1 正交多项式作为基函数下的多项式投影

由于在第三次作业中已经使用了 Gauss 数值积分的方法作出离散投影，本次作业的改动之处为不再使用第三方库 Boost 库和 GSL 库，而是根据 **Golub-Welsch 算法**求解 Gauss 积分的积分节点和积分权重。

1.1 Gauss 积分

理论上构造 Gauss 积分的方法如下：

1. 作 $[a,b]$ 上权函数 $w(x)$ 的正交多项式 $P_n(x)$
2. 求出 $P_{n+1}(x)$ 的零点 $x_0, x_1, \ldots, x_n$
3. 计算
$$w_i = \int_a^b \prod_{j\neq i} \frac{x-x_i}{x_i-x_j}\, w(x)\, dx, \quad i = 0, \ldots, n$$
4. 得到
$$G_n[f] = \sum_{i=0}^{n} w_i f(x_i)$$

Golub-Welsch 算法的具体步骤如下：

Algorithm 1 Golub-Welsch 算法

Require:

- 递推系数：$\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$
- 递推系数：$\beta_0, \beta_1, \ldots, \beta_{n-2}$
- 零阶矩：$\mu_0 = \int_a^b w(x)\,\mathrm{d}x$

Ensure:

- 积分节点：$x_0, x_1, \ldots, x_{n-1}$
- 积分权重：$w_0, w_1, \ldots, w_{n-1}$

构造 $n \times n$ 对称 Jacobi 矩阵 J：

$$J = \begin{bmatrix} \alpha_0 & \beta_0 & & \\ \beta_0 & \alpha_1 & \ddots & \\ & \ddots & \ddots & \beta_{n-2} \\ & & \beta_{n-2} & \alpha_{n-1} \end{bmatrix}$$

求解特征值分解：$J = Q\Lambda Q^T$

提取特征值：$\mathbf{x} \leftarrow \mathrm{diag}(\Lambda)$

提取特征向量矩阵第一行：$q_{0,i} \leftarrow Q(0,i)$，其中 $i = 0, \ldots, n-1$

计算权重：$w_i \leftarrow \mu_0 \cdot q_{0,i}^2$，其中 $i = 0, \ldots, n-1$

return $\mathbf{x}, \mathbf{w}$

对于正交多项式，我们有递推关系

$$xp_k(x) = \beta_k p_{k+1}(x) + \alpha_k p_k(x) + \beta_{k-1} p_{k-1}(x)$$

据此构建 Jocobi 矩阵，它的特征值就是第 n 阶正交多项式 $P_n(x)$ 的零点，即积分节点。而使用 Jocobi 矩阵特征向量计算出的积分权重，和上面通过拉格朗日基函数计算出的积分权重是等价的。

实现 Golub-Welsch 算法的代码如下：

```
/// @brief 使用 Golub-Welsch 算法计算高斯积分的点和权重
/// @param n 积分点数量
/// @param alphas 雅可比矩阵的主对角线元素 (递推系数 alpha)
/// @param betas 雅可比矩阵的次对角线元素 (递推系数 beta)
/// @param mu0 权函数的零阶矩 ( w(x)dx)
/// @param points 输出的积分点
/// @param weights 输出的积分权重
void gauss_quadrature(
    int n,
    const std::vector<double> &alphas,
    const std::vector<double> &betas,
    double mu0,
    std::vector<double> &points,
    std::vector<double> &weights)
{
    if (n <= 0)
        return;

    // 构建 n x n 的雅可比矩阵 J
    Eigen::MatrixXd J = Eigen::MatrixXd::Zero(n, n);
    // 主对角线元素
    for (int i = 0; i < n; ++i)
    {
        J(i, i) = alphas[i];
    }
    // 次对角线元素
    for (int i = 0; i < n - 1; ++i)
    {
        J(i, i + 1) = betas[i];
        J(i + 1, i) = betas[i];
    }

    // 求解特征值
    Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> eigensolver(J);

    // 特征值是积分点
    Eigen::VectorXd eigenvalues = eigensolver.eigenvalues();
    points.assign(eigenvalues.data(), eigenvalues.data() + eigenvalues.size());
```

```
    // 权重的计算依赖于特征向量的第一个分量
    Eigen::MatrixXd eigenvectors = eigensolver.eigenvectors();
    weights.resize(n);
    for (int i = 0; i < n; ++i)
    {
        double first = eigenvectors(0, i);
        weights[i] = mu0 * first * first;
    }
}
```

