積分計算

考慮以下二元函數 $f(x, y)$ 及積分區域 $\boldsymbol{D}$：

$$f(x, y) = 3 \times \sin(8\pi x) \times \cos(8\pi y) + x + y + 1$$
$$\boldsymbol{D} = [2, 6] \times [2, 6]$$

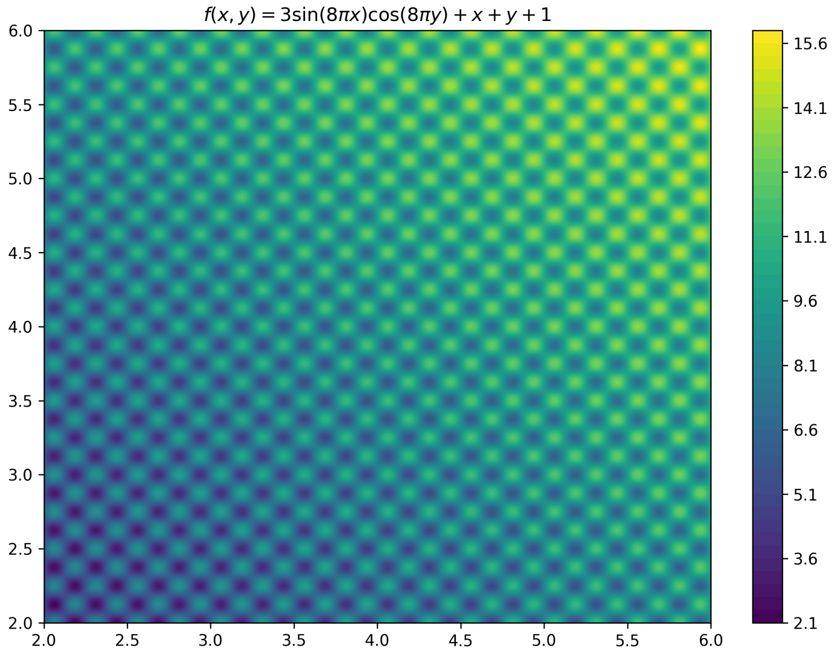

Figure 1: 函數 $f(x, y)$ 的等高線圖

解析解

注意到 $\sin(8\pi x)$ 與 $\cos(8\pi y)$ 的週期皆為 $\frac{1}{4}$，而積分區域 $\boldsymbol{D}$ 在 x 與 y 方向上長度皆為 4，恰好包含了 16 個完整的週期，因此積分結果中關於三角函數的部分會相互抵銷，即

$$\iint_{\boldsymbol{D}} 3\sin(8\pi x)\cos(8\pi y)\, dy\, dx = 0 \tag{1}$$

因此只需計算剩餘部分的積分：

$$\begin{aligned}
\iint_{\boldsymbol{D}} f(x, y)\, dy\, dx &= \iint_{\boldsymbol{D}} [3\sin(8\pi x)\cos(8\pi y) + x + y + 1]\, dy\, dx && (2)\\
&= \int_2^6 \int_2^6 (x + y + 1)\, dy\, dx && (3)\\
&= \int_2^6 \left[(x + 1)y + \frac{y^2}{2}\right]_2^6 dx && (4)\\
&= \int_2^6 \left[(x + 1)(6 - 2) + \frac{36 - 4}{2}\right] dx && (5)\\
&= \int_2^6 [4x + 20]\, dx && (6)\\
&= \left[2x^2 + 20x\right]_2^6 && (7)\\
&= 2 \times (36 - 4) + 20 \times (6 - 2) = 64 + 80 = 144 && (8)
\end{aligned}$$

所以 $f(x,y)$ 在區域 $\boldsymbol{D}$ 上積分的解析解為 144。

數值解：梯形法則

將區域 $\boldsymbol{D}$ 等分成 $n\times n$ 個小區域，則每個小區域的邊長為 $h=\frac{4}{n}$。根據梯形法則，雙重積分的數值解可表示為

$$\iint_{\boldsymbol{D}} f(x,y)\,dy\,dx \approx \frac{h^2}{4}\sum_{i=0}^{n}\sum_{j=0}^{n} c_{ij} f(x_i,y_j) \tag{9}$$

