微分方程数值解 - 第九章上机作业

樊睿强基数学 2001 班

2023 年 4 月 18 日

摘要

本项目实现了对一维规则区域（以 $[0,1]$ 为例）、二维规则区域（以 $[0,1]^2$ 为例）、非规则区域（以 $x=0, x=1, y=1, y=\frac{1}{16}\sin(\pi x)$ 所围成的区域为例）上 Poisson 方程的 Dirichlet 和 Neumann 边值问题的有限差分法求解。并分析了不同限制算子和插值算子的收敛速度和运行效率。

1 设计文档

1.1 稀疏矩阵类

若使用通常的二维数组存储矩阵，则时空效率都是 $O(n^{2d})$ 的，非常低效。因此采用稀疏矩阵存储。根据本章实际应用场景，因为每一行都有非零元素，所以我们用 `std::map` 来存储每行的非零元，这样比较方便处理且不会造成空间的浪费。

稀疏矩阵类只需实现和向量的乘法即可。

1.2 离散网格类

设计 `Discretor`，对区域 $\Omega=[0,1]^d, d=1,2$，右端项 f，边界条件 g 和网格数 n，构造出 d 维 Poisson 方程 $-\Delta u=f, u|_{\partial\Omega}=g$ 的离散矩阵和右端项。

1.3 算子函数

定义函数限制算子，输入 n 和一个 $(n+1)^d$ 维向量（细网格上的值），返回一个 $(\frac{n}{2}+1)^d$ 维向量（粗网格上的值）。

定义函数插值算子，输入 n 和一个 $(n+1)^d$ 维向量（粗网格上的值），返回一个 $(2n+1)^d$ 维向量（细网格上的值）。

1.4 多重网格类

设计 `Multigrid` 模板类，模板参数为维度 $d\in\{1,2\}$ 和边界条件 Cond。构造参数为微分方程的右端函数和边界函数。即类的每个实例只用于解一个确定的方程。边界条件由一个二进制数给出，二进制数的每一位代表一个点（边）处的边界条件，0 代表 Dirichlet，1 代表 Neumann。

定义 `Jacobi` 函数，输入 n，$(n+1)^d$ 维初始向量，离散矩阵 A，离散右端项 b 和迭代次数 T，返回 T 次迭代后的矩阵。

定义 `V-Cycle` 和 `FMG-Cycle` 递归函数，输入 n，$(n+1)^d$ 维初始向量，当前的离散矩阵 A 和右端项 b，迭代次数 ν_1,ν_2。分别按教材 Def 9.32 和 Def 9.34 实现。

定义 `Solve` 函数，输入 n，限制算子（`Injection` 或 `Full-Weighting`），插值算子（`Linear` 或 `Quadratic`），迭代方法（`V-Cycle` 或 `FMG`），ν_1，ν_2，ϵ，是否为测试（即是否知道解的具体表达式，若是则传入解的表达式）。在 Solve 函数中迭代使用 VC 或 FMG 迭代直到 Residual 的增量小于 ϵ，并报告其迭代次数和 Residual。如果是测试，则还报告其 Solution Error。

1.5 代码实现

稀疏矩阵类的设计：

```
#ifndef SPARSEDMATRIX
#define SPARSEDMATRIX
#include <bits/stdc++.h>
#include "Matrix.h"
using namespace std;

template<class type>
struct Sparsed_Matrix {
    int n, m;
    vector<map<int,type>> ele;
    Sparsed_Matrix(int n, int m): n(n), m(m) {ele.resize(n);}
    map<int,type>& operator[](int i) {
        return ele[i];
    }
    const map<int,type>& operator[](int i) const {
        return ele[i];
    }
    Colvec<type> operator*(const Colvec<type>& v)const{
        Colvec<type> res(n);
        for (int i = 0; i < n; ++ i)
            for (auto& [j, x] : ele[i])
                res[i] += v[j] * x;
        return res;
    }
    operator Matrix<type>() const {
        Matrix<type> res(n, m);
        for (int i = 0; i < n; ++ i)
            for (auto& [j, x] : ele[i])
                res[i][j] = x;
        return res;
    }
};
#endif
```

多重网格类的总体设计：

```
/*
    Cond:
        Boundary Condition. An Integer.
        The kth bit is 0 if the corresponding boundary condition is Dirichlet and 1 if that
             is Neumann.
*/
typedef unsigned int Cond_t;
// Restriction: Full Weighting and Injection.
enum Restriction_method {Full_Weighting, Injection};
// Interpolation: Linear and Quadratic.
enum Interpolation_method {Linear, Quadratic};
// Cycle: V-cycle and FMG.
enum Cycle_method {V_cycle, FMG};

// Definite the dim and boundary condition type as the template parameters.
template<int dim, Cond_t Cond_type>
```

```
class Multigrid {};
```

一维多重网格类的设计：

```
template <Cond_t Cond_type>
class Multigrid <1, Cond_type> {
private:
    double w;                                   // w is the release coefficient.
    const Function<double>& f;                  // The rhs function.
    const Function<double>& g;                  // The boundary function.
    Colvec<double> (*Restriction) (int, const Colvec<double>&);
    Colvec<double> (*Interpolation) (int ,const Colvec<double>&);
    map<int, Discretor<1, Cond_type>> D;        // Discretors for different grids.
    Colvec<double> sol;                         // The solution vector.
public:
    Multigrid(const Function<double>& f, const Function<double>& g):
        w(2.0/3), f(f), g(g) {}
private:
    Colvec<double> Jacobi(int n, Sparsed_Matrix<double>& A, Colvec<double>& b, const Colvec
        <double>& v0, int T) {
        Colvec<double> v(v0);
        Sparsed_Matrix<double> trns(n+1, n+1);
        Colvec<double> c(n+1);
        for (int i = 0; i <= n; ++ i) {
            double dii = A[i][i];
            for (auto & [j, x] : A[i])
                if (j != i) trns[i][j] = -x / dii;
            c[i] = b[i] / dii;
        }
        Colvec<double> u(n+1);
        for (int i = 0; i < T; ++ i) {
            u = trns * v + c;
            v = w * u + (1-w) * v;
        }
        return v;
    }

    // v: Initial guess.
    // A: Discrete matrix for grid number n.
    // b: Discrete rhs for grid number n.
    Colvec<double> VC(int n, Sparsed_Matrix<double>& A, Colvec<double>& b, Colvec<double>&
        v0, int T1, int T2) {
        Colvec<double> v = Jacobi(n, A, b, v0, T1);
        if (n <= 2) return Jacobi(n, A, b, v, T2);
        Colvec<double> c = Restriction(n, b - A * v);
        if (!D.count(n/2)) D.insert({n/2, Discretor<1, Cond_type>(n/2, f, g)});
        Colvec<double> zero(n/2+1);
        Colvec<double> v1 = VC(n/2, D[n/2].coef, c, zero, T1, T2);
        v = v + Interpolation(n/2, v1);
        return Jacobi(n, A, b, v, T2);
    }
    Colvec<double> FMGC(int n, Sparsed_Matrix<double>& A, Colvec<double>& b, int T1, int T2
        ) {
        Colvec<double> zero(n+1);
        if (n <= 2) return VC(n, A, b, zero, T1, T2);
```

```
49          Colvec<double> c = Restriction(n, b);
50          if (!D.count(n/2)) D.insert({n/2, Discretor<1, Cond_type>(n/2, f, g)});
51          Colvec<double> v = Interpolation(n/2, FMGC(n/2, D[n/2].coef, c, T1, T2));
52          return VC(n, A, b, v, T1, T2);
53      }
54  public:
55      // n, the finest grid, must be a power of 2.
56      // eps is the upper bound of the residual.
57      // T1 and T2 are the Two-Grid iteration times specified by the user.
58      void Solve(int n, Restriction_method Restriction_type, Interpolation_method
            Interpolation_type, Cycle_method Cycle_type, int T1 = 5, int T2 = 5, double eps = 1
            e-8, bool is_test = 0, const Function<double>& real = _0<double>()) {
59          if (Restriction_type == Injection) Restriction = Injection_Restriction_1D;
60          else if (Restriction_type == Full_Weighting) Restriction =
                Full_Weighting_Restriction_1D;
61          if (Interpolation_type == Linear) Interpolation = Linear_Interpolation_1D;
62          else if (Interpolation_type == Quadratic) Interpolation =
                Quadratic_Interpolation_1D;
63          if (!D.count(n)) D.insert({n, Discretor<1, Cond_type>(n, f, g)});
64          Colvec<double> zero(n+1);
65          Colvec<double> rhs = D[n].rhs;
66          sol = zero;
67          int iter = 0;
68          if (Cycle_type == V_cycle) {
69              while (1) {
70                  ++ iter;
71                  Colvec<double> dta = VC(n, D[n].coef, rhs, zero, T1, T2);
72                  sol = sol + dta;
73              //  cout << iter << ' ' << vert_2(D[n].coef * dta) / sqrt(n+1) << endl;
74                  if (vert_2(D[n].coef * dta) / sqrt(n+1) < eps) break;
75                  rhs = rhs - D[n].coef * dta;
76              }
77          }
78          else {
79              while (1) {
80                  ++ iter;
81                  Colvec<double> dta = FMGC(n, D[n].coef, rhs, T1, T2);
82                  sol = sol + dta;
83              //  cout << iter << ' ' << vert_2(D[n].coef * dta) / sqrt(n+1) << endl;
84                  if (vert_2(D[n].coef * dta) / sqrt(n+1) < eps) break;
85                  rhs = rhs - D[n].coef * dta;
86              }
87          }
88          cout << "Dimension : " << 1 << endl;
89          cout << "Condition : " << Cond_type << endl;
90          cout << "Grid Number : " << n << endl;
91          cout << "Restriction : " << (Restriction_type == Injection ? "Injection" : "Full
                Weighting") << endl;
92          cout << "Interpolation : " << (Interpolation_type == Linear ? "Linear" : "Quadratic
                ") << endl;
93          cout << "Cycle : " << (Cycle_type == V_cycle ? "V_cycle" : "FMG") << endl;
94          cout << "Iteration times : " << T1 << ", " << T2 << endl;
95          cout << "Cycle times : " << iter << endl;
96          Colvec<double> e = D[n].coef * sol - D[n].rhs;
97          cout << "Residual Error in L_1 : " << vert_1(e) / (n+1) << endl;
```

```
        cout << "Residual Error in L_2 : " << vert_2(e) / sqrt(n+1) << endl;
        cout << "Residual Error in L_inf : " << vert_inf(e) << endl;
        if (is_test) {
            double h = 1.0 / n;
            double E1 = 0, E2 = 0, Einf = 0, C = 0;
            // Assume the average of solution be the arbitary constant.
            if (Cond_type == 3) {
                for (int i = 0; i <= n; ++ i) C += sol[i] - real(i*h);
                C /= (n+1);
            }
            for (int i = 0; i <= n; ++ i) {
                double ei = fabs(sol[i] - C - real(i*h));
                E1 += fabs(ei);
                E2 += ei * ei;
                Einf = max(Einf, ei);
            }
            E1 /= n+1, E2 /= n+1, E2 = sqrt(E2);
            cout << "Solution Error in L_1 : " << E1 << endl;
            cout << "Solution Error in L_2 : " << E2 << endl;
            cout << "Solution Error in L_inf : " << Einf << endl;
        }
    }
};
```

