Project: 四阶有限体积方法

作者：Wenchong Huang

时间：Aug 20, 2023

目录

第 1 章 用户手册

1.1 编译与测试说明

请在项目根目录下执行以下命令完成编译：

```
make
```

1.1.1 测试用例一、二

对于第一个测试用例，请使用以下命令运行测试：

```
time ./test1 M
```

其中M表示将区域划分为 $M \times M$ 的网格。程序将会输出误差、求解时间，并将求解结果输出到result.txt。在 matlab 中编写如下脚本即可绘制图像。

```
[x,y]=meshgrid(0:1/M:(1-1/M),0:1/M:(1-1/M));
z = load("result.txt")';
pcolor(x,y,z)
shading interp;
```

对于第二个测试用例，运行与绘图方法与第一个测试用例完全相同，将test1改为test2即可。但是程序将不会输出误差，需要用 matlab 读取解并用 Richardson 外插法计算误差。

1.1.2 测试用例三

对于第三个测试用例，请使用以下命令运行测试：

```
time ./test3 M Re Cr eps
```

其中M表示将区域划分为 $M \times M$ 的网格；Re（正整数）表示雷诺数；Cr（正实数）表示柯朗数；eps（正实数）表示多重网格的迭代精度。注意，eps并不是越小越好，过小会导致求解速度很慢但解的质量几乎没有提升。

程序将会输出误差、求解时间，并将求解结果输出到result.txt。用 matlab 读取时，请使用以下脚本：

```
sol = load("result.txt");
ux = reshape(sol(1,:),M,M);
uy = reshape(sol(2,:),M,M);
p = reshape(sol(3,:),M,M);
```

1.1.3 对流扩散方程求解器接口

以测试用例一为例，求解器的调用如下。

```
// 新建求解器
FV_MOL_Solver solver;
// 设置网格大小
solver.setGridSize(stoi(argv[1]));
// 设置终止时间
solver.setEndTime(1.0);
```

```
// 设置扩散系数
solver.setNu(nu);
// 设置时间步长，三个参数依次为：柯朗数、ux最大值、uy最大值
solver.setTimeStepWithCaurant(1.0, 1.0, 0.5);
// 设置外力项
solver.setForcingTerm(&f);
// 设置初值条件
solver.setInitial(&phi);
// 设置边值条件
solver.setBondary("down", &phi, "Dirichlet");
solver.setBondary("left", &phi, "Dirichlet");
solver.setBondary("up", &dyphi, "Neumann");
solver.setBondary("right", &dxphi, "Neumann");
// 设置速度，当速度场为常向量时，用setConstVelocity能显著提速
solver.setConstVelocity(1.0, 0.5);
// 求解
solver.solve();
// 将解输出到文件
solver.output("result.txt");
// 计算误差。需要提供真解、范数，在norm.h中提供了p范数、无穷范数可供调用
cout << "Error in max-norm: " << solver.checkerr(&phi, Norm_inf()) << endl;
cout << "Error in 1-norm: " << solver.checkerr(&phi, Norm_p(1)) << endl;
cout << "Error in 2-norm: " << solver.checkerr(&phi, Norm_p(2)) << endl;
```

以测试用例二为例，求解器的调用如下。

```
FV_MOL_Solver solver;
solver.setGridSize(stoi(argv[1]));
solver.setEndTime(10.0);
solver.setNu(nu);
solver.setTimeStepWithCaurant(1.0, 0.1, 0.1);
solver.setInitial(&initphi);
// 设置外力项为0
solver.setNoForcingTerm();
// 设置边值条件为周期
solver.setPeriodicBondary();
// 设置速度场（非常值）
solver.setVelocity(&ux, &uy);
solver.solve();
solver.output("result.txt");
```

初值条件、边值条件、外力项使用的函数均为TimeFunction2D的派生类，其原型的一部分如下

```
class TimeFunction2D{
public:
    virtual double at (const double &x, const double &y, const double &t) const = 0;
    virtual double intFixX(const double &x, const double &d, const double &u, const double &t) const;
    virtual double intFixY(const double &y, const double &d, const double &u, const double &t) const;
    virtual double int2D(const double &l, const double &r, const double &d, const double &u, const
        double &t) const;
```

```
    virtual double accInt2D(const double &l, const double &r, const double &d, const double &u, const
        double &t) const;
    virtual double int2D_order6(const double &l, const double &r, const double &d, const double &u,
        const double &t) const;
    virtual double accInt2D_order6(const double &l, const double &r, const double &d, const double &u,
        const double &t) const;
};
```