其中 $x_i=2+i\cdot h$，$y_j=2+j\cdot h$，且 c_{ij} 為

$$c_{ij}=\begin{cases}1, & \text{if } (i,j) \text{ 是四個角落的點}\\ 2, & \text{if } (i,j) \text{ 是在邊界上但不是角落的點}\\ 4, & \text{if } (i,j) \text{ 是內部的點}\end{cases} \tag{10}$$

```c
/**
 * 使用梯形法則計算 f(x, y) 在區域 [a, b] x [c, d] 上使用 n x m 分割的雙重積分值
 */
double trapezoid(double a, double b, double c, double d, int n, int m) {
    double hx = (b - a) / n, hy = (d - c) / m;
    double integral = 0.0;

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            double x = a + i * hx, y = c + j * hy;
            double c = 4.0;  // 預設為內部點
            if (i == 0 || i == n) c /= 2.0;  // 在 x 邊界上
            if (j == 0 || j == m) c /= 2.0;  // 在 y 邊界上
            integral += c * f(x, y);
        }
    }

    return integral * (hx * hy / 4.0);
}
```

使用此方法計算當 $n=4,8,16,32,64$ 時的數值解如下表（Table 1）所示。

n	數值解	絕對誤差	相對誤差
4	143.999999999999812	1.880995×10^{-13}	1.306247×10^{-15}
8	143.999999999999897	1.028344×10^{-13}	7.141278×10^{-16}
16	143.999999999999961	3.901046×10^{-14}	2.709060×10^{-16}
32	144.000000000000000	1.387779×10^{-17}	9.637353×10^{-20}
64	144.000000000000000	2.775558×10^{-17}	1.927471×10^{-19}

Table 1: 梯形法則積分結果與誤差比較（已知解析解 = 144）

Gaussian Quadrature

首先，我們將積分區域 $\boldsymbol{D}$ 切分成若干個小區域 (cell)，並對每個 cell 使用 $n\times n$ 個 Gaussian-Legendre 節點和其對應的權重來進行積分。最後，將所有 cells 的積分結果相加，即可得到整個區域的積分結果。

假設某一 cell 為 $[a,b]\times[c,d]$，則需要將 Gaussian 節點從標準區間 $[-1,1]$ 映射到該 cell 上。映射公式如下所示：

$$x(\xi)=\frac{b-a}{2}\xi+\frac{a+b}{2} \tag{11}$$

$$y(\eta) = \frac{d-c}{2}\eta + \frac{c+d}{2} \tag{12}$$

其中 ξ, η 為標準區間 $[-1, 1]$ 上的 Gaussian 節點，並且有 Jacobian 行列式

$$|\boldsymbol{J}| = \frac{(b-a)(d-c)}{4} \tag{13}$$

因此，該 cell 上的積分可以表示為

$$\iint_{\text{cell}} f(x,y)\,dy\,dx \approx \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j f(x(\xi_i), y(\eta_j))|\boldsymbol{J}| \tag{14}$$

其中 ξ_i 與 η_j 為 Legendre 多項式 $P_n(x)$ 的根，w_i 與 w_j 為對應的權重。

```cpp
/**
 * 使用 Gauss-Legendre 積分公式計算 f(x, y) 在區域 [a, b] x [c, d] 上的雙重積分值
 */
long double integrate_cell_gauss_legendre(long double a, long double b,
                                          long double c, long double d,
                                          const std::vector<long double>& roots,
                                          const std::vector<long double>& weights) {
    long double integral = 0.0L;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            // 將 Gaussian 節點轉換為區域 [a,b] x [c,d] 上的座標
            long double x = ((b - a) / 2.0L) * roots[i] + (a + b) / 2.0L;
            long double y = ((d - c) / 2.0L) * roots[j] + (c + d) / 2.0L;

            // 累加對應的權重和函數值
            integral += weights[i] * weights[j] * f(x, y);
        }
    }

    // 乘上 Jacobian 行列式
    return integral * ((b - a) / 2.0L) * ((d - c) / 2.0L);
}
```