对于 Legendre 多项式和 Hermite 多项式，只需要取不同的 α 和 β 即可。

1.2 Legendre 多项式

Legendre 多项式的 L2 误差如表1所示：

表 1: 不同函数在 Legendre 正交投影下的 L^2 误差

N	f_1	f_2	f_3	f_4
2	5.0088e-01	1.0202e-01	2.7579e-01	7.0706e-01
4	3.2354e-01	5.0984e-02	2.5962e-02	5.3022e-01
5	3.2354e-01	5.0984e-02	2.5962e-02	4.4175e-01
6	8.0286e-02	3.1840e-02	1.2347e-03	4.4175e-01
7	8.0286e-02	3.1840e-02	1.2347e-03	3.8641e-01
8	5.3275e-02	2.2264e-02	3.5557e-05	3.8641e-01
9	5.3275e-02	2.2264e-02	3.5557e-05	3.4764e-01
10	3.6787e-02	1.6677e-02	6.8785e-07	3.4764e-01
12	1.7895e-02	1.3083e-02	9.5643e-09	3.1852e-01
14	8.8870e-03	1.0611e-02	1.0026e-10	2.9560e-01

对 $f_1(x)$,$f_2(x)$,$f_4(x)$ 画 $log(error)$ 与 $logN$ 的坐标图，对 $f_3(x)$ 画 $log(error)$ 与 N 的坐标图，如图1所示：

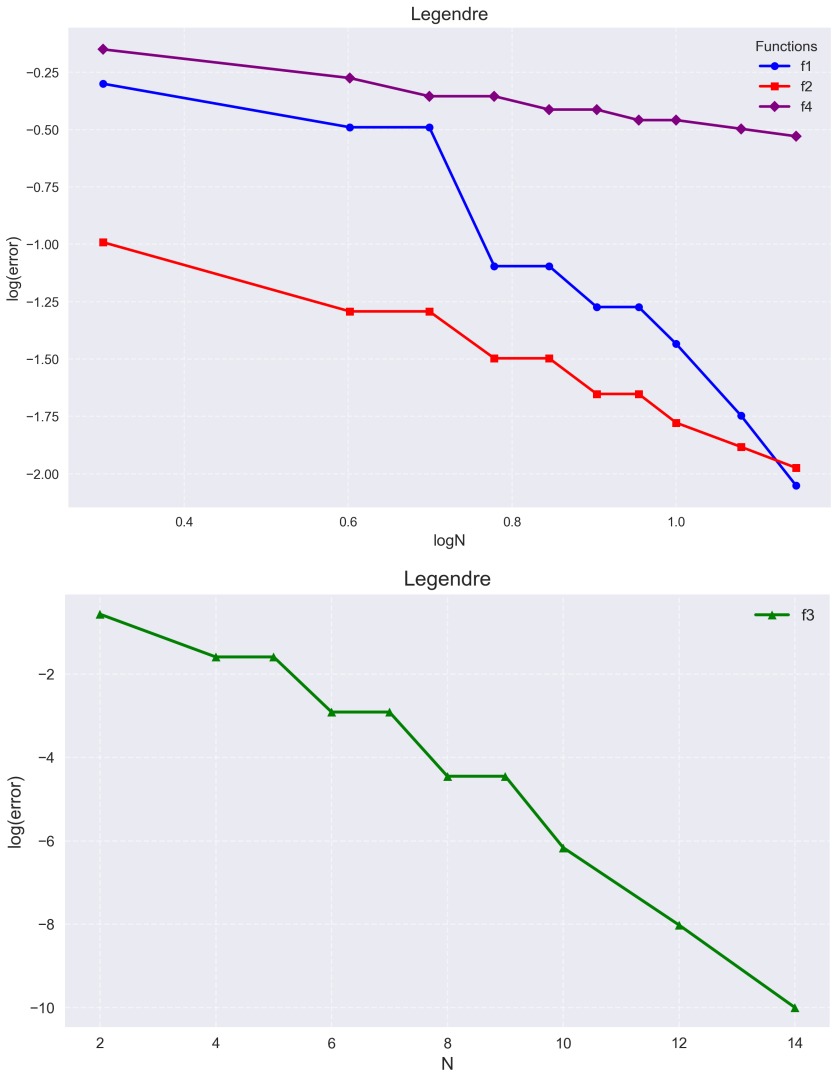

图 1: Legendre 投影的 L^2 误差

从图1可以看出，对于函数 $f_1(x) = |\sin(\pi x)|^3$,$f_2(x) = |x|$,$f_4(x) = sign(x)$，$log(error) = -klog(N)$，接近直线，这与理论分析相符合。并且，$|\sin(\pi x)|^3$ 在 $x = 0$ 处是二阶可导的，比 $|x|$ 更加光滑，因此坐标图中 $f_1(x)$ 的斜率更陡，收敛更快。