用户的自定义函数必须继承TimeFunction2D,并实现函数at(x,y,t),其返回值为用户自定义函数在(x, y, t)处的点值。若用户知道函数积分的解析表达式，也可以在子类中覆盖intFixX（固定x对y积分）和intFixY（固定y对x积分）。此外，若用户知道二重积分的解析表达式，建议用户将int2D、accInt2D、int2D_order6、accInt2D_order6全部覆盖。例如，测试用例二的u_x应该定义为：

```
class FUNCUX : public TimeFunction2D{
  public:
      double at (const double &x, const double &y, const double &t) const{
          return 0.1 * sin(pi*x) * sin(pi*x) * sin(2*pi*y);
      }
      double intFixX(const double &x, const double &d, const double &u, const double &t) const{
          return 0.1 * sin(pi*x) * sin(pi*x) * (cos(2*pi*d) - cos(2*pi*u)) / (2*pi);
      }
      double intFixY(const double &y, const double &d, const double &u, const double &t) const{
          return 0.1 * sin(2*pi*y) * (2*pi*(u-d) + sin(2*pi*d) - sin(2*pi*u)) / (4*pi);
      }
  } ux;
```

1.1.4 INSE 求解器接口

INSE 求解器的调用与对流扩散方程求解器类似。以测试用例三为例，INSE 求解器的调用如下，我们在不同之处添加了注释。

```
INSE_Solver solver;
solver.setGridSize(stoi(argv[1]));
solver.setEndTime(0.5);
solver.setNu(nu);
solver.setTimeStepWithCaurant(stod(argv[3]), 3.0, 3.0);
solver.setNoForcingTerm();
// 设置初值（u为包含两个元素的TimeFunction2D指针数组）
solver.setInitial(u);
// 设置多重网格的迭代精度，若不设，则默认为1e-9
solver.setEps(stod(argv[4]));
solver.solve();
solver.output("result.txt");
// 计算误差，需要提供速度场真解（包含两个元素的TimeFunction2D指针数组）、压强真解（TimeFunction2D指
    针）、范数。checkerr的返回值为一个数组，包含3个元素，分别为ux、uy、p的数值解误差
auto err = solver.checkerr(u, p, Norm_inf());
cout << "Error in max-norm:\t" << err[0] << "\t" << err[1] << "\t" << err[2] << endl;
```

注意，我们的求解器只支持周期边界条件，因此不提供设置边界条件的接口。

第 2 章 数值积分

2.1 辛普森法则

本文中的辛普森法则均指辛普森 1/3 法则，它是 Newton-Cotes 公式在 $n = 2$ 时的情形。辛普森法则用于估计如下形式的一维闭区间积分：

$$\int_a^b f(x)\,\mathrm{d}x.$$

其估计如下：

$$I^S(f;a,b) = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right).$$

对于辛普森法则的误差估计，我们有：

$$\left|I^S(f;a,b) - \int_a^b f(x)\,\mathrm{d}x\right| \le \frac{(b-a)^5}{2880}M,$$

其中 M 为 $|f^{(4)}(x)|$ 的最大值。

2.2 Boole 法则

Boole 法则是 Newton-Cotes 公式在 $n = 4$ 时的情形。Boole 法则对一维闭区间积分估计如下：

$$I^B(f;a,b) = \frac{b-a}{90}\left(7f(a) + 32f\left(\frac{3a+b}{4}\right) + 12f\left(\frac{a+b}{2}\right) + 32f\left(\frac{a+3b}{4}\right) + 7f(b)\right).$$

对于 Boole 法则的误差估计，我们有：

$$\left|I^B(f;a,b) - \int_a^b f(x)\,\mathrm{d}x\right| \le \frac{(b-a)^7}{945\times 2^{11}}M,$$