二维多重网格类只需在一维的基础上稍加修改即可，这里略去，详见程序代码。

以上设计文档中，稀疏矩阵类、多重网格类的结构对于不同维数和不同区域都是类似的。所以在项目框架设计完成后，对于不同的区域，只需着重设计离散网格、限制算子和插值算子即可。

1.6 测试函数

分别使用函数 $u(x) = \exp\{\sin(x)\}$ 和 $u(x, y) = \exp\{y + \sin(x)\}$ 进行一维和二维 Poisson 方程的测试。

2 一维规则区域 Poisson 方程的求解

$$-u_{xx} = f, u \in \Omega = [0, 1]. \tag{1}$$

2.1 差分格式

将 $[0, 1]$ 用等距的 n 个网格划分，每个小网格的边长记为 $h = \frac{1}{n}$。

将求解未知函数 u 的问题转化为求解 u 在网格点 $u(ih)$ （$0 \le i \le n$）处的值的问题。

进而可以通过插值等方法求出 u 在区域内任意一点的值。

根据 Poisson 方程的差分格式，有

$$\frac{1}{h^2}(2u(ih) - u((i-1)h) - u((i+1)h)) = f(ih, jh). \tag{2}$$

其中 $1 \le i \le n-1$。

这样就得到了所有内部网格点的差分格式。边界点的差分格式将通过具体的边界条件给出。

2.2 Dirichlet 边界条件

$$u|_{\partial\Omega} = g. \tag{3}$$

将边界条件离散到矩形边界的格点上，有

$$u(0) = g(0), u(1) = g(1). \tag{4}$$

将边界条件与内部网格点的差分格式联立，求解关于各网格点函数值的线性方程组即可。

根据教材 Exercise 7.40 可知，该差分格式的误差为 $O(h^2)$。

2.3 Neumann 边界条件

$$\frac{\partial u}{\partial n}|_{\partial\Omega} = g. \tag{5}$$

根据 Example 6.38 的一阶差分格式可得：

$$\frac{1}{2h}(-3u(0) + 4u(h) - u(2h)) = g(0), \tag{6}$$

$$\frac{1}{2h}(-3u(1) + 4u(1-h) - u(1-2h)) = g(1), \tag{7}$$

将边界条件与内部网格点的差分格式联立，得到的方程组是奇异的（恰有一个冗余方程），这个结果恰和 Neumann 边界条件解中的任意常数 C 对应。

在第七章用高斯消元解这个方程组时，需要引入新的方程来保证它有唯一解。但因为本章的迭代法不需要方程组的非奇异性，因此可不做处理，最终的解一定会收敛到解空间内。

2.4 限制算子

$$I_h^{2h} : \mathbb{R}^{n+1} \to \mathbb{R}^{\frac{n}{2}+1} \tag{8}$$

2.4.1 嵌入算子

直接将细网格上和粗网格对应的点作为粗网格上该点的估计。

$$v_j^{2h} = v_{2j}^h. \tag{9}$$

2.4.2 全加权算子

将细网格不在粗网格上的点的权值均分加到相邻的两个点上。再将所有点的权值除 2。

$$\begin{aligned} v_0^{2h} &= v_0^h, \\ v_n^{2h} &= v_{2n}^h, \\ v_j^{2h} &= \frac{1}{4}(v_{2j-1}^h + 2v_{2j}^h + v_{2j+1}^h), 1 \le j \le n-1. \end{aligned} \tag{10}$$

2.5 插值算子

$$I_h^{\frac{h}{2}} : \mathbb{R}^{n+1} \to \mathbb{R}^{2n+1}. \tag{11}$$

2.5.1 线性插值算子

对于在细网格而不在粗网格上的点 P，用和 P 相邻的两个点做线性插值，以插值多项式在 P 处的值作为估计。

$$\begin{aligned} v_{2j}^{\frac{h}{2}} &= v_j^h, \\ v_{2j+1}^{\frac{h}{2}} &= \frac{1}{2}(v_j^h + v_{j+1}^h). \end{aligned} \tag{12}$$

2.5.2 二次插值算子

对于在细网格而不在粗网格上的点 P，用和 P 相邻的三个点做二次插值，以插值多项式在 P 处的值作为估计。

$$\begin{aligned} v_{2j}^{\frac{h}{2}} &= v_j^h, \\ v_{2j+1}^{\frac{h}{2}} &= \frac{1}{8}(3v_j^h + 6v_{j+1}^h - v_{j+2}^h), 1 \le j < \frac{n}{2}, \\ v_{2j+1}^{\frac{h}{2}} &= \frac{1}{8}(3v_{j+1}^h + 6v_j^h - v_{j-1}^h), \frac{n}{2} \le j < n. \end{aligned} \tag{13}$$

2.6 代码实现

离散网格的代码如下：

```
template <Cond_t Cond_type>
class Discretor<1, Cond_type> {
private:
    int n;                      // The grid number.
    double h;                   // The grid size.
    const Function<double>& f;  // The rhs function.
    const Function<double>& g;  // The boundary function.
public:
    Sparsed_Matrix<double> coef;    // The coefficients.
    Colvec<double> rhs;             // The right terms.
private:
    void Normal_Laplace_Discretor_1D(int i0, int i1, int i2) {
        coef[i0][i0] = 2 / (h*h);
        coef[i0][i1] = -1 / (h*h);
        coef[i0][i2] = -1 / (h*h);
        rhs[i0] = f(i0*h);
    }
    void Normal_Dirichlet_Discretor_1D(int i0) {
        coef[i0][i0] = 1;
        rhs[i0] = g(i0*h);
    }
    void Normal_Neumann_Discretor_1D(int i0, int i1, int i2) {
        coef[i0][i0] = -1.5 / h;
        coef[i0][i1] = 2 / h;
        coef[i0][i2] = -0.5 / h;
        rhs[i0] = g(i0*h);
    }
public:
    Discretor() : n(n), f(_0<double>()), g(_0<double>()), coef(0, 0), rhs(0) {}
    Discretor(int n, const Function<double>& f, const Function<double>& g)
        :n(n), h(1.0/n), f(f), g(g), coef(n+1, n+1), rhs(n+1) {
```

```
        h = 1.0/n;
        if (Cond_type & 1) Normal_Neumann_Discretor_1D(0, 1, 2);
        else Normal_Dirichlet_Discretor_1D(0);
        if (Cond_type & 2) Normal_Neumann_Discretor_1D(n, n-1, n-2);
        else Normal_Dirichlet_Discretor_1D(n);
        for (int i = 1; i < n; ++ i) Normal_Laplace_Discretor_1D(i, i-1, i+1);
    }
};
```

各种算子代码如下：

```
// Restriction for 1D.
// Input: A vector with n elements.
// Output: A vector with n/2 elements.
Colvec<double> Injection_Restriction_1D(int n, const Colvec<double>& v) {
    Colvec<double> u(n/2+1);
    for (int i = 0; i <= n/2; ++ i)
        u[i] = v[i*2];
    return u;
}
Colvec<double> Full_Weighting_Restriction_1D(int n, const Colvec<double>& v) {
    Colvec<double> u(n/2+1);
    u[0] = v[0], u[n/2] = v[n];
    for (int i = 1; i <= n/2-1; ++ i)
        u[i] = (v[i*2-1] + 2*v[i*2] + v[i*2+1]) / 4;
    return u;
}

// Interpolation for 1D.
// Input: A vector with n elements.
// Output: A vector with n*2 elements.
Colvec<double> Linear_Interpolation_1D(int n, const Colvec<double>& v) {
    Colvec<double> u(n*2+1);
    for (int i = 0; i < n; ++ i) {
        u[i*2] = v[i];
        u[i*2+1] = (v[i] + v[i+1]) / 2;
    }
    u[n*2] = v[n];
    return u;
}
Colvec<double> Quadratic_Interpolation_1D(int n, const Colvec<double>& v) {
    Colvec<double> u(n*2+1);
    for (int i = 0; i < n-1; ++ i) {
        u[i*2] = v[i];
        u[i*2+1] = (3*v[i] + 6*v[i+1] - v[i+2])/ 8;
    }
    u[n*2-2] = v[n-1];
    u[n*2-1] = (-v[n-2] + 6*v[n-1] + 3*v[n]) / 8;
    u[n*2] = v[n];
    return u;
}
```

2.7 数据测试

一维多重网格类的测试程序为`main1.cpp`。

运行测试程序时，从命令行中依次输入 n，限制算子，插值算子，迭代方式，ν_1, ν_2。

```cpp
//main1.cpp
#include<bits/stdc++.h>
#include "Function.h"
#include "multigrid.h"
using namespace std;

class U : public Function<double> {
public:
    virtual double operator ()(const double& x) const {
        return exp(sin(x));
    }
    virtual double d(const double& x, const int& k = 1) const {
        if (k == 1) return cos(x) * exp(sin(x));
        if (k == 2) return (-sin(x) + cos(x)*cos(x)) * exp(sin(x));
        throw 0;
    }
}u;

class F : public Function<double> {
public:
    Function<double>& u;
    F(Function<double>& u) : u(u) {}
    virtual double operator()(const double& x) const {
        return -u.d(x, 2);
    }
};

typedef unsigned int Cond_t;
template<Cond_t Cond_type>
class G : public Function<double> {
public:
    Function<double>& u;
    G(Function<double>& u) : u(u) {}
    virtual double operator()(const double& x) const {
        if (x == 0){
            if (Cond_type & 1) return u.d(0);
            else return u(0);
        }
        if (x == 1){
            if (Cond_type & 2) return -u.d(1);
            else return u(1);
        }
        cerr << "Undefined!" << endl;
        exit(-1);
    }
};

int main(int argc, char** argv){
    int n = atoi(argv[1]);
    const Restriction_method i1 = string(argv[2]) == "I" ? Injection : Full_Weighting;
    const Interpolation_method i2 = string(argv[3]) == "L" ? Linear : Quadratic;
    const Cycle_method i3 = string(argv[4]) == "V" ? V_cycle : FMG;
    int T1 = atoi(argv[5]), T2 = atoi(argv[6]);
```

```
    auto Solver_D = Multigrid<1, 0>(F(u), G<0>(u));
    Solver_D.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
    auto Solver_N = Multigrid<1, 3>(F(u), G<3>(u));
    Solver_N.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
    auto Solver_M = Multigrid<1, 1>(F(u), G<1>(u));
    Solver_M.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
}
```