而 $f_3(x) = \cos(\pi x)$ 是解析的，因此 $f_3(x)$ 的坐标图中 $log(error) = -kN$，同样接近直线。

我们仍然画出 Legendre 多项式的投影与原函数的对比，如图2所示，可以看出随着正交多项式最高阶数 N 的增大，投影结果越来越接近原函数。

同时，由于 $f(x) = sign(x)$ 是间断的，因此图像中出现了 Gibbs 现象。

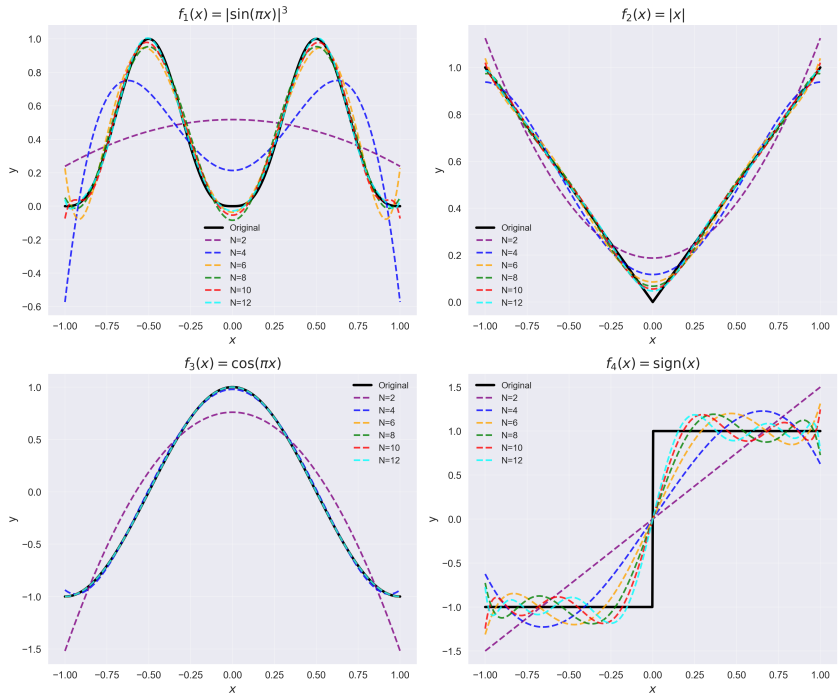

图 2: Legendre 投影与原函数

1.3 Hermite 多项式

Hermite 多项式的 L2 误差如表??所示:

表 2: 不同函数在 Hermite 正交投影下的 L^2 误差

N	f_1	f_2	f_3	f_4
2	4.8453e-01	1.9694e-01	8.4753e-01	7.9964e-01
4	4.8452e-01	1.2479e-01	6.3508e-01	6.7009e-01
5	4.8452e-01	1.2479e-01	6.3508e-01	6.0219e-01
6	4.8441e-01	9.3111e-02	3.8425e-01	6.0219e-01
7	4.8441e-01	9.3111e-02	3.8425e-01	5.5748e-01
8	4.8368e-01	7.4930e-02	1.9094e-01	5.5748e-01
9	4.8368e-01	7.4930e-02	1.9094e-01	5.2459e-01
10	4.8051e-01	6.2977e-02	7.9842e-02	5.2459e-01
12	4.7106e-01	5.4441e-02	2.8713e-02	4.9878e-01
14	4.5028e-01	4.8033e-02	9.2407e-03	4.7762e-01

画出 $log(error)$ 与 $logN$ 和 $log(error)$ 与 N 的坐标图如图3所示:

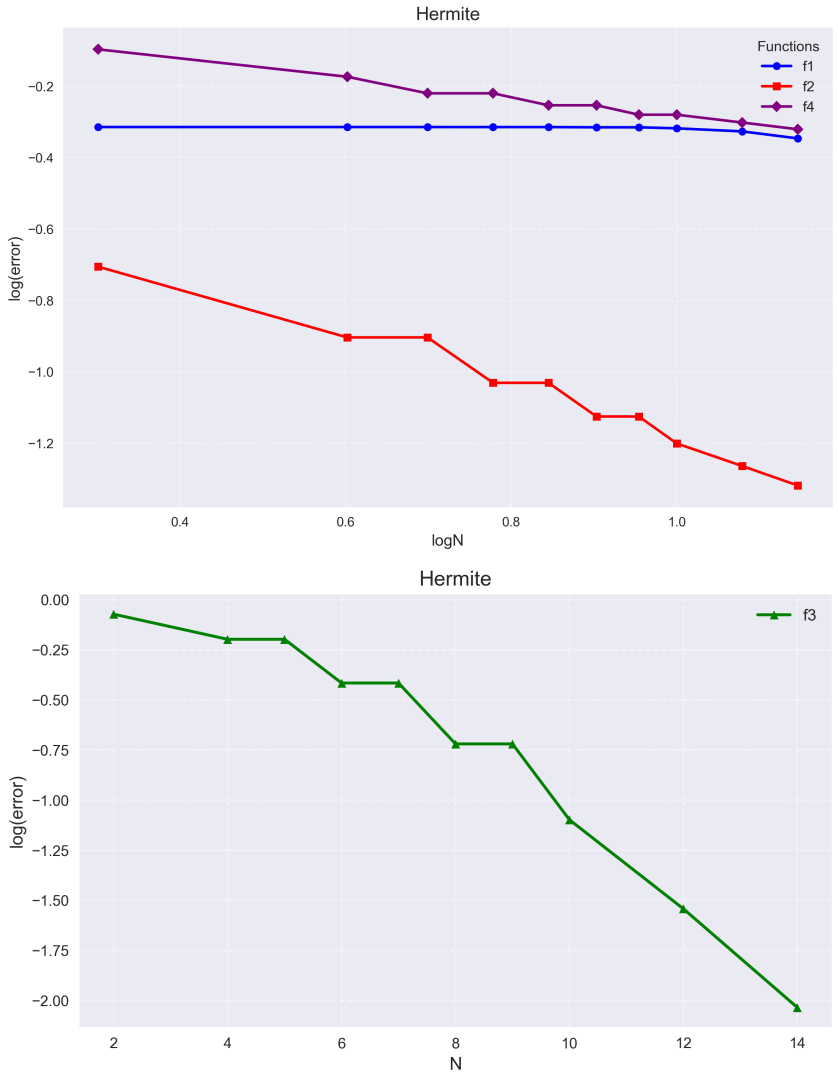

图 3: Hermite 投影的 L^2 误差

对于函数 $f(x) = |\sin(\pi x)|^3$,$f(x) = |x|$,$f(x) = sign(x)$，$log(error) = -klog(N)$，接近直线。而 $f(x) = \cos(\pi x)$ 是解析的，因此 $f_3(x)$ 的坐标图中 $log(error) = -kN$，同样接近直线。

但在以 Hermite 多项式作为基函数的情况下，$f_1(x)$ 的投影误差且随着 N 的增大几乎没有下降。这可能是因为权函数 $w(x) = e^{-x^2}$ 更关注在原点附近函数的行为，在远离原点处快速衰减，而 $f_1(x) = |\sin(\pi x)|^3$ 非解析，且光滑性较差的点（如 $x = 0, x = \pm 1$）恰好在原点附近被放大。

我们同样画出 Hermite 多项式的投影与原函数的对比，如图4所示，可以看出对 $f_1(x)$ 的投影与 $f_1(x)$ 差距较大。由于 $f(x) = sign(x)$ 是间断的，图像中出现了 Gibbs 现象。

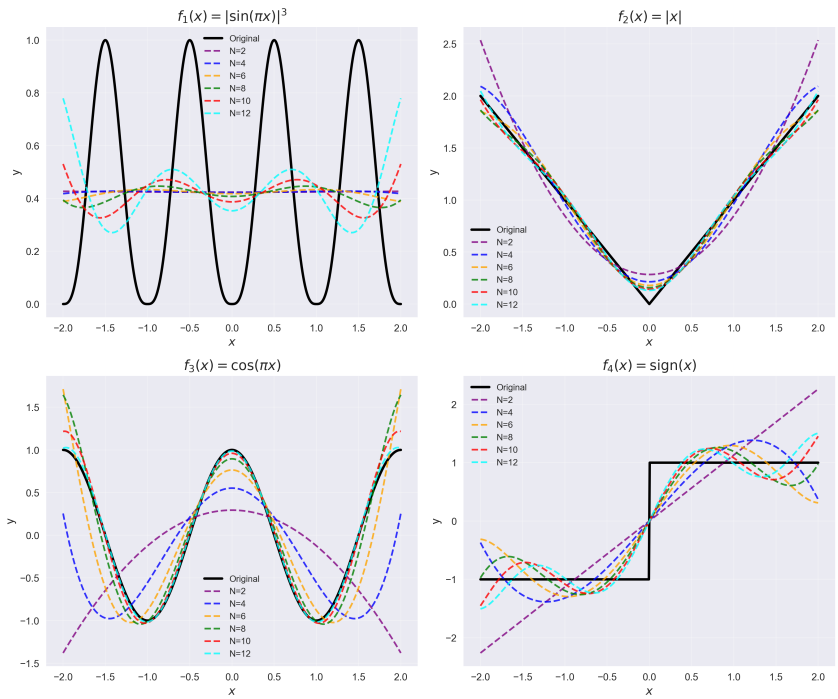

图 4: Hermite 投影与原函数

2 使用的第三方库

本次作业使用了 Eigen 库 (version 4.0) 来帮助求解矩阵的特征值和特征向量.