最後，將所有 cell 的積分結果相加，即可得到整個區域 $\boldsymbol{D}$ 的積分結果

$$\iint_{\boldsymbol{D}} f(x,y)\,dy\,dx \approx \sum_{\text{cells}} \iint_{\text{cell}} f(x,y)\,dy\,dx \tag{15}$$

```cpp
/**
 * 使用 Gauss-Legendre 積分公式計算 f(x, y) 在區域 [a, b] x [c, d] 上
 * 使用 n 個節點和 meshs x meshs 分割的雙重積分值
 */
long double integrate_gauss_legendre(long double a, long double b,
                                     long double c, long double d,
                                     int n, int meshs) {
    // 取得 Legendre 多項式 P_n(x) 的根與權重
    auto [roots, weights] = legendre_roots_weights(n);
    // 計算每個網格單元的大小
    long double hx = (b - a) / meshs, hy = (d - c) / meshs;
    long double integral = 0.0L;

    // 對每個 cell 進行積分並累加結果
    for (int i = 0; i < meshs; i++) {
        for (int j = 0; j < meshs; j++) {
            // 計算每個 cell 的邊界
            long double cell_a = a + i * hx, cell_b = cell_a + hx;
            long double cell_c = c + j * hy, cell_d = cell_c + hy;

            // 計算該 cell 的積分並累加
            integral += integrate_cell_gauss_legendre(cell_a, cell_b, cell_c, cell_d, roots,
                weights);
        }
    }
```

```
    return integral;
}
```

在主程式中，我們可以設定多組不同的區塊數量 M 和 Gaussian 節點數量 N（亦即 Legendre 多項式的次數），並使用 Gauss-Legendre 積分公式來計算每一組參數下的積分結果。計算完成後，將其與解析解進行比較，並計算相應的誤差。此外，在計算過程中，還會記錄執行 100 次所花費的時間，以便評估不同參數組合下的效能表現。

```
std::vector<int> mesh_values = {1, 2, 4, 8, 16, 32, 64};  // 要測試的 mesh
std::vector<int> n_values = {1, 2, 3, 4, 5, 6, 7};         // 要測試的 n 值

for (auto&& m : mesh_values) {
    for (auto&& n : n_values) {
        ld result;
        auto start = std::chrono::high_resolution_clock::now(); // 開始計時

        for (int iter = 0; iter < 100; iter++)
            result = integrate_gauss_legendre(2.0L, 6.0L, 2.0L, 6.0L, n, m);

        auto end = std::chrono::high_resolution_clock::now();   // 結束計時
        auto duration = end - start;

        ld error = result - 144.0L;  // 已知解析解為 144

        std::cout << "mesh = " << m << ", n = " << n << ", result = " << result
                  << ", err = " << error << ", rel_err = " << error / 144.0L
                  << ", time = " << duration.count() / 100
                  << std::endl;
    }
    std::cout << "----------------------------------------" << std::endl;
}
```

成果

程式對於每組 M 和 N 計算出數值積分結果後，將其與解析解 144 進行比較，並計算出絕對誤差與相對誤差，同時記錄每組參數下的平均執行時間。執行結果請見下頁表格 Table 2，從表格中我們可以觀察到不同網格數量與 Gaussian 節點數量對積分結果精度與計算時間的影響，如下所示：

1. 隨著區塊數量 M 的增加，積分值逐漸接近解析解 144，絕對誤差也逐漸減小，最終相對誤差接近 $10^{-19} \sim 10^{-18}$ 的量級，相當於 `long double` 的機器精度限制。然而，當 $M \geq 16$ 時，誤差的變化趨於平緩，變化幅度變得不明顯。
2. 隨著區塊數量的增加，計算所需的執行時間顯著增長，從微秒級別增長到毫秒級別。
3. 在較小的區塊數量下，增加 Gaussian 節點數量 N 可以顯著提高積分的精度，減少誤差。然而，當區塊數量達到一定程度（例如 $M \geq 16$）後，進一步增加 N 對誤差的改善效果變得不顯著。
4. 增加節點數量 N 會導致執行時間增加，因為更多的節點意味著更多的計算步驟。