因为可能的输入组合数量过多（3 种边界、2 种限制算子、2 种插值算子、2 种迭代方法、5 个网格规模，共 120 种组合），所以这里只选取部分输入进行测试。

固定 $\epsilon = 10^{-8}$。

根据多重网格的收敛定理，$\nu_1 = \nu_2 = 2$ 时即可有效过滤掉高频波。但 ν_1, ν_2 与迭代次数之间有非线性的负相关性，实际应用时取 $\nu_1 = \nu_2 = 5$ 最佳。

表中“残差”和“误差”是无穷范数意义下的。其他范数（一范数和二范数）意义下的残差和误差见测试结果`test1.txt`。

注：一维情形下即使 n 达到最大的 512，运行时间也不超过 10 毫秒，因此不进行时间效率对比。

网格大小	边界条件	限制算子	插值算子	迭代方法	残差	误差	迭代次数
32	Dirichlet	Injection	Linear	V-Cycle	6.04e-11	4.94e-05	5
	Dirichlet	Full-Weighting	Linear	V-Cycle	3.27e-11	4.94e-05	8
	Dirichlet	Full-Weighting	Quadratic	V-Cycle	6.86e-12	4.94e-05	7
	Dirichlet	Injection	Linear	FMG	1.20e-12	4.94e-05	5
	Dirichlet	Full-Weighting	Linear	FMG	1.61e-11	4.94e-05	4
	Dirichlet	Full-Weighting	Quadratic	FMG	1.17e-12	4.94e-05	4
	Neumann	Injection	Linear	V-Cycle	2.22e-03	4.08e-04	13
	Neumann	Full-Weighting	Linear	V-Cycle	2.23e-03	4.08e-04	14
	Neumann	Full-Weighting	Quadratic	V-Cycle	2.22e-03	4.08e-04	13
	Neumann	Injection	Linear	FMG	2.25e-03	4.08e-04	6
	Neumann	Full-Weighting	Linear	FMG	2.26e-03	4.08e-04	6
	Neumann	Full-Weighting	Quadratic	FMG	2.26e-03	4.08e-04	6
	Mixed	Injection	Linear	V-Cycle	3.08e-09	1.63e-04	15
	Mixed	Full-Weighting	Linear	V-Cycle	4.56e-09	1.63e-04	15
	Mixed	Full-Weighting	Quadratic	V-Cycle	4.55e-09	1.63e-04	14
	Mixed	Injection	Linear	FMG	1.32e-10	1.63e-04	6
	Mixed	Full-Weighting	Linear	FMG	1.86e-10	1.63e-04	6
	Mixed	Full-Weighting	Quadratic	FMG	2.14e-11	1.63e-04	6
64	Dirichlet	Full-Weighting	Quadratic	V-Cycle	2.64e-11	1.24e-05	7
	Dirichlet	Full-Weighting	Quadratic	FMG	2.16e-11	1.24e-05	3
	Neumann	Full-Weighting	Quadratic	V-Cycle	6.95e-04	1.02e-04	13
	Neumann	Full-Weighting	Quadratic	FMG	7.08e-04	1.02e-04	6
	Mixed	Full-Weighting	Quadratic	V-Cycle	2.70e-09	4.34e-05	15
	Mixed	Full-Weighting	Quadratic	FMG	2.26e-10	4.34e-05	5
128	Dirichlet	Full-Weighting	Quadratic	V-Cycle	1.01e-10	3.09e-06	7
	Dirichlet	Full-Weighting	Quadratic	FMG	1.72e-11	3.09e-06	3
	Neumann	Full-Weighting	Quadratic	V-Cycle	2.16e-04	2.54e-05	14
	Neumann	Full-Weighting	Quadratic	FMG	2.21e-04	2.54e-05	6
	Mixed	Full-Weighting	Quadratic	V-Cycle	5.94e-09	1.12e-05	15
	Mixed	Full-Weighting	Quadratic	FMG	5.43e-11	1.12e-05	5
	Dirichlet	Full-Weighting	Quadratic	V-Cycle	4.67e-10	7.73e-07	7

	Dirichlet	Full-Weighting	Quadratic	FMG	5.50e-10	7.73e-07	3
	Neumann	Full-Weighting	Quadratic	V-Cycle	6.73e-05	6.34e-06	14
	Neumann	Full-Weighting	Quadratic	FMG	6.88e-05	6.34e-06	6
	Mixed	Full-Weighting	Quadratic	V-Cycle	1.21e-08	2.85e-06	15
	Mixed	Full-Weighting	Quadratic	FMG	8.34e-10	2.85e-06	4
512	Dirichlet	Full-Weighting	Quadratic	V-Cycle	5.99e-10	1.93e-07	8
	Dirichlet	Full-Weighting	Quadratic	FMG	2.93e-10	1.93e-07	3
	Neumann	Full-Weighting	Quadratic	V-Cycle	2.09e-05	1.58e-06	15
	Neumann	Full-Weighting	Quadratic	FMG	2.14e-05	1.58e-06	5
	Mixed	Full-Weighting	Quadratic	V-Cycle	5.55e-09	7.17e-07	16
	Mixed	Full-Weighting	Quadratic	FMG	3.83e-10	7.18e-07	4

2.8 误差分析（在这里回答讲义中 C 题）

从上表可以看出：

- 当 ϵ 取到 10^{-8} 时，解的误差已经收敛，即使残差的数量级有明显差异，误差也大致相等。
- 对于同样的 Cycle，用不同的限制算子和插值算子并没有明显的效率差异。甚至复杂的算子有时反而收敛更慢。
- FMG 的收敛速度明显快于 V-Cycle。
- 收敛速度和 n 无关。事实上，它只和 ϵ, ν_1, ν_2 有关。
- 对于纯 Neumann 边界条件，因为矩阵欠定，残差不收敛到 0，但会收敛到一个小常数。
- 对于每个边界条件，算法的收敛阶都是二阶的。n 每增加 2 倍，误差变为原来的 $\frac{1}{4}$。

对 $n = 512$ 的最细网格，将 ϵ 依次降为 $10^{-9}, 10^{-10}, 10^{-11}, 10^{-12}, 10^{-13}$，残差随之降低，但误差仍为 $1.93e-07$。

为什么 ϵ 的数量级降低，总体误差的数量级不变？

多重网格法求解的实际上是离散的差分方程组，而不是原来的 Poisson 方程。而离散方程组相对 Poisson 方程本身就有一个二阶的离散误差。

当 $n = 512, h = \frac{1}{512}$ 时，$h^2 = \frac{1}{262144}$。

粗略估计 $\|u^{(4)}\|_\infty \leq 12$，因此 $\|\tau\|_2 \leq \frac{1}{262144} \approx 4 \times 10^{-6}$。

当 ϵ 取 10^{-8} 时，误差的主要来源已经是 LTE。因此再降低 ϵ 不会使总体误差明显减小。

3 二维规则区域 Poisson 方程的求解

$$-u_{xx} - u_{yy} = f, u \in \Omega = [0,1]^2 \tag{14}$$

3.1 差分格式

将 $[0,1]^2$ 用等距的 $n \times n$ 网格划分，每个小网格的边长记为 $h = \frac{1}{n}$。

将求解未知函数 u 的问题转化为求解 u 在网格点 $u(ih, jh)$ （$0 \leq i \leq n, 0 \leq j \leq n$）处的值的问题。

进而可以通过插值等方法求出 u 在区域内任意一点的值。

根据 Poisson 方程的差分格式，有

$$\frac{1}{h^2}(4u(ih,jh)-u((i-1)h,jh)-u((i+1)h-jh)-u(ih,(j-1)h)-u(ih,(j+1)h))=f(ih,jh). \quad (15)$$

其中 $1\le i,j\le n-1$。

这样就得到了所有内部网格点的差分格式。边界点的差分格式将通过具体的边界条件给出。

3.2 Dirichlet 边界条件

$$u|_{\partial\Omega}=g \quad (16)$$

将边界条件离散到矩形边界的格点上，有

$$u(ih,0)=g(ih,0),0\le i\le n, \quad (17)$$
$$u(ih,1)=g(ih,1),0\le i\le n, \quad (18)$$
$$u(0,jh)=g(0,jh),0\le j\le n, \quad (19)$$
$$u(1,jh)=g(1,jh),0\le j\le n. \quad (20)$$

将边界条件与内部网格点的差分格式联立，求解关于各网格点函数值的线性方程组即可。

根据教材 Exercise 7.40 可知，该差分格式的误差为 $O(h^2)$。

3.3 Neumann 边界条件

$$\frac{\partial u}{\partial n}|_{\partial\Omega}=g. \quad (21)$$

因为是规则区域，所以法向导数的方向均为水平或竖直的。但因为区域外部的函数是没有定义的，所以只能用区域内部的点来估计。根据 Example 6.38 的一阶差分格式可得：

$$\frac{1}{2h}(-3u(ih,0)+4u(ih,h)-u(ih,2h))=g(ih,0), \quad (22)$$
$$\frac{1}{2h}(-3u(ih,1)+4u(ih,1-h)-u(ih,1-2h))=g(ih,1), \quad (23)$$
$$\frac{1}{2h}(-3u(0,jh)+4u(h,jh)-u(2h,jh))=g(0,jh), \quad (24)$$
$$\frac{1}{2h}(-3u(1,jh)+4u(1-h,jh)-u(1-2h,jh))=g(1,jh). \quad (25)$$

