
478     plt.show()
479
480     # 定义标准sRGB的R,G,B基色坐标
```

```
    PRIMARY_R = [0.64, 0.33]
    PRIMARY_G = [0.30, 0.60]
    PRIMARY_B = [0.15, 0.06]

    # 定义相机新增的 'V' (Violet/紫色) 基色坐标
    # 选择一个在蓝色和光谱轨迹紫色区域之间的点
    PRIMARY_V = [0.16, 0.03]

    # 定义LED屏新增的 'C' (Cyan/青色) 和 'X' (假设为一种更深的红色) 基色坐标
    # 选择一个能扩展蓝绿边界的青色点
    PRIMARY_C = [0.18, 0.45]
    # 选择一个比sRGB的R更红的点，以扩展红色边界
    PRIMARY_X = [0.70, 0.30]

    # 组合成输入和输出系统的基色字典
    input_system_primaries_coords = {
        'RGBV Input Gamut': [PRIMARY_R, PRIMARY_G, PRIMARY_B, PRIMARY_V]
    }

    output_system_primaries_coords = {
        'RGBCX Output Gamut': [PRIMARY_R, PRIMARY_G, PRIMARY_C, PRIMARY_B,
            ↪ PRIMARY_X]
    }

    # 合并所有色域数据用于绘图
    all_gamuts_for_plotting = {**input_system_primaries_coords, **
        ↪ output_system_primaries_coords}

    # 可视化DeltaE2000误差分布
    visualize_errors(model, X_val, Y_val)

    # 绘制色度图和色域三角形
    plot_chromaticity_with_triangles(all_gamuts_for_plotting)

    # 可视化部分样本的预测效果
    visualize_sample_predictions(model, X_val, Y_val, num_samples=8)
```

C. 问题 3 使用代码

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.optimize import minimize, differential_evolution

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei', 'DejaVu Sans']
plt.rcParams['axes.unicode_minus'] = False