其中 M 为 $|f^{(6)}(x)|$ 的最大值。

2.3 二重积分

对于二重积分

$$\int_a^b \int_c^d f(x,y)\,\mathrm{d}y\,\mathrm{d}x,$$

可以应用一维闭区间积分公式两次，得到对应的二重积分公式。例如，应用辛普森法则，我们有：

$$\begin{aligned} I^{S,2D}(f;a,b,c,d) = \Bigg[&\left(f(a,c) + 4f\left(\frac{a+b}{2},c\right) + f(b,c)\right) \\ &+ 4\left(f\left(a,\frac{c+d}{2}\right) + 4f\left(\frac{a+b}{2},\frac{c+d}{2}\right) + f\left(b,\frac{c+d}{2}\right)\right) \\ &+ \left(f(a,d) + 4f\left(\frac{a+b}{2},d\right) + f(b,d)\right)\Bigg] \times \frac{(b-a)(d-c)}{36} \end{aligned}$$

而误差估计可以为

$$\left|I^{S,2D}(f;a,b,c,d) - \int_a^b \int_c^d f(x,y)\,\mathrm{d}y\,\mathrm{d}x\right| = O((b-a)^5),$$

或

$$\left|I^{S,2D}(f;a,b,c,d) - \int_a^b \int_c^d f(x,y)\,\mathrm{d}y\,\mathrm{d}x\right| = O((d-c)^5),$$

对于 Boole 法则，有类似的结果。

2.4 自适应积分

我们希望把积分计算得更加精确，最好将误差控制在预先给定的 ε 以内。例如，在二维区域中，用某种法则计算积分值，记为 A；然后将区域四等分，用同样的法则分别计算 4 个等分区域的积分值，记作 $A_1, ..., A_4$，然后计算误差：

$$E = |A - \sum_{i=1}^{4} A_i|.$$

若 $E < \varepsilon$，则将 $\sum_{i=1}^{4} A_i$ 作为结果返回，否则递归计算四个子区域。

2.5 应用场景

计算初值、真解的积分值时，我们将使用用自适应积分方法，将误差控制在 10^{-14} 以内。

考虑到在求解过程中计算面积分、体积分时应用自适应方法代价过大，因此求解过程中，遇到要算积分，我们将直接使用积分公式。对于一维平均积分，使用辛普森法则，有误差估计：

$$\frac{1}{h}\int_{i}^{i+h} g(x)\,\mathrm{d}x = \frac{1}{h} I^{S}(g; i, i+h) + O(h^4).$$

对于二维平均积分，辛普森公式的阶数不够，因此采用 Boole 法则导出的二重积分公式，有误差估计：

$$\frac{1}{h^2}\int_{i}^{i+h}\int_{j}^{j+h} f(x,y)\,\mathrm{d}y\,\mathrm{d}x = \frac{1}{h^2} I^{B,2D}(f; i, i+h, j, j+h) + O(h^5).$$

第 3 章 代数多重网格

为节省工作量，本文所有求解器的解方程部分均采用代数多重网格。经测试，选取恰当的强依赖阈值，可以使代数多重网格的性能与几何多重网格相当，甚至超越几何多重网格[1]。本章的内容来源于我在微分方程数值解课程中的多重网格大作业，并额外添加了 3.6 节。

3.1 背景与介绍

代数多重网格（AMG）是几何多重网格的一个自然推广，当我们发现由偏微分方程引出的离散方程组，使用多重网格具有如此优秀的表现时，自然会想，能不能将其推广到更一般的方程组，解更一般的大型稀疏矩阵。

答案是肯定的，这就是代数多重网格。具体而言，代数多重网格对 M 矩阵具有较好的表现，M 矩阵是满足正定对称、对角元为正数、非对角元非正的矩阵。事实上，大部分椭圆方程的离散矩阵都是 M 矩阵。

对一个 $n \times n$ 的矩阵 A^h，我们不妨将 $1, ..., n$ 看作点，将 A^h 看作一个邻接矩阵，这样我们就得到了一个稀疏图，于是就有了网格结构。