将边界条件与内部网格点的差分格式联立，得到的方程组是奇异的（恰有一个冗余方程），这个结果恰和 Neumann 边界条件解中的任意常数 C 对应。和一维相同，我们无需对它进行特别处理。

特别地，对于矩形的四个角，虽然我们在计算矩形内部的离散点值时无需用到它们的值，但为了保证限制算子和插值算子的正确性，我们仍需要维护这四个点的值。

根据教材 Exercise6.42 可知，该差分格式在边界点处的 LTE 为 $O(h^2)$。它在内部点处的 LTE 也是 $O(h^2)$。

$\|A^{-1}\|_2=O(1)$。因此，算法总体误差 $O(h^2)$。

3.4 限制算子

$$I_h^{2h}:\mathbb{R}^{(n+1)^2}\to\mathbb{R}^{(\frac{n}{2}+1)^2} \quad (26)$$

3.4.1 嵌入算子

直接将细网格上和粗网格对应的点作为粗网格上该点的估计。

$$v_{i,j}^{2h} = v_{2i,2j}^{h}. \tag{27}$$

3.4.2 全加权算子

将细网格不在粗网格上，但在粗网格的边的点的权值均分加到相邻的两个点上。
将细网格不在粗网格上，且不在粗网格的边上的点的权值均分加到周围的四个点上。
最后将所有点的权值除 4。
特别地，粗网格四个角处的值仍为细网格四个角处的值，网格边界上的值全部按一维的方式做全加权。

$$\begin{aligned} v_{i,j}^{2h} = \frac{1}{16}(&v_{2i-1,2j-1}^{h} + 2v_{2i-1,2j}^{h} + v_{2i-1,2j+1}^{h} \\ &+ 2v_{2i,2j-1}^{h} + 4v_{2i,2j}^{h} + 2v_{2i,2j+1}^{h} \\ &+ v_{2i+1,2j-1}^{h} + 2v_{2i+1,2j}^{h} + v_{2i+1,2j+1}^{h}) \end{aligned} \tag{28}$$

3.5 插值算子

$$I_{h}^{\frac{h}{2}} : \mathbb{R}^{(n+1)^2} \to \mathbb{R}^{(2n+1)^2}. \tag{29}$$

3.5.1 线性插值算子

对于在细网格而不在粗网格上的点 P，用和 P 在水平或竖直方向相邻的两个点做线性插值，以插值多项式在 P 处的值作为估计。如果相邻的两个点仍不在粗网格上，则再将它们向另一个方向二次插值（即用四个点插值）。

$$\begin{aligned} v_{2i,2j}^{\frac{h}{2}} &= v_{i,j}^{h}, \\ v_{2i,2j+1}^{\frac{h}{2}} &= \frac{1}{2}(v_{i,j}^{h} + v_{i,j+1}^{h}), \\ v_{2i+1,2j}^{\frac{h}{2}} &= \frac{1}{2}(v_{i,j}^{h} + v_{i+1,j}^{h}), \\ v_{2i+1,2j+1}^{\frac{h}{2}} &= \frac{1}{4}(v_{i,j}^{h} + v_{i,j+1}^{h} + v_{i+1,j}^{h} + v_{i+1,j+1}^{h}). \end{aligned} \tag{30}$$

3.5.2 二次插值算子

对于在细网格而不在粗网格上的点 P，用和 P 在水平或竖直方向相邻的三个点做二次插值，以插值多项式在 P 处的值作为估计。如果相邻的三个点仍不在粗网格上，则再将它们向另一个方向二次插值（即用九个点插值）。

根据待插值点与 $(\frac{1}{2}, \frac{1}{2})$ 的位置关系（左上、右上、左下、右下），要分四种情况讨论三个或九个插值点的位置。但插值系数都是相同的。这里仅以待插值点在左下为例。

$$
\begin{aligned}
v^{\frac{h}{2}}_{2i,2j} &= v^h_{i,j},\\
v^{\frac{h}{2}}_{2i,2j+1} &= \frac{1}{8}(3v^h_{i,j} + 6v^h_{i,j+1} - v^h_{i,j+2}),\\
v^{\frac{h}{2}}_{2i+1,2j} &= \frac{1}{8}(3v^h_{i,j} + 6v^h_{i+1,j} - v^h_{i+2,j}),\\
v^{\frac{h}{2}}_{2i+1,2j+1} &= \frac{1}{64}(9v^h_{i,j} + 18v^h_{i,j+1} - 3v^h_{i,j+1}\\
&\quad + 18v^h_{i+1,j} + 36v^h_{i+1,j+1} - 6v^h_{i+1,j+2}\\
&\quad - 3v^h_{i+2,j} - 6v^h_{i+2,j+1} + v^h_{i+2,j+2}).
\end{aligned}
\tag{31}
$$

3.6 代码实现

离散网格的代码如下：

```
template <Cond_t Cond_type>
class Discretor<2, Cond_type> {
private:
    int n;                  // The grid number.
    double h;               // The grid size.
    const Function_2D<double>& f;   // The rhs function.
    const Function_2D<double>& g;   // The boundary function.
public:
    Sparsed_Matrix<double> coef;    // The coefficients.
    Colvec<double> rhs;     // The right terms.
private:
    int id(int i, int j) {
        return i * (n+1) + j;
    }
    void Normal_Laplace_Discretor_2D(int i0, int i1, int i2, int i3, int i4) {
        coef[i0][i0] += 4 / (h*h);
        coef[i0][i1] += -1 / (h*h);
        coef[i0][i2] += -1 / (h*h);
        coef[i0][i3] += -1 / (h*h);
        coef[i0][i4] += -1 / (h*h);
        rhs[i0] += f(i0/(n+1)*h, i0%(n+1)*h);
    }
    void Normal_Dirichlet_Discretor_2D(int i0) {
        coef[i0][i0] += 1;
        rhs[i0] += g(i0/(n+1)*h, i0%(n+1)*h);
    }
    void Normal_Neumann_Discretor_2D(int i0, int i1, int i2) {
        coef[i0][i0] += -1.5 / h;
        coef[i0][i1] += 2 / h;
        coef[i0][i2] += -0.5 / h;
        rhs[i0] += g(i0/(n+1)*h, i0%(n+1)*h);
    }
public:
    Discretor() : n(0), f(_0_2D<double>()), g(_0_2D<double>()), coef(0, 0), rhs(0) {}
    Discretor(int n, const Function_2D<double>& f, const Function_2D<double>& g)
        :n(n), h(1.0/n), f(f), g(g), coef((n+1)*(n+1), (n+1)*(n+1)), rhs((n+1)*(n+1)) {
        h = 1.0/n;
        for (int i = 0; i <= n; ++ i) {
            if (Cond_type & 1) Normal_Neumann_Discretor_2D(id(i,0), id(i,1), id(i,2));
            else Normal_Dirichlet_Discretor_2D(id(i,0));
```

```
        }
        for (int j = 0; j <= n; ++ j) {
            if (Cond_type & 2) Normal_Neumann_Discretor_2D(id(0,j), id(1,j), id(2,j));
            else Normal_Dirichlet_Discretor_2D(id(0,j));
        }
        for (int i = 0; i <= n; ++ i) {
            if (Cond_type & 4) Normal_Neumann_Discretor_2D(id(i,n), id(i,n-1), id(i,n-2));
            else Normal_Dirichlet_Discretor_2D(id(i,n));
        }
        for (int j = 0; j <= n; ++ j) {
            if (Cond_type & 8) Normal_Neumann_Discretor_2D(id(n,j), id(n-1,j), id(n-2,j));
            else Normal_Dirichlet_Discretor_2D(id(n,j));
        }
        for (int i = 1; i < n; ++ i)
            for (int j = 1; j < n; ++ j)
                Normal_Laplace_Discretor_2D(id(i,j), id(i-1,j), id(i+1,j), id(i,j-1), id(i,
                    j+1));
    }
};
```