Mesh	N	高斯積分值	絕對誤差	相對誤差	執行時間
1	1	143.999999999999812	1.881×10^{-13}	1.30625×10^{-15}	1.241 µs
	2	143.999999999999897	1.03029×10^{-13}	7.15477×10^{-16}	1.685 µs
	3	143.999999999999927	7.34829×10^{-14}	5.10298×10^{-16}	3.321 µs
	4	143.999999999999990	1.02557×10^{-14}	7.122×10^{-17}	3.853 µs
	5	143.999999999999991	8.89566×10^{-15}	6.17754×10^{-17}	11.47 µs
2	1	143.999999999999812	1.88113×10^{-13}	1.30634×10^{-15}	1.31 µs
	2	143.999999999999975	2.53131×10^{-14}	1.75785×10^{-16}	2.152 µs
	3	143.999999999999917	8.29892×10^{-14}	5.76314×10^{-16}	4.23 µs
	4	143.999999999999938	6.20476×10^{-14}	4.30886×10^{-16}	5.941 µs
	5	143.999999999999978	2.15938×10^{-14}	1.49957×10^{-16}	13.23 µs
4	1	143.999999999999982	1.75832×10^{-14}	1.22105×10^{-16}	1.991 µs
	2	143.999999999999941	5.86198×10^{-14}	4.07082×10^{-16}	3.882 µs
	3	144.000000000000002	1.79023×10^{-15}	1.24322×10^{-17}	8.123 µs
	4	144.000000000000001	6.80012×10^{-16}	4.7223×10^{-18}	12.18 µs
	5	144.000000000000012	1.22263×10^{-14}	8.49051×10^{-17}	23.05 µs
8	1	144.000000000000025	2.5091×10^{-14}	1.74243×10^{-16}	11.06 µs
	2	143.999999999999919	8.06855×10^{-14}	5.60316×10^{-16}	12.38 µs
	3	144.000000000000006	6.17562×10^{-15}	4.28862×10^{-17}	24.27 µs
	4	144.000000000000000	2.77556×10^{-16}	1.92747×10^{-18}	58.72 µs
	5	144.000000000000000	4.71845×10^{-16}	3.2767×10^{-18}	64.84 µs
16	1	144.000000000000046	4.64351×10^{-14}	3.22466×10^{-16}	14.19 µs
	2	143.999999999999998	2.1233×10^{-15}	1.47451×10^{-17}	39.54 µs
	3	144.000000000000001	6.80012×10^{-16}	4.7223×10^{-18}	82.74 µs
	4	144.000000000000000	1.38778×10^{-17}	9.63735×10^{-20}	146.7 µs
	5	144.000000000000000	1.38778×10^{-17}	9.63735×10^{-20}	220.3 µs
32	1	144.000000000000000	2.77556×10^{-17}	1.92747×10^{-19}	53.9 µs
	2	144.000000000000000	1.38778×10^{-17}	9.63735×10^{-20}	151.9 µs
	3	144.000000000000000	2.77556×10^{-17}	1.92747×10^{-19}	332.1 µs
	4	144.000000000000000	2.77556×10^{-17}	1.92747×10^{-19}	579.5 µs
	5	144.000000000000000	1.38778×10^{-17}	9.63735×10^{-20}	859.9 µs
64	1	144.000000000000000	2.77556×10^{-17}	1.92747×10^{-19}	207.8 µs
	2	144.000000000000000	1.249×10^{-16}	8.67362×10^{-19}	603.8 µs
	3	144.000000000000000	8.32667×10^{-17}	5.78241×10^{-19}	1251 µs
	4	144.000000000000000	1.249×10^{-16}	8.67362×10^{-19}	2178 µs
	5	144.000000000000000	8.32667×10^{-17}	5.78241×10^{-19}	3332 µs