```

```
class LEDColorCorrection:
    """
    基于三基色原理和CIE Lab色彩空间的颜色校正
    使用差分进化算法优化校正矩阵
    """

    def __init__(self):
        self.correction_matrix = None
        self.correction_bias = None
        self.gamma_correction = None
        self.measured_data = None
        self.target_data = None

    def load_excel_data(self, excel_path):
        """从Excel文件加载数据"""
        print(f"正在加载Excel文件: {excel_path}")

        sheets = ['R', 'G', 'B', 'target_R', 'target_G', 'target_B']
        data_dict = {}

        for sheet_name in sheets:
            df = pd.read_excel(excel_path, sheet_name=sheet_name, header=None)
                ↪ .iloc[0:64,0:64]
            data_dict[sheet_name] = df.values
            print(f"已加载工作表 '{sheet_name}': {df.shape}")

        # 组织数据
        self.measured_data = np.stack([
            data_dict['R'],
            data_dict['G'],
            data_dict['B']
        ], axis=-1)

        self.target_data = np.stack([
            data_dict['target_R'],
            data_dict['target_G'],
            data_dict['target_B']
        ], axis=-1)

        print(f"测量数据形状: {self.measured_data.shape}")
        print(f"目标数据形状: {self.target_data.shape}")

    def rgb_to_xyz(self, rgb):
        """RGB转XYZ色彩空间"""
        rgb_norm = rgb / 255.0

        # Gamma校正
        rgb_linear = np.where(rgb_norm <= 0.04045,
```

```
                                rgb_norm / 12.92,
                                np.power((rgb_norm + 0.055) / 1.055, 2.4))

        # sRGB到XYZ的转换矩阵
        transform_matrix = np.array([
            [0.4124564, 0.3575761, 0.1804375],
            [0.2126729, 0.7151522, 0.0721750],
            [0.0193339, 0.1191920, 0.9503041]
        ])

        xyz = np.dot(rgb_linear, transform_matrix.T)
        return xyz

    def xyz_to_lab(self, xyz):
        """XYZ转CIE Lab色彩空间"""
        # D65白点
        Xn, Yn, Zn = 0.95047, 1.00000, 1.08883

        x = xyz[..., 0] / Xn
        y = xyz[..., 1] / Yn
        z = xyz[..., 2] / Zn

        # 立方根变换
        fx = np.where(x > 0.008856, np.power(x, 1/3), (7.787 * x + 16/116))
        fy = np.where(y > 0.008856, np.power(y, 1/3), (7.787 * y + 16/116))
        fz = np.where(z > 0.008856, np.power(z, 1/3), (7.787 * z + 16/116))

        L = 116 * fy - 16
        a = 500 * (fx - fy)
        b = 200 * (fy - fz)

        return np.stack([L, a, b], axis=-1)

    def calculate_color_difference(self, lab1, lab2):
        """计算CIE Delta E 2000色差"""
        L1, a1, b1 = lab1[..., 0], lab1[..., 1], lab1[..., 2]
        L2, a2, b2 = lab2[..., 0], lab2[..., 1], lab2[..., 2]

        C1 = np.sqrt(a1**2 + b1**2)
        C2 = np.sqrt(a2**2 + b2**2)
        C_bar = 0.5 * (C1 + C2)

        G = 0.5 * (1 - np.sqrt(C_bar**7 / (C_bar**7 + 25**7)))
        a1p = (1 + G) * a1
        a2p = (1 + G) * a2

        C1p = np.sqrt(a1p**2 + b1**2)
        C2p = np.sqrt(a2p**2 + b2**2)
```

```
105
106         h1p = np.degrees(np.arctan2(b1, a1p)) % 360
107         h2p = np.degrees(np.arctan2(b2, a2p)) % 360
108
109         dLp = L2 - L1
110         dCp = C2p - C1p
111
112         dhp = h2p - h1p
113         dhp = dhp - 360 * (dhp > 180) + 360 * (dhp < -180)
114         dHp = 2 * np.sqrt(C1p * C2p) * np.sin(np.radians(dhp / 2))
115
116         L_bar = 0.5 * (L1 + L2)
117         C_bar_p = 0.5 * (C1p + C2p)
118
119         h_bar_p = (h1p + h2p + 360 * (np.abs(h1p - h2p) > 180)) / 2
120         h_bar_p %= 360
121
122         T = (1
123             - 0.17 * np.cos(np.radians(h_bar_p - 30))
124             + 0.24 * np.cos(np.radians(2 * h_bar_p))
125             + 0.32 * np.cos(np.radians(3 * h_bar_p + 6))
126             - 0.20 * np.cos(np.radians(4 * h_bar_p - 63)))
127
128         Sl = 1 + (0.015 * (L_bar - 50)**2) / np.sqrt(20 + (L_bar - 50)**2)
129         Sc = 1 + 0.045 * C_bar_p
130         Sh = 1 + 0.015 * C_bar_p * T
131
132         delta_theta = 30 * np.exp(-((h_bar_p - 275)/25)**2)
133         Rc = 2 * np.sqrt(C_bar_p**7 / (C_bar_p**7 + 25**7))
134         Rt = -np.sin(np.radians(2 * delta_theta)) * Rc
135
136         dE = np.sqrt(
137             (dLp / Sl)**2 +
138             (dCp / Sc)**2 +
139             (dHp / Sh)**2 +
140             Rt * (dCp / Sc) * (dHp / Sh)
141         )
142
143         return dE
144
145     def estimate_gamma_parameters(self):
146         """估计LED的Gamma参数（保留线性比例偏移）"""
147         print("正在估计Gamma参数...")
148         gamma_params = {}
149         for i, channel in enumerate(['R', 'G', 'B']):
150             meas = self.measured_data[..., i].flatten() / 255.0
151             targ = self.target_data[..., i].flatten() / 255.0
152             mask = (targ > 0.04) & (targ < 0.96) & (meas > 0)
```

```
153             m = meas[mask]