各种算子代码如下：

```
// The 2D Case is Similar with 1D case.
// The ID of the grid point.
inline int id(int n, int i, int j) {
    return i * (n+1) + j;
}
// 2D Injection Restriction.
Colvec<double> Injection_Restriction_2D(int n, const Colvec<double>& v) {
    Colvec<double> u((n/2+1)*(n/2+1));
    for (int i = 0; i <= n/2; ++ i)
        for (int j = 0; j <= n/2; ++ j)
            u[id(n/2, i, j)] = v[id(n, i*2, j*2)];
    return u;
}
// 2D Full Weighting Restriction.
Colvec<double> Full_Weighting_Restriction_2D(int n, const Colvec<double>& v) {
    Colvec<double> u((n/2+1)*(n/2+1));
    u[id(n/2,0,0)]      = v[id(n,0,0)];
    u[id(n/2,0,n/2)]    = v[id(n,0,n)];
    u[id(n/2,n/2,0)]    = v[id(n,n,0)];
    u[id(n/2,n/2,n/2)]  = v[id(n,n,n)];
    for (int i = 1; i < n/2; ++ i) {
        u[id(n/2, i, 0)] = (
              v[id(n, i*2, 0)]
            + v[id(n, i*2-1, 0)] * 0.5
            + v[id(n, i*2+1, 0)] * 0.5
        ) / 2;
        u[id(n/2, i, n/2)] = (
              v[id(n, i*2, n)]
            + v[id(n, i*2-1, n)] * 0.5
            + v[id(n, i*2+1, n)] * 0.5
        ) / 2;
    }
    for (int j = 1; j < n/2; ++ j) {
```

```
        u[id(n/2, 0, j)] = (
              v[id(n, 0, j*2)]
            + v[id(n, 0, j*2-1)] * 0.5
            + v[id(n, 0, j*2+1)] * 0.5
        ) / 2;
        u[id(n/2, n/2, j)] = (
              v[id(n, n, j*2)]
            + v[id(n, n, j*2-1)] * 0.5
            + v[id(n, n, j*2+1)] * 0.5
        ) / 2;
    }
    for (int i = 1; i < n/2; ++ i)
        for (int j = 1; j < n/2; ++ j) {
            u[id(n/2, i, j)] = (
                  v[id(n, i*2, j*2)]
                + v[id(n, i*2-1, j*2)] * 0.5
                + v[id(n, i*2+1, j*2)] * 0.5
                + v[id(n, i*2, j*2-1)] * 0.5
                + v[id(n, i*2, j*2+1)] * 0.5
                + v[id(n, i*2-1, j*2-1)] * 0.25
                + v[id(n, i*2-1, j*2+1)] * 0.25
                + v[id(n, i*2+1, j*2-1)] * 0.25
                + v[id(n, i*2+1, j*2+1)] * 0.25
            ) / 4;
        }
    return u;
}
// 2D Linear Interpolation.
Colvec<double> Linear_Interpolation_2D(int n, const Colvec<double>& v) {
    Colvec<double> u((n*2+1)*(n*2+1));
    for (int i = 0; i <= n; ++ i)
        for (int j = 0; j <= n; ++ j) {
            u[id(n*2, i*2, j*2)] = v[id(n, i, j)];
            if (i < n) u[id(n*2, i*2+1, j*2)] = (v[id(n, i, j)] + v[id(n, i+1, j)]) / 2;
            if (j < n) u[id(n*2, i*2, j*2+1)] = (v[id(n, i, j)] + v[id(n, i, j+1)]) / 2;
            if (i < n && j < n) u[id(n*2, i*2+1, j*2+1)] = (v[id(n, i, j)] + v[id(n, i+1, j
                )] + v[id(n, i, j+1)] + v[id(n, i+1, j+1)]) / 4;
        }
    return u;
}
// 2D Quadratic Interpolation.
Colvec<double> Quadratic_Interpolation_2D(int n, const Colvec<double>& v) {
    Colvec<double> u((n*2+1)*(n*2+1));
    for (int i = 0; i <= n; ++ i)
        for (int j = 0; j <= n; ++ j) {
            int i11, i12, i13, i21, i22, i23, i31, i32, i33;
            u[id(n*2, i*2, j*2)] = v[id(n, i, j)];
            if (i < n) {
                if (i < n/2)    i11 = id(n, i, j), i21 = id(n, i+1, j), i31 = id(n, i+2, j)
                    ;
                else            i11 = id(n, i+1, j), i21 = id(n, i, j), i31 = id(n, i-1, j)
                    ;
                u[id(n*2, i*2+1, j*2)] = (v[i11] * 3 + v[i21] * 6 - v[i31]) / 8;
            }
            if (j < n) {
```

```
                if (j < n/2)    i11 = id(n, i, j  ), i12 = id(n, i, j+1), i13 = id(n, i, j
                    +2);
                else            i11 = id(n, i, j+1), i12 = id(n, i, j  ), i13 = id(n, i, j
                    -1);
                u[id(n*2, i*2, j*2+1)] = (v[i11] * 3 + v[i12] * 6 - v[i13]) / 8;
            }
            if (i < n && j < n) {
                if (i < n/2 && j < n/2) {
                    i11 = id(n, i  , j  ), i12 = id(n, i  , j+1), i13 = id(n, i  , j+2);
                    i21 = id(n, i+1, j  ), i22 = id(n, i+1, j+1), i23 = id(n, i+1, j+2);
                    i31 = id(n, i+2, j  ), i32 = id(n, i+2, j+1), i33 = id(n, i+2, j+2);
                }
                else if (i < n/2 && j >= n/2) {
                    i11 = id(n, i  , j+1), i12 = id(n, i  , j  ), i13 = id(n, i  , j-1);
                    i21 = id(n, i+1, j+1), i22 = id(n, i+1, j  ), i23 = id(n, i+1, j-1);
                    i31 = id(n, i+2, j+1), i32 = id(n, i+2, j  ), i33 = id(n, i+2, j-1);
                }
                else if (i >= n/2 && j < n/2) {
                    i11 = id(n, i+1, j  ), i12 = id(n, i+1, j+1), i13 = id(n, i+1, j+2);
                    i21 = id(n, i  , j  ), i22 = id(n, i  , j+1), i23 = id(n, i  , j+2);
                    i31 = id(n, i-1, j  ), i32 = id(n, i-1, j+1), i33 = id(n, i-1, j+2);
                }
                else {
                    i11 = id(n, i+1, j+1), i12 = id(n, i+1, j  ), i13 = id(n, i+1, j-1);
                    i21 = id(n, i  , j+1), i22 = id(n, i  , j  ), i23 = id(n, i  , j-1);
                    i31 = id(n, i-1, j+1), i32 = id(n, i-1, j  ), i33 = id(n, i-1, j-1);
                }
                u[id(n*2, i*2+1, j*2+1)] = (
                      v[i11] *  9 + v[i12] * 18 + v[i13] * -3
                    + v[i21] * 18 + v[i22] * 36 + v[i23] * -6
                    + v[i31] * -3 + v[i32] * -6 + v[i33] *  1
                ) / 64;
            }
        }
    return u;
}
```

3.7 数据测试

二维多重网格类的测试程序为main2.cpp。

运行测试程序时，从命令行中依次输入 n，限制算子，插值算子，迭代方式，ν_1, ν_2。

```
// main2.cpp
#include<bits/stdc++.h>
#include "Function.h"
#include "multigrid.h"
using namespace std;

class U : public Function_2D<double> {
public:
    double operator ()(const double& x, const double& y) const {
        return exp(y + sin(x));
    }
    double partial(const double& x, const double& y, const int& i, const int& j) const {
```

```
        if (i == 0) return exp(y + sin(x));
        if (i == 1 && j == 0) return cos(x) * exp(y + sin(x));
        if (i == 2 && j == 0) return (cos(x) * cos(x) - sin(x)) * exp(y + sin(x));
        throw 0;
    }
} u;

class F : public Function_2D<double> {
public:
    Function_2D<double>& u;
    F(Function_2D<double>& u) : u(u) {}
    double operator ()(const double& x, const double& y) const {
        return -(u.partial(x, y, 2, 0) + u.partial(x, y, 0, 2));
    }
};

typedef unsigned int Cond_t;
template<Cond_t Cond_type>
class G : public Function_2D<double> {
public:
    Function_2D<double>& u;
    G(Function_2D<double>& u) : u(u) {}
    double operator ()(const double& x, const double& y) const {
        if (y == 0) return Cond_type & 1 ? u.partial(x, 0, 0, 1) : u(x, 0);
        if (x == 0) return Cond_type & 2 ? u.partial(0, y, 1, 0) : u(0, y) ;
        if (y == 1) return Cond_type & 4 ? -u.partial(x, 1, 0, 1) : u(x, 1);
        if (x == 1) return Cond_type & 8 ? -u.partial(1, y, 1, 0) : u(1, y);
        throw 0;
    }
};

int main(int argc, char** argv) {
    int n = atoi(argv[1]);
    const Restriction_method i1 = string(argv[2]) == "I" ? Injection : Full_Weighting;
    const Interpolation_method i2 = string(argv[3]) == "L" ? Linear : Quadratic;
    const Cycle_method i3 = string(argv[4]) == "V" ? V_cycle : FMG;
    int T1 = atoi(argv[5]), T2 = atoi(argv[6]);
    auto Solver_D = Multigrid<2, 0>(F(u), G<0>(u));
    Solver_D.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
    auto Solver_N = Multigrid<2, 15>(F(u), G<15>(u));
    Solver_N.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
    auto Solver_M = Multigrid<2, 3>(F(u), G<3>(u));
    Solver_M.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
}
```

因为可能的输入组合数量过多（3 种边界、2 种限制算子、2 种插值算子、2 种迭代方法、5 个网格规模，共 120 种组合），所以这里只选取部分输入进行测试。

固定 $\epsilon = 10^{-8}$。

根据多重网格的收敛定理，$\nu_1 = \nu_2 = 2$ 时即可有效过滤掉高频波。但 ν_1, ν_2 与迭代次数之间有非线性的负相关性，实际应用时取 $\nu_1 = \nu_2 = 5$ 最佳。

表中“残差”和“误差”是无穷范数意义下的。其他范数（一范数和二范数）意义下的残差和误差见测试结果`test2.txt`。

表中“运行时间”的单位为毫秒。

网格大小	边界条件	限制算子	插值算子	迭代方法	残差	误差	迭代次数	运行时间
32	Dirichlet	Injection	Linear	V-Cycle	5.80e-11	3.45e-05	9	45
	Dirichlet	Full-Weighting	Linear	V-Cycle	2.85e-09	3.45e-05	11	54
	Dirichlet	Full-Weighting	Quadratic	V-Cycle	1.10e-09	3.45e-05	9	45
	Dirichlet	Injection	Linear	FMG	3.55e-11	3.45e-05	6	48
	Dirichlet	Full-Weighting	Linear	FMG	7.28e-11	3.45e-05	6	46
	Dirichlet	Full-Weighting	Quadratic	FMG	2.95e-10	3.45e-05	5	33
	Neumann	Injection	Linear	V-Cycle	2.89e-03	8.35e-04	25	131
	Neumann	Full-Weighting	Linear	V-Cycle	2.92e-03	8.36e-04	25	130
	Neumann	Full-Weighting	Quadratic	V-Cycle	2.82e-03	8.35e-04	23	126
	Neumann	Injection	Linear	FMG	3.08e-03	8.38e-04	13	96
	Neumann	Full-Weighting	Linear	FMG	3.09e-03	8.38e-04	12	90
	Neumann	Full-Weighting	Quadratic	FMG	2.95e-03	8.36e-04	10	74
	Mixed	Injection	Linear	V-Cycle	2.14e-08	3.87e-04	23	114
	Mixed	Full-Weighting	Linear	V-Cycle	4.59e-08	3.87e-04	22	112
	Mixed	Full-Weighting	Quadratic	V-Cycle	3.10e-08	3.87e-04	19	93
	Mixed	Injection	Linear	FMG	1.26e-08	3.87e-04	10	71
	Mixed	Full-Weighting	Linear	FMG	7.87e-09	3.87e-04	10	77
	Mixed	Full-Weighting	Quadratic	FMG	2.91e-09	3.87e-04	9	61
64	Dirichlet	Full-Weighting	Quadratic	V-Cycle	2.48e-09	8.62e-06	9	240
	Dirichlet	Full-Weighting	Quadratic	FMG	1.42e-10	8.62e-06	5	125
	Neumann	Full-Weighting	Quadratic	V-Cycle	9.35e-04	2.08e-04	23	611
	Neumann	Full-Weighting	Quadratic	FMG	9.90e-04	2.08e-04	10	273
	Mixed	Full-Weighting	Quadratic	V-Cycle	3.91e-08	9.78e-05	20	526
	Mixed	Full-Weighting	Quadratic	FMG	1.42e-08	9.78e-05	8	207
128	Dirichlet	Full-Weighting	Quadratic	V-Cycle	3.09e-10	2.16e-06	10	873
	Dirichlet	Full-Weighting	Quadratic	FMG	1.54e-10	2.16e-06	5	647
	Neumann	Full-Weighting	Quadratic	V-Cycle	3.10e-04	5.32e-05	24	2086
	Neumann	Full-Weighting	Quadratic	FMG	3.32e-04	5.32e-05	11	1389
	Mixed	Full-Weighting	Quadratic	V-Cycle	4.74e-08	2.46e-05	21	1851
	Mixed	Full-Weighting	Quadratic	FMG	7.82e-09	2.46e-05	8	954
256	Dirichlet	Full-Weighting	Quadratic	V-Cycle	1.01e-09	5.39e-07	10	3584
	Dirichlet	Full-Weighting	Quadratic	FMG	5.98e-10	5.39e-07	5	2327
	Neumann	Full-Weighting	Quadratic	V-Cycle	1.03e-04	1.35e-05	25	9254
	Neumann	Full-Weighting	Quadratic	FMG	1.11e-04	1.34e-05	11	5214
	Mixed	Full-Weighting	Quadratic	V-Cycle	5.54e-08	6.17e-06	22	8295
	Mixed	Full-Weighting	Quadratic	FMG	3.88e-08	6.17e-06	7	3261
512	Dirichlet	Full-Weighting	Quadratic	V-Cycle	4.92e-09	1.35e-07	10	14934
	Dirichlet	Full-Weighting	Quadratic	FMG	2.01e-09	1.35e-07	5	10342
	Neumann	Full-Weighting	Quadratic	V-Cycle	3.44e-05	3.38e-06	25	39169
	Neumann	Full-Weighting	Quadratic	FMG	3.72e-05	3.38e-06	11	22276
	Mixed	Full-Weighting	Quadratic	V-Cycle	1.72e-07	1.54e-06	22	35791
	Mixed	Full-Weighting	Quadratic	FMG	2.19e-08	1.54e-06	7	15419