Table 2: Gauss-Legendre 積分結果與誤差比較（已知解析解 = 144）

H-refinement

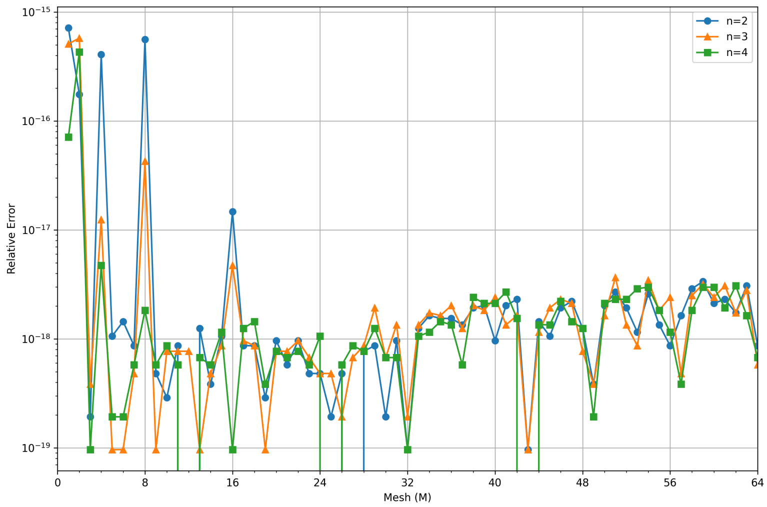

Figure 2: 分割空間成更多小區塊對相對誤差的影響

從上圖 Figure 2 可見，在區塊數量 M 較小時（如 $M \leq 16$），相對誤差的波動非常劇烈，這說明在只有少量區塊的情況下，積分結果對於區塊劃分的敏感度較高，可能會導致較大的誤差。

隨著 M 的增加，相對誤差顯著下降並趨於穩定，主要集中在 10^{-19} 到 10^{-18} 的範圍內，這表明增加區塊數量能有效提高幾分的穩定性和精度。特別是在 $M \geq 24$ 之後，使用不同取樣點數量 $N = 2, 3, 4$ 的三種方法，其相對誤差幾乎趨近一致，這說明了 h-refinement 已經將誤差逼近到 `long double` 的機器精度極限了。在到達此階段後，繼續增加 M 對誤差的改善效果將不再顯著。

P-refinement

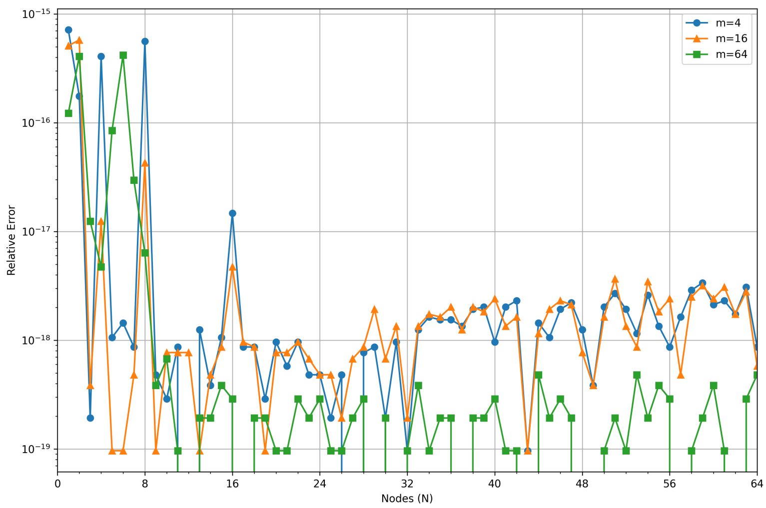

Figure 3: 固定 mesh，增加取樣點數量相對誤差的影響

從上圖 Figure 3 可見與 h-refinement 類似的，當取樣點數量 N 較少時（例如 $N \leq 3$），相對誤差的波動較大，這說明在取樣點數量不足的情況下，積分結果對於取樣點的選擇較為敏感。

當 N 增加到一定程度後（如 $N \geq 16$），在低區塊數量 $M = 16, 32$ 的情況下，相對誤差趨近於穩定在 10^{-18} 到 10^{-17} 左右。雖然將區塊數量從 $M = 32$ 提高到 64 時可以使相對誤差降低大約一個量級，但是這項改進是以執行時間大幅增加為代價的。這表示 p-refinement 在達到一定程度後，進一步增加 N 對誤差的改善效果不再顯著，而且會導致計算成本急劇攀升。

比較 h-refinement 與 p-refinement

在 Gaussian Quadrature 中，h-refinement 與 p-refinement 是兩種提升計算精度的關鍵策略，兩者各有優勢並在精度與效率之間扮演不同的角色。

H-refinement 透過將積分區域切割成更多、更細的區塊（M）來運作。這種方式能有效降低每個 cell 的局部誤差，並最終使整體精度逼近機器精度的下限。H-refinement 的優勢在於能提升最終可達到的最高準確度，但其收斂速度屬於代數級，且區塊數量 M 增加後，計算量亦會大幅成長，因此計算成本較高。