154             t = targ[mask]
155             if len(m) > 0:
156                 # 拟合 log(m) = gamma * log(t) + offset
157                 A = np.vstack([np.log(t + 1e-8), np.ones_like(t)]).T
158                 gamma, offset = np.linalg.lstsq(A, np.log(m + 1e-8), rcond=
                        ↪ None)[0]
159                 gamma = float(np.clip(gamma, 0.1, 3.0))
160                 scale = float(np.exp(offset))
161             else:
162                 gamma, scale = 1.0, 1.0
163             gamma_params[channel] = {'gamma': gamma, 'scale': scale}
164             print(f"{channel}通道 Gamma: {gamma:.3f}, Scale: {scale:.3f}")
165         self.gamma_correction = gamma_params
166         return gamma_params
167
168     def apply_gamma_correction(self, rgb_data, inverse=False):
169         """应用Gamma校正: 在归一化 [0,1] 空间先应用线性比例, 再做幂运算"""
170         if self.gamma_correction is None:
171             return rgb_data
172         data = rgb_data.astype(np.float32) / 255.0
173         out = np.zeros_like(data)
174         for i, channel in enumerate(['R', 'G', 'B']):
175             gamma = self.gamma_correction[channel]['gamma']
176             scale = self.gamma_correction[channel]['scale']
177             ch = data[..., i]
178             if not inverse:
179                 # 前向: 先比例, 再幂
180                 tmp = ch * scale
181                 tmp = np.clip(tmp, 0.0, 1.0)
182                 out_ch = np.power(tmp, gamma)
183             else:
184                 # 反向: 开幂, 再去比例
185                 tmp = np.power(ch, 1.0 / gamma)
186                 out_ch = tmp / np.maximum(scale, 1e-8)
187             out[..., i] = np.clip(out_ch, 0.0, 1.0)
188         # 恢复到 [0,255]
189         return (out * 255.0).astype(rgb_data.dtype)
190
191     def correction_function(self, params, measured_lin, target_lin):
192         """
193         优化函数: 线性校正矩阵 M 和偏置 b, params 长度 12。
194         corrected = clip(M @ measured + b, [0,1])
195         计算 ΔE  + 正则化。
196         """
197         M = params[:9].reshape(3,3)
198         b = params[9:].reshape(1,3)
199
```

```
200         # 应用矩阵和偏置
201         corr = np.dot(measured_lin, M.T) + b
202         corr = np.clip(corr, 0.0, 1.0)
203 
204         # 转到 XYZ → Lab
205         transform = np.array([[0.4124564,0.3575761,0.1804375],
206                               [0.2126729,0.7151522,0.0721750],
207                               [0.0193339,0.1191920,0.9503041]])
208         tgt_xyz = np.dot(target_lin, transform.T)
209         corr_xyz = np.dot(corr, transform.T)
210         tgt_lab = self.xyz_to_lab(tgt_xyz.reshape(-1,3)).reshape(corr.shape)
211         corr_lab = self.xyz_to_lab(corr_xyz.reshape(-1,3)).reshape(corr.shape)
212 
213         # 色差
214         deltaE = self.calculate_color_difference(tgt_lab, corr_lab)
215         loss = np.mean(deltaE)
216 
217         # 矩阵正则 + 偏置正则
218         loss += 0.001 * (np.sum((M - np.eye(3))**2) + np.sum(b**2))
219         det = np.linalg.det(M)
220         if det <= 0 or abs(det) < 0.1:
221             loss += 1000.0
222         return loss
223 
224 
225     def calibrate_correction_matrix(self):
226         print("开始校正: 矩阵 + 偏置...")
227         self.estimate_gamma_parameters()
228         # 预处理: 线性化
229         meas = self.apply_gamma_correction(self.measured_data.astype(np.
                ↪ float32), inverse=True)/255.0
230         targ = self.apply_gamma_correction(self.target_data.astype(np.float32)
                ↪ , inverse=True)/255.0
231         meas_flat = meas.reshape(-1,3)
232         targ_flat = targ.reshape(-1,3)
233         # 差分进化优化 12 参数
234         bounds = [(-2,2)]*9 + [(-0.1,0.1)]*3
235         res = differential_evolution(
236             self.correction_function, bounds,
237             args=(meas_flat, targ_flat), maxiter=200, popsize=15, seed=42
238         )
239         x0 = res.x
240         # 局部 L-BFGS-B
241         local = minimize(
242             self.correction_function, x0, args=(meas_flat,targ_flat),
243             method='L-BFGS-B', options={'maxiter':500}
244         )
245         M_opt = local.x[:9].reshape(3,3)
```

```
246         b_opt = local.x[9:].reshape(3)
247         self.correction_matrix = M_opt
248         self.correction_bias = b_opt
249         print("校正完成; 矩阵行列式: ", np.linalg.det(M_opt))
250         print("偏置: ", b_opt)
251         return M_opt, b_opt
252 
253 
254     def apply_correction(self, input_rgb):
255         """应用带偏置的线性校正"""
256         lin = self.apply_gamma_correction(input_rgb.astype(np.float32),
                ↪ inverse=True)/255.0
257         flat = lin.reshape(-1,3)
258         corr = np.dot(flat, self.correction_matrix.T) + self.correction_bias