3.8 误差分析和效率分析（在这里回答作业文档中的 B 题）

- 当 ϵ 取到 10^{-8} 时，解的误差已经收敛，即使残差的数量级有明显差异，误差也大致相等。
- 二维情况下，复杂的限制和插值算子明显优于简单的。
- FMG 的收敛速度明显快于 V-Cycle。
- 虽然 FMG 的过程比 V-Cycle 复杂的多，但它们单次循环的时间基本上是相同的。
- 收敛速度和 n 无关。事实上，它只和 ϵ, ν_1, ν_2 有关。
- 对于纯 Neumann 边界条件，因为矩阵欠定，残差不收敛到 0，但会收敛到一个小常数。
- 对于每个边界条件，算法的收敛阶都是二阶的。n 每增加 2 倍，误差变为原来的 $\frac{1}{4}$。

我们在第七章中实现了用 LU 分解法求解离散方程组。二者的区别在于：

- 多重网格法的时间复杂度是 $O(n^d \log \epsilon^{-1})$，而 LU 分解法的时间复杂度为 $O(n^{3d})$，加入冗余行列优化后为 $O(n^{2d})$。
- 多重网格法的空间复杂度是 $O(n^d)$，LU 分解法的空间复杂度是 $O(n^{2d})$。
- 多重网格法的残差可以任意指定（但不能低于 $O(n^d \epsilon_u)$ 即矩阵乘向量的机器精度），而 LU 分解法的残差是 $O(n^{2d}\epsilon_u)$。

我们比较两种算法（多重网格用 Full-Weighted+Quadratic+FMG）在二维规则区域，网格数为 $n = 16, 32, 64, 128$ 时的时间效率：

- $n = 16$，多重网格 9ms，误差 1.38e-04；LU 分解 8ms，误差 1.22e-04。
- $n = 32$，多重网格 33ms，误差 3.45e-05；LU 分解 63ms，误差 3.24e-05。
- $n = 64$，多重网格 125ms，误差 8.62e-06；LU 分解 922ms，误差 8.36e-06。
- $n = 128$，多重网格 647ms，误差 2.16e-06；LU 分解 21344ms，误差 1.99e-06（LU 分解需要 4GB 以上的内存，在某些机器上可能无法运行）。
- n 更大时，LU 分解所需空间已经超出本机内存限制，无法测试。

当 n 很小时，两种算法的效率基本相同；n 越大，二者的差异越明显。

4 二维非规则区域 Poisson 方程的求解

$$-u_{xx} - u_{yy} = f, u \in \Omega \tag{32}$$

Ω 是 $x = 0, x = 1, y = 1$ 和 $y = \frac{1}{16}\sin(\pi x)$ 围成的区域。

4.1 差分格式

将 Ω 用 $n \times n$ 等距网格划分。

多重网格法要保证网格的完整性，即不能出现 unvisable 的网格。

而对于非规则区域，如果使用与坐标轴平行的标准网格，则不可避免地会有网格 unvisable。

为了保证网格的完整，我们可以将 unvisable 的所有网格全部作为 ghost cell。这样五点差分格式无需进行任何修改。

对于边界 Dirichlet 条件，利用非规则边界与网格的交点的上下两个点对交点进行插值，线性插值有二阶精度；

对于边界 Neumann 条件，利用非规则边界与网格的交点的周围六个点对交点的法向导数进行插值。这样也可以达到二阶精度。

这样实现的好处是，只需修改离散矩阵`Discretor`，几乎无需对多重网格的部分进行任何修改。这个算法在`extra_failed.cpp`中进行了实现。

然而很遗憾，经过验证（用高斯 LU 分解法对相应方程进行了求解），虽然该离散方程是正确的，且有二阶精度，但它的迭代矩阵有模长大于 1 的特征值。因此，该差分格式不能用多重网格法求解。

4.2 改进的差分格式——扭曲网格

将网格用 $n \times n$ 非等距网格划分。垂直网格仍等距，水平网格从上到下按非规则边界的方向逐渐扭曲。即，第 i 行第 j 列网格坐标为 $(ih, (1-jh)\frac{1}{16}\sin(\pi ih) + jh)$。

这样得到的网格与 x 轴不是平行的，但斜率很小，得到的离散矩阵的性质与规则区域接近。

因为网格是扭曲的，格点处的拉普拉斯算子不能使用通常的五点差分法进行离散（否则会有二阶混合导数项 u_{xy} 无法消去）。

对于格点 $u_{i,j}$，用六个点对它的拉普拉斯算子进行插值。为保证对称性，用四个方向的点反复插值四次取平均。即一共用了九个点对一个拉普拉斯算子插值。

因为插值系数的表达式难以计算，所以采用如下数值计算方法：

- 设这六个点为 (x_i, y_i)，系数为 w_i，$i = 0, 1, 2, 3, 4, 5$。
- 将这六个点在 (x_0, y_0) 泰勒展开至二阶。
- 比较两端系数，得到方程组 $Aw = b$。$A = (a_{i,j})_{6\times 6}$。
  - $a_{i,0} = 1$,
  - $a_{i,1} = x_i - x_0$,
  - $a_{i,2} = y_i - y_0$,
  - $a_{i,3} = \dfrac{(x_i - x_0)^2}{2}$,
  - $a_{i,4} = \dfrac{(y_i - y_0)^2}{2}$,
  - $a_{i,5} = (x_i - x_0)(y_i - y_0)$,
  - $b_3 = b_4 = -1, b_0 = b_1 = b_2 = b_5 = 0$
- 解方程组得到各项系数，填入离散矩阵中。

因为我们消去了误差中所有含零至二阶导数的项，所以该格式至少是一阶收敛的。

事实上经过数值验证可以发现该格式是二阶收敛的。这是因为，若以 (x_0, y_0) 点为一个格点建立规则网格，则每个 p_i 和规则网格点之间的距离都为 $O(h^2)$。这就导致三阶导数的系数虽然不是 0，但也是 $O(h^2)$ 的。

4.3 扭曲网格边界条件的处理

对于 Dirichlet 边界条件，无需进行特殊处理。和规则网格同样处理即可。

对于 Neumann 边界条件，下、左、右边界用六个点插值，上边界同规则边界用三个点插值。对于下、左、右边界，A 和上一节相同，$b_1 = \cos\langle n, x\rangle, b_2 = \cos\langle n, y\rangle$。

4.4 限制算子和插值算子

限制算子和插值算子无需做任何更改。理由如下：

我们的目的是设计一个二阶收敛的差分格式。

以线性插值为例，它得到的结果是两个粗网格格点的中点处的值。而粗网格格点的中点与细网格的距离是 $O(h^2)$ 的，根据泰勒展开，误差也是 $O(h^2)$ 的。