相對地，p-refinement 則是在每個 cell 內增加高斯取樣點（N），從而充分發揮高斯積分的指數級收斂性。這意味著只需少量增加取樣點 N，誤差便能迅速下降，使計算成本相對更具效率。然而，p-refinement 的精度極限會受到區塊解析度 M 的限制；當 M 不足時，p-refinement 無法突破由區塊數量下限所設定的誤差瓶頸。

兩者之間的依賴關係呈現出明確的分工：H-refinement 提供了最終準確度的上限，決定了誤差能否逼近機器精度；而 p-refinement 提供了快速收斂的能力，決定了在有限計算時間內能否有效降低誤差。

整體而言，在本題這種針對平滑函數進行積分的情況下，p-refinement 的高效率與快速收斂特性更具實際價值。因此，在大多數需要兼顧準確度與計算成本的情境中，p-refinement 的重要性略高於 h-refinement。然而，在追求極限精度的應用中，兩者仍需共同作用，才能達到最佳的數值表現。

計算 Legendre 多項式根與權重

在這次作業中，我參考了 Numpy 的相關原始碼實作，使用 Golub-Welsch 演算法來計算 Legendre 多項式 $P_n(x)$ 的根與權重。

在這裡我們使用二維 `vector` 來表示矩陣：

```
using Matrix = std::vector<std::vector<long double>>;
```

首先，我們需要建立一個大小為 $n \times n$ 的對稱三對角矩陣 $\boldsymbol{J}$，其為多項式 $P_n(x)$ 的伴隨矩陣。對於 Legendre 多項式，矩陣 $\boldsymbol{J}$ 的對角線元素皆為 0，而次對角線元素則為 $\beta_k = \frac{k}{\sqrt{4k^2-1}}$，其中 $k = 1, 2, \ldots, n-1$。矩陣 $\boldsymbol{J}$ 的形式如下所示：

$$\boldsymbol{J} = \begin{bmatrix} 0 & \beta_1 & 0 & \cdots & 0 & 0 \\ \beta_1 & 0 & \beta_2 & \cdots & 0 & 0 \\ 0 & \beta_2 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \beta_{k-1} \\ 0 & 0 & 0 & \cdots & \beta_{k-1} & 0 \end{bmatrix}_{n\times n} \tag{16}$$

```
/**
 * 產生 n 次 Legendre 多項式的伴隨矩陣
 */
Matrix legendre_companion_matrix(int n) {
    Matrix J(n, std::vector<long double>(n, 0.0L));

    // 填充 beta_k = k / sqrt(4k^2 - 1)
    for (int k = 1; k < n; k++) {
        long double beta = k / std::sqrt(4.0L * k * k - 1.0L);
```

```
        J[k][k - 1] = J[k - 1][k] = beta;
    }

    return J;
}
```

恰好矩陣 $\boldsymbol{J}$ 的特徵值即為 Legendre 多項式 $P_n(x)$ 的根，而對應的特徵向量則可用來計算權重。接著，我們可以使用線性代數的方法（如 QR 分解、Jacobi 方法等）來求解矩陣 $\boldsymbol{J}$ 的特徵值，即可獲得所需的所有根。其中我設定最大迭代次數為 $1000n$，並以長雙精度浮點數的機器精度作為收斂閾值。

```
/**
 * 使用 Jacobi 方法計算對稱矩陣的特徵值
 */
std::vector<long double> jacobi_eigen(const Matrix& _A) {
    const long double eps = LDBL_EPSILON;    // 收斂閾值
    const int max_iter = 1000 * n;  // 最大迭代次數
    // 這裡應該實作特徵值分解的數值方法
    // 返回特徵值陣列
}
```

由於我們使用 Jacobi 方法計算出初步的根後，這些根可能還不是非常精確，因此我們接下來使用牛頓法來修正，進一步提升根的精度。對於每個初始根 x_i，我們可以使用以下迭代公式進行更新：

$$x_i \leftarrow x_i - \frac{P_n(x_i)}{P_n'(x_i)} \tag{17}$$

其中 $P_n(x)$ 為 Legendre 多項式，可以透過遞迴關係計算：

$$P_0(x) = 1, \quad P_1(x) = x, \tag{18}$$

$$P_{k+1}(x) = \frac{(2k+1)xP_k(x) - kP_{k-1}(x)}{k+1}, \quad k = 1, 2, \ldots, n-1 \tag{19}$$