259         corr = np.clip(corr, 0.0, 1.0).reshape(input_rgb.shape)
260         out = (corr * 255.0).astype(np.float32)
261         final = self.apply_gamma_correction(out, inverse=False)
262         return final.astype(np.uint8)
263 
264     def evaluate_correction(self):
265         """评估校正效果"""
266         corrected = self.apply_correction(self.measured_data.astype(np.float32
                ↪ ))
267 
268         measured_xyz = self.rgb_to_xyz(self.measured_data.astype(np.float32))
269         corrected_xyz = self.rgb_to_xyz(corrected.astype(np.float32))
270         target_xyz = self.rgb_to_xyz(self.target_data.astype(np.float32))
271 
272         measured_lab = self.xyz_to_lab(measured_xyz)
273         corrected_lab = self.xyz_to_lab(corrected_xyz)
274         target_lab = self.xyz_to_lab(target_xyz)
275 
276         diff_before = self.calculate_color_difference(measured_lab, target_lab
                ↪ )
277         diff_after = self.calculate_color_difference(corrected_lab, target_lab
                ↪ )
278 
279         print("="*50)
280         print("校正效果评估报告")
281         print("="*50)
282         print(f"校正前平均色差: {np.mean(diff_before):.3f}")
283         print(f"校正后平均色差: {np.mean(diff_after):.3f}")
284         print(f"色差改善: {np.mean(diff_before) - np.mean(diff_after):.3f}")
285         print(f"改善百分比: {((np.mean(diff_before) - np.mean(diff_after)) /
                ↪ np.mean(diff_before) * 100):.1f}%")
286         print(f"校正前最大色差: {np.max(diff_before):.3f}")
287         print(f"校正后最大色差: {np.max(diff_after):.3f}")
288         print(f"色差<1.0的像素比例: 校正前{np.mean(diff_before < 1.0)*100:.1f
```

```
            ↪ }%, 校正后{np.mean(diff_after < 1.0)*100:.1f}%")
289         print("="*50)
290
291         return corrected, diff_before, diff_after
292
293     def visualize_results(self):
294         """可视化校正结果"""
295         corrected_data = self.apply_correction(self.measured_data.astype(np.
                ↪ float32))
296
297         fig, axes = plt.subplots(3, 4, figsize=(20, 15))
298
299         # 第一行：测量数据
300         for i, (channel, color) in enumerate(zip(['R', 'G', 'B'], ['Reds', '
                ↪ Greens', 'Blues'])):
301             im = axes[0, i].imshow(self.measured_data[:, :, i], cmap=color,
                    ↪ vmin=0, vmax=255)
302             axes[0, i].set_title(f'测量值 - {channel} 通道')
303             axes[0, i].axis('off')
304             plt.colorbar(im, ax=axes[0, i], fraction=0.046, pad=0.04)
305
306         measured_rgb = np.clip(self.measured_data / 255.0, 0, 1)
307         axes[0, 3].imshow(measured_rgb)
308         axes[0, 3].set_title('测量值 - RGB合成')
309         axes[0, 3].axis('off')
310
311         # 第二行：目标数据
312         for i, (channel, color) in enumerate(zip(['R', 'G', 'B'], ['Reds', '
                ↪ Greens', 'Blues'])):
313             im = axes[1, i].imshow(self.target_data[:, :, i], cmap=color, vmin
                    ↪ =0, vmax=255)
314             axes[1, i].set_title(f'目标值 - {channel} 通道')
315             axes[1, i].axis('off')
316             plt.colorbar(im, ax=axes[1, i], fraction=0.046, pad=0.04)
317
318         target_rgb = np.clip(self.target_data / 255.0, 0, 1)
319         axes[1, 3].imshow(target_rgb)
320         axes[1, 3].set_title('目标值 - RGB合成')
321         axes[1, 3].axis('off')
322
323         # 第三行：校正后数据
324         for i, (channel, color) in enumerate(zip(['R', 'G', 'B'], ['Reds', '
                ↪ Greens', 'Blues'])):
325             im = axes[2, i].imshow(corrected_data[:, :, i], cmap=color, vmin
                    ↪ =0, vmax=255)
326             axes[2, i].set_title(f'校正后 - {channel} 通道')
327             axes[2, i].axis('off')
328             plt.colorbar(im, ax=axes[2, i], fraction=0.046, pad=0.04)
```

```
329
330         corrected_rgb = np.clip(corrected_data / 255.0, 0, 1)
331         axes[2, 3].imshow(corrected_rgb)
332         axes[2, 3].set_title('校正后 - RGB合成')
333         axes[2, 3].axis('off')
334
335         plt.tight_layout()
336         plt.show()
337
338
339 # 主函数
340 if __name__ == "__main__":
341     files = ["MathModel_Code\\data\\preprocess\\p3\\RedPicture.xlsx", "
        ↪ MathModel_Code\\data\\preprocess\\p3\\GreenPicture.xlsx", "
        ↪ MathModel_Code\\data\\preprocess\\p3\\BluePicture.xlsx"]
342
343     corrector = LEDColorCorrection()
344
345     for filepath in files:
346         corrector.load_excel_data(filepath)
347         correction_matrix = corrector.calibrate_correction_matrix()
348
349         print("\n评估校正效果:")
350         corrected_display, diff_before, diff_after = corrector.
            ↪ evaluate_correction()
351
352         corrector.visualize_results()
353
354         print("\n校正完成!")
```

D. 像素数据集