同样的结论也可应用于限制算子。

4.5 代码实现

```cpp
#include <bits/stdc++.h>
#include "function.h"
#include "Matrix.h"
#include "Sparsed_Matrix.h"
#include "multigrid.h"
using namespace std;

const double pi = acos(-1);

inline int sgn(double x) {
    if (fabs(x) < 1e-12) return 0;
    return x>0 ? 1 : -1;
}

struct vec {
    double x,y;
    vec() {}
    vec(double x,double y):x(x),y(y){}
    vec operator + (const vec& p) const {
        return vec(x+p.x, y+p.y);
    }
    vec operator - (const vec& p) const {
        return vec(x-p.x, y-p.y);
    }
    double norm() const {
        return sqrt(x*x + y*y);
    }
    bool operator < (const vec& p) const {
        return sgn(x-p.x) == -1 || (sgn(x-p.x) == 0 && sgn(y-p.y) == -1);
    }
    bool operator == (const vec& p) const {
        return sgn(x-p.x) == 0 && sgn(y-p.y) == 0;
    }
};
typedef vec pnt;

inline double dis(const pnt& p, const pnt& q) {
    return (p-q).norm();
}

typedef unsigned int Cond_t;
template <Cond_t Cond_type>
class Irnormal_Discretor {
private:
    int n;                  // The grid number.
    double h;               // The grid size.
    const Function_2D<double>& f;   // The rhs function.
    const Function_2D<double>& g;   // The boundary function.
```

```
public:
    vector<pnt> p;
    Sparsed_Matrix<double> coef;    // The coefficients.
    Colvec<double> rhs;     // The right terms.
private:
    int id(int i, int j) {
        return i * (n+1) + j;
    }
    void Irnormal_Laplace_Discretor_2D(const vector<int>& id) {
        vector<pnt> q;
        for (int i:id) q.push_back(p[i]);
        Matrix<double> w(6, 6);
        Colvec<double> b(6);
        for (int i = 0; i < 6; ++ i) {
            w[0][i] = 1;
            w[1][i] = q[i].x - q[0].x;
            w[2][i] = q[i].y - q[0].y;
            w[3][i] = (q[i].x - q[0].x) * (q[i].x - q[0].x) / 2;
            w[4][i] = (q[i].y - q[0].y) * (q[i].y - q[0].y) / 2;
            w[5][i] = (q[i].x - q[0].x) * (q[i].y - q[0].y);
        }
        b[3] = b[4] = -1;
        Colvec<double> sol = Gauss_Improved_Solve(w, b);
        for (int i = 0; i < 6; ++ i) coef[id[0]][id[i]] += sol[i];
        rhs[id[0]] += f(q[0].x, q[0].y);
    }
    void Dirichlet_Discretor_2D(int i0) {
        coef[i0][i0] += 1;
        rhs[i0] += g(p[i0].x, p[i0].y);
    }
    void Normal_Neumann_Discretor_2D(const vector<int>& id) {
        double hx = (1 - sin(pi * p[id[0]].x) / 16) / n;
        coef[id[0]][id[0]] += -1.5 / hx;
        coef[id[0]][id[1]] += 2 / hx;
        coef[id[0]][id[2]] += -0.5 / hx;
        rhs[id[0]] += g(p[id[0]].x, p[id[0]].y);
    }
    void Irnormal_Neumann_Discretor_2D(const vector<int>& id) {
        vector<pnt> q;
        for (int i:id) q.push_back(p[i]);
        Matrix<double> w(6, 6);
        Colvec<double> b(6);
        for (int i = 0; i < 6; ++ i) {
            w[0][i] = 1;
            w[1][i] = q[i].x - q[0].x;
            w[2][i] = q[i].y - q[0].y;
            w[3][i] = (q[i].x - q[0].x) * (q[i].x - q[0].x) / 2;
            w[4][i] = (q[i].y - q[0].y) * (q[i].y - q[0].y) / 2;
            w[5][i] = (q[i].x - q[0].x) * (q[i].y - q[0].y);
        }
        if (q[0].x == 0) b[1] = 1;
        else if (q[0].x == 1) b[1] = -1;
        else {
            double dx = pi*cos(pi*q[0].x)/16, dy = -1;
            double dr = sqrt(dx*dx + dy*dy);
```

```
            b[1] = dx/dr, b[2] = dy/dr;
        }
        Colvec<double> sol = Gauss_Improved_Solve(w, b);
        for (int i = 0; i < 6; ++ i) coef[id[0]][id[i]] += sol[i];
        rhs[id[0]] += g(q[0].x, q[0].y);
    }
public:
    Irnormal_Discretor() : n(0), f(_0_2D<double>()), g(_0_2D<double>()), coef(0, 0), rhs(0)
        {}
    Irnormal_Discretor(int n, const Function_2D<double>& f, const Function_2D<double>& g)
        :n(n), h(1.0/n), f(f), g(g), coef((n+1)*(n+1), (n+1)*(n+1)), rhs((n+1)*(n+1)) {
        p.resize((n+1)*(n+1));
        for (int i = 0; i <= n; ++ i)
            for (int j = 0; j <= n; ++ j)
                p[id(i,j)] = pnt(i*h, (1 - j*h) * sin(i*h*pi) / 16 + j*h);
        for (int i = 0; i <= n; ++ i) {
            if (Cond_type & 1) {
                if (i == 0 || i == n) Normal_Neumann_Discretor_2D({id(i,0), id(i,1), id(i
                    ,2)});
                else {
                    if (i < n/2)    Irnormal_Neumann_Discretor_2D({id(i,0), id(i+1,0), id(i
                        +2,0), id(i,1), id(i+1,1), id(i,2)});
                    else            Irnormal_Neumann_Discretor_2D({id(i,0), id(i-1,0), id(i
                        -2,0), id(i,1), id(i-1,1), id(i,2)});
                }
            }
            else Dirichlet_Discretor_2D(id(i,0));
        }
        for (int j = 1; j < n; ++ j) {
            if (Cond_type & 2) {
                if (j < n/2)    Irnormal_Neumann_Discretor_2D({id(0,j), id(1,j), id(2,j),
                    id(0,j+1), id(1,j+1), id(0,j+2)});
                else            Irnormal_Neumann_Discretor_2D({id(0,j), id(1,j), id(2,j),
                    id(0,j-1), id(1,j-1), id(0,j-2)});
            }
            else Dirichlet_Discretor_2D(id(0,j));
        }
        for (int i = 0; i <= n; ++ i) {
            if (Cond_type & 4) Normal_Neumann_Discretor_2D({id(i,n), id(i,n-1), id(i,n-2)})
                ;
            else Dirichlet_Discretor_2D(id(i,n));
        }
        for (int j = 1; j < n; ++ j) {
            if (Cond_type & 8) {
                if (j < n/2)    Irnormal_Neumann_Discretor_2D({id(n,j), id(n-1,j), id(n-2,j
                    ), id(n,j+1), id(n-1,j+1), id(n,j+2)});
                else            Irnormal_Neumann_Discretor_2D({id(n,j), id(n-1,j), id(n-2,j
                    ), id(n,j-1), id(n-1,j-1), id(n,j-2)});
            }
            else Dirichlet_Discretor_2D(id(n,j));
        }
        for (int i = 1; i < n; ++ i)
            for (int j = 1; j < n; ++ j) {
                Irnormal_Laplace_Discretor_2D({id(i,j), id(i-1,j), id(i+1,j), id(i,j-1), id
                    (i,j+1), id(i-1,j-1)});
```

```
149                 Irnormal_Laplace_Discretor_2D({id(i,j), id(i-1,j), id(i+1,j), id(i,j-1), id
                        (i,j+1), id(i-1,j+1)});
150                 Irnormal_Laplace_Discretor_2D({id(i,j), id(i-1,j), id(i+1,j), id(i,j-1), id
                        (i,j+1), id(i+1,j-1)});
151                 Irnormal_Laplace_Discretor_2D({id(i,j), id(i-1,j), id(i+1,j), id(i,j-1), id
                        (i,j+1), id(i+1,j+1)});
152             }
153     }
154 };
155
156 template<Cond_t Cond_type>
157 class Irnormal_Multigrid {
158 private:
159     double w;                                       // w is the release coefficient.
160     Colvec<double> (*Restriction) (int, const Colvec<double>&);     // Restriction: Full
            Weighting and Injection.
161     Colvec<double> (*Interpolation) (int ,const Colvec<double>&);   // Interpolation:
            Linear and Quadratic.
162     Cycle_method Cycle_type;                        // Cycle: V-cycle and FMG.
163     map<int, Irnormal_Discretor<Cond_type>> D;  // Discretors for different grids.
164     const Function_2D<double>& f;                   // The rhs function.
165     const Function_2D<double>& g;                   // The boundary function.
166     Colvec<double> sol;                             // The solution vector.
167 public:
168     Irnormal_Multigrid(const Function_2D<double>& f, const Function_2D<double>& g): w
            (2.0/3), f(f), g(g) {}
169 private:
170     Colvec<double> Jacobi(int n, Sparsed_Matrix<double>& A, Colvec<double>& b, const Colvec
            <double>& v0, int T) {
171         int r = (n+1)*(n+1);
172         Colvec<double> v(v0);
173         Sparsed_Matrix<double> trns(r, r);
174         Colvec<double> c(r);
175         for (int i = 0; i < r; ++ i) {
176             double dii = A[i][i];
177             for (auto & [j, x] : A[i])
178                 if (j != i) trns[i][j] = -x / dii;
179             c[i] = b[i] / dii;
180         }
181         Colvec<double> u(r);
182         for (int i = 0; i < T; ++ i) {
183             u = trns * v + c;
184             v = w * u + (1-w) * v;
185         }
186         return v;
187     }
188
189     // v: Initial guess.
190     // A: Discrete matrix for grid number n.
191     // b: Discrete rhs for grid number n.
192     Colvec<double> VC(int n, Sparsed_Matrix<double>& A, Colvec<double>& b, Colvec<double>&
            v0, int T1, int T2) {
193         int r = (n+1)*(n+1);
194         Colvec<double> v = Jacobi(n, A, b, v0, T1);
195         if (n <= 4) return Jacobi(n, A, b, v, T2);
```

```
196         Colvec<double> c = Restriction(n, b - A * v);
197         if (!D.count(n/2)) D.insert({n/2, Irnormal_Discretor<Cond_type>(n/2, f, g)});
198         Colvec<double> zero((n/2+1)*(n/2+1));
199         Colvec<double> v1 = VC(n/2, D[n/2].coef, c, zero, T1, T2);
200         v = v + Interpolation(n/2, v1);
201         return Jacobi(n, A, b, v, T2);
202     }
203     Colvec<double> FMGC(int n, Sparsed_Matrix<double>& A, Colvec<double>& b, int T1, int T2
        ) {
204         Colvec<double> zero((n+1)*(n+1));
205         if (n <= 4) return VC(n, A, b, zero, T1, T2);
206         Colvec<double> c = Restriction(n, b);
207         if (!D.count(n/2)) D.insert({n/2, Irnormal_Discretor<Cond_type>(n/2, f, g)});
208         Colvec<double> v = Interpolation(n/2, FMGC(n/2, D[n/2].coef, c, T1, T2));
209         return VC(n, A, b, v, T1, T2);
210     }
211 public:
212     // n, the finest grid, must be a power of 2.
213     // T1 and T2 are the Two-Grid iteration times specified by the user.
214     void Solve(int n, Restriction_method Restriction_type, Interpolation_method
        Interpolation_type, Cycle_method Cycle_type, int T1 = 5, int T2 = 5, double eps = 1
        e-8, bool is_test = 0, const Function_2D<double>& real = _0_2D<double>()) {
215         if (Restriction_type == Injection) Restriction = Injection_Restriction_2D;
216         else if (Restriction_type == Full_Weighting) Restriction =
                Full_Weighting_Restriction_2D;
217         if (Interpolation_type == Linear) Interpolation = Linear_Interpolation_2D;
218         else if (Interpolation_type == Quadratic) Interpolation =
                Quadratic_Interpolation_2D;
219         if (!D.count(n)) D.insert({n, Irnormal_Discretor<Cond_type>(n, f, g)});
220         Colvec<double> zero((n+1)*(n+1));
221         Colvec<double> rhs = D[n].rhs;
222         int r = (n+1)*(n+1);
223         sol = zero;
224         int iter = 0;
225         double start_time = clock();
226         if (Cycle_type == V_cycle) {
227             while (1) {
228                 ++ iter;
229                 Colvec<double> dta = VC(n, D[n].coef, rhs, zero, T1, T2);
230                 sol = sol + dta;
231                 // cout << iter << ' ' << vert_2(D[n].coef * dta) / (n+1) << endl;
232                 if (vert_2(D[n].coef * dta) / (n+1) < eps) break;
233                 rhs = rhs - D[n].coef * dta;
234             }
235         }
236         else {
237             while (1) {
238                 ++ iter;
239                 Colvec<double> dta = FMGC(n, D[n].coef, rhs, T1, T2);
240                 sol = sol + dta;
241                 // cout << iter << ' ' << vert_2(D[n].coef * dta) / (n+1) << endl;
242                 if (vert_2(D[n].coef * dta) / (n+1) < eps) break;
243                 rhs = rhs - D[n].coef * dta;
244             }
245         }
```

```
246         double end_time = clock();
247         cout << "Dimension :" << 2 << endl;
248         cout << "Condition :" << Cond_type << endl;
249         cout << "Grid Number :" << n << endl;
250         cout << "Restriction :" << (Restriction_type == Injection ? "Injection" : "Full
                Weighting") << endl;
251         cout << "Interpolation :" << (Interpolation_type == Linear ? "Linear" : "Quadratic
                ") << endl;
252         cout << "Cycle :" << (Cycle_type == V_cycle ? "V_cycle" : "FMG") << endl;
253         cout << "Iteration times :" << T1 << "," << T2 << endl;
254         cout << "Cycle times :" << iter << endl;
255         Colvec<double> e = D[n].coef * sol - D[n].rhs;
256         cout << "Residual Error in L_1 :" << vert_1(e) / r << endl;
257         cout << "Residual Error in L_2 :" << vert_2(e) / sqrt(r) << endl;
258         cout << "Residual Error in L_inf :" << vert_inf(e) << endl;
259         if (is_test) {
260             double h = 1.0 / n;
261             double E1 = 0, E2 = 0, Einf = 0, C = 0;
262             if (Cond_type == 15) {
263                 for (int i = 0; i <= n; ++ i)
264                     for (int j = 0; j <= n; ++ j)
265                         C += sol[id(n,i,j)] - real(D[n].p[id(n,i,j)].x, D[n].p[id(n,i,j)].y
                            );
266                 C /= r;
267             }
268             for (int i = 0; i <= n; ++ i)
269                 for (int j = 0; j <= n; ++ j) if (i!=0&&i!=n || j!=0&&j!=n){
270                     double eij = fabs(sol[id(n,i,j)] - C - real(D[n].p[id(n,i,j)].x, D[n].p
                        [id(n,i,j)].y));
271                     E1 += fabs(eij);
272                     E2 += eij * eij;
273                     Einf = max(Einf, eij);
274                 }
275             E1 /= r, E2 /= r, E2 = sqrt(E2);
276             cout << "Solution Error in L_1 :" << E1 << endl;
277             cout << "Solution Error in L_2 :" << E2 << endl;
278             cout << "Solution Error in L_inf :" << Einf << endl;
279         }
280         cout << "Run time :" << (end_time - start_time) / CLOCKS_PER_SEC << endl;
281     }
282 };
283
284 class U : public Function_2D<double> {
285 public:
286     double operator ()(const double& x, const double& y) const {
287         return exp(y + sin(x));
288     }
289     double partial(const double& x, const double& y, const int& i, const int& j) const {
290         if (i == 0) return exp(y + sin(x));
291         if (i == 1 && j == 0) return cos(x) * exp(y + sin(x));
292         if (i == 2 && j == 0) return (cos(x) * cos(x) - sin(x)) * exp(y + sin(x));
293         throw 0;
294     }
295 } u;
296
```

```
297 class F : public Function_2D<double> {
298 public:
299     Function_2D<double>& u;
300     F(Function_2D<double>& u) : u(u) {}
301     double operator ()(const double& x, const double& y) const {
302         return -(u.partial(x, y, 2, 0) + u.partial(x, y, 0, 2));
303     }
304 };
305 
306 typedef unsigned int Cond_t;
307 template<Cond_t Cond_type>
308 class G : public Function_2D<double> {
309 public:
310     Function_2D<double>& u;
311     G(Function_2D<double>& u) : u(u) {}
312     double operator ()(const double& x, const double& y) const {
313         if (x == 0) return Cond_type & 2 ? u.partial(0, y, 1, 0) : u(0, y) ;
314         if (y == 1) return Cond_type & 4 ? -u.partial(x, 1, 0, 1) : u(x, 1);
315         if (x == 1) return Cond_type & 8 ? -u.partial(1, y, 1, 0) : u(1, y);
316         if (fabs(y - sin(pi * x) / 16) < eps) {
317             if (Cond_type & 1) {
318                 double dx = pi*cos(pi*x)/16, dy = -1;
319                 double dr = sqrt(dx*dx + dy*dy);
320                 return (u.partial(x, y, 1, 0) * dx + u.partial(x, y, 0, 1) * dy) / dr;
321             }
322             else return u(x, y);
323         }
324         cerr << "?" << x<<'␣'<<y<<endl;
325         throw 0;
326     }
327 };
328 
329 int main(int argc, char** argv) {
330     int n = atoi(argv[1]);
331     const Restriction_method i1 = string(argv[2]) == "I" ? Injection : Full_Weighting;
332     const Interpolation_method i2 = string(argv[3]) == "L" ? Linear : Quadratic;
333     const Cycle_method i3 = string(argv[4]) == "V" ? V_cycle : FMG;
334     int T1 = atoi(argv[5]), T2 = atoi(argv[6]);
335     auto Solver_D = Irnormal_Multigrid<0>(F(u), G<0>(u));
336     Solver_D.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
337     auto Solver_N = Irnormal_Multigrid<15>(F(u), G<15>(u));
338     Solver_N.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
339     auto Solver_M = Irnormal_Multigrid<3>(F(u), G<3>(u));
340     Solver_M.Solve(n, i1, i2, i3, T1, T2, 1e-8, 1, u);
341 }
```