導數 $P_n'(x)$ 也可以透過以下關係計算：

$$P_n'(x) = \frac{n}{x^2-1}\left[xP_n(x) - P_{n-1}(x)\right] \tag{20}$$

```
/**
 * 計算 Legendre 多項式 P_n(x)
 */
ld legendre_value(int n, ld x) {
    if (n == 0) return 1.0L;  // P_0(x) = 1
    if (n == 1) return x;     // P_1(x) = x
    ld Pn_2 = 1.0L, Pn_1 = x, Pn;  // P_0(x), P_1(x)

    // 使用遞迴公式從 P_2(x) 計算到 P_n(x)
    for (int k = 2; k <= n; k++) {
        Pn = ((2.0L * k - 1.0L) * x * Pn_1 - (k - 1.0L) * Pn_2) / k;
        Pn_2 = Pn_1; Pn_1 = Pn;
    }
    return Pn;
}

/**
 * 計算 Legendre 多項式 P_n(x) 的導數 P_n'(x)
 */
ld legendre_derivative(int n, ld x) {
    if (n == 0) return 0.0L;  // P_0(x) = 1 -> P_0'(x) = 0
    if (n == 1) return 1.0L;  // P_1(x) = x -> P_1'(x) = 1
    ld Pn = legendre_value(n, x);
    ld Pn_1 = legendre_value(n - 1, x);
    return n / (x * x - 1.0L) * (x * Pn - Pn_1);
}
```

得到修正後的根後，我們可以計算對應的權重 w_i，其計算公式如下：

$$w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2} \tag{21}$$

由於 Legendre 多項式是偶對稱的，我們知道它的根是對稱的，即對於每個根 x_i，都有一個對應的根 $-x_i$。因此，在數值積分中，我們可以對計算出的根和權重進行對稱化：

- 對稱化根：對於每一對根 $(-x_i, x_i)$，我們可以將它們合併成一個對稱的表示。
- 對稱化權重：對於每一對根對應的權重，我們可以將它們進行平均，這樣保證積分結果對稱。

最後，為了確保 Gaussian Quadrature 的結果正確，我們需要對權重進行縮放。由於 Gaussian Quadrature 要求權重的總和等於 2，我們需要對權重進行正規化處理：

$$w_i \leftarrow \frac{w_i}{\sum w_i} \times 2 \tag{22}$$

範例程式碼如下所示：

```
/**
 * 計算 n 次 Legendre 多項式的根和權重
 */
std::pair<std::vector<ld>, std::vector<ld>> legendre_roots_weights(int n) {
    // first approximation of roots. We use the fact that the companion
    // matrix is symmetric in this case in order to obtain better zeros.
    Matrix J = legendre_companion_matrix(n);
    std::vector<ld> roots = jacobi_eigen(J);
    std::sort(roots.begin(), roots.end());  // 將 roots 由小排到大

    std::vector<ld> dy(n), df(n), fm(n), weights(n);
    ld max_df = 0, max_fm = 0;

    // improve roots by one application of Newton's method
    for (int i = 0; i < n; i++) {
        dy[i] = legendre_value(n, roots[i]);
        df[i] = legendre_derivative(n, roots[i]);
        roots[i] -= dy[i] / df[i];
        max_df = std::max(max_df, std::fabs(df[i]));
    }

    // compute the weights. We scale the factor to avoid possible numerical
    // overflow.
    for (int i = 0; i < n; i++) {
        fm[i] = legendre_value(n - 1, roots[i]);
        max_fm = std::max(max_fm, std::fabs(fm[i]));
    }

    for (int i = 0; i < n; i++)
        fm[i] /= max_fm, df[i] /= max_df;
    for (int i = 0; i < n; i++)
        weights[i] = 1.0L / (fm[i] * df[i]);

    // for Legendre we can also symmetrize the roots and weights
    std::vector<ld> w_copy = weights, r_copy = roots;
    ld w_sum = 0.0L;
    for (int i = 0; i < n; i++) {
        weights[i] = (w_copy[i] + w_copy[n - 1 - i]) / 2.0L;
        roots[i] = (r_copy[i] - r_copy[n - 1 - i]) / 2.0L;
        w_sum += weights[i];
    }

    // scale weights to get the right value
    for (int i = 0; i < n; i++)
        weights[i] *= 2.0L / w_sum;

    return {roots, weights};
}
```