4.6 数据测试

对 $n = 32, 64, 128, 256, 512$ 的三种边界条件分别用 VC 和 FMG 求解。

固定 $\epsilon = 10^{-8}$。

根据多重网格的收敛定理，$\nu_1 = \nu_2 = 2$ 时即可有效过滤掉高频波。但 ν_1, ν_2 与迭代次数之间有非线性的负相关性，实际应用时取 $\nu_1 = \nu_2 = 5$ 最佳。

表中“残差”和“误差”是无穷范数意义下的。其他范数（一范数和二范数）意义下的残差和误差见

测试结果`test2.txt`。

表中“运行时间”的单位为毫秒。

对各种算子的比较在规则边界中已经比较完善，这里不再比较，所有限制算子均采用全加权算子，所有插值算子均采用二次插值算子。

网格大小	边界条件	迭代方法	残差	误差	迭代次数	运行时间
32	Dirichlet	V-Cycle	3.52e-10	3.49e-05	11	103
	Dirichlet	FMG	1.20e-10	3.49e-05	6	76
	Neumann	V-Cycle	1.27e-01	3.52e-02	29	278
	Neumann	FMG	1.34e-01	3.52e-02	13	168
	Mixed	V-Cycle	5.81e-08	4.90e-04	43	389
	Mixed	FMG	9.53e-09	4.90e-04	15	187
64	Dirichlet	V-Cycle	4.45e-10	8.72e-06	11	429
	Dirichlet	FMG	2.08e-10	8.72e-06	6	323
	Neumann	V-Cycle	8.09e-02	2.23e-02	30	1211
	Neumann	FMG	8.59e-02	2.23e-02	12	662
	Mixed	V-Cycle	6.85e-08	1.25e-04	43	1658
	Mixed	FMG	8.22e-09	1.25e-04	13	701
128	Dirichlet	V-Cycle	9.86e-10	2.18e-06	11	1905
	Dirichlet	FMG	1.49e-09	2.18e-06	5	1142
	Neumann	V-Cycle	5.25e-02	1.35e-02	30	4880
	Neumann	FMG	5.63e-02	1.35e-02	12	2744
	Mixed	V-Cycle	8.12e-08	3.15e-05	43	7154
	Mixed	FMG	8.86e-09	3.15e-05	12	2603
256	Dirichlet	V-Cycle	3.91e-09	5.45e-07	11	7834
	Dirichlet	FMG	2.70e-09	5.45e-07	5	5082
	Neumann	V-Cycle	3.45e-02	7.89e-03	31	21753
	Neumann	FMG	3.73e-02	7.89e-03	12	11499
	Mixed	V-Cycle	9.31e-08	7.92e-06	43	34290
	Mixed	FMG	2.73e-08	7.92e-06	11	11759
512	Dirichlet	V-Cycle	1.72e-08	1.36e-07	11	34343
	Dirichlet	FMG	1.02e-08	1.36e-07	5	19902
	Neumann	V-Cycle	2.28e-02	4.51e-03	31	129667
	Neumann	FMG	2.48e-02	4.51e-03	12	47439
	Mixed	V-Cycle	1.58e-07	1.99e-06	42	125197
	Mixed	FMG	1.24e-07	1.99e-06	10	40004

4.7 误差分析和效率分析

通过上面的结果，我们得到如下结论：

- 当 ϵ 取到 10^{-8} 时，解的误差已经收敛，即使残差的数量级有明显差异，误差也大致相等。
- FMG 的收敛速度明显快于 V-Cycle。
- 虽然 FMG 的过程比 V-Cycle 复杂的多，但它们单次循环的时间基本上是相同的。
- 收敛速度和 n 无关。事实上，它只和 ϵ, ν_1, ν_2 有关。
- 对于纯 Neumann 边界条件，因为矩阵欠定，残差不收敛到 0，但会收敛到一个小常数。

- 对于 Dirichlet 和 Mixed 边界条件，算法的收敛阶都是二阶的。n 每增加 2 倍，误差变为原来的 $\frac{1}{4}$。且误差和规则边界的数量级是相同的。
- 对于 Neumann 边界条件，算法的收敛阶低于一阶。具体原因尚未确定。
- 非规则区域方程的求解远慢于规则区域方程。因为几乎每个 Stencil 的建立都需要解一个 6×6 的方程组。