要想应用多重网格的思路，我们必须将方程限制到粗网格中，求解后返回细网格调整。但此时我们面临着严峻的问题：我们不知道网格的结构，只知道一个矩阵。我们需要一些算法来选取粗网格点 Ω^{2h}，并且定义限制算子 I_h^{2h} 与插值算子 I_{2h}^{h}，还要给出粗网格上的矩阵 A^{2h}。

[1]这项比较基于我的结果（代数多重网格）、樊睿的结果（几何多重网格）、凌子恒的结果（几何多重网格）。

3.2 选取粗网格点

要想从中选取粗网格点，首先给出如下定义：

定义 3.1

若 i, j 满足：

$$|a_{i,j}| \geq \theta \max_{k \neq i} |a_{i,k}| \tag{3.1}$$

则称 i 强依赖于 j，也称 j 强影响 i. 其中 θ 是一个事先给定的阈值，称为“强依赖阈值”.

记 N_i 为所有使 $a_{i,j} \neq 0$ 的点 j 构成的集合，称为“相邻点”集。记 S_i 为所有强影响 i 的点构成的集合，S_i^T 为所有强依赖于 i 的点构成的集合。 ♣

在下文中，我们记粗网格 $\Omega^{2h} = C$，细网格 $\Omega^h = C \cup F$，即 F 表示所有在细网格但不在粗网格中的点。我们还记 $C_i = S_i \cap C$，$D_i^s = S_i \cap F$，$D_i^w = N_i \setminus S_i$.

我们希望用一种**启发式的方法**来选取粗网格点，具体地，我们希望：

(1) $\forall i \in F$，对于 $j \in S_i$，或 $j \in C_i$，或 j 强依赖于 C_i 中某点;

(2) C 中的点尽可能多，但需要保持：$\forall i \in C$，没有 $j \in C$ 使得 i 强依赖于 j。

当然，我们只是希望上述性质尽可能得到满足，为此，William L. Briggs 的书上给出了 **“染色算法”**，如下：

1 $C, F \leftarrow \varnothing, \lambda_i \leftarrow |S_i^T|$.

2 选取 $i \in \Omega^h \setminus (C \cup F)$，使 λ_i 最大.

3 $C \leftarrow C \cup \{i\}, F \leftarrow F \cup (S_i^T \setminus C)$.

4 $\forall j \in S_i^T$, $\forall k \in S_j \setminus (C \cup F)$, 令 $\lambda_k \leftarrow \lambda_k + 1$.

5 若 $C \cup F = \Omega^h$，结束，否则返回第 2 步.

注意到上述过程需要维护一个 λ_i 数组，支持增加、删除、求最大下标三种操作，我们期望单次操作的复杂度不超过 $O(\log n)$，因此我们实现了一个**二叉搜索树**。当然，使用线段树、平衡树也是可以的，不过因为可以提前建树，所以平衡操作是不必要的。

3.3 构建插值算子

对于 $\mathbf{e} \in \Omega^{2h}$，插值算子具有如下格式：

$$(I_{2h}^h \mathbf{e})_i = \begin{cases} e_i & \text{若} i \in C, \\ \sum_{j \in C_i} \omega_{ij} e_j & \text{若} i \in F. \end{cases} \tag{3.2}$$

在粗网格中，我们求解的方程是 $A^{2h}\mathbf{e} = \mathbf{f}$，其中 $\mathbf{f}$ 是光滑化（即若干次 G-S 迭代）后的残差，通常比较小，因此

$$a_{ii} e_i \approx -\sum_{j \in N_i} a_{ij} e_j.$$

将 N_i 展开成三类，得

$$a_{ii} e_i \approx -\sum_{j \in C_i} a_{ij} e_j - \sum_{j \in D_i^s} a_{ij} e_j - \sum_{j \in D_i^w} a_{ij} e_j. \tag{3.3}$$

对于 $j \in D_i^w$，将 e_j 近似为 e_i，得

$$\left(a_{ii} + \sum_{j \in D_i^w} a_{ij} \right) e_i \approx -\sum_{j \in C_i} a_{ij} e_j - \sum_{j \in D_i^s} a_{ij} e_j. \tag{3.4}$$

对于 $e_j \in D_i^s$，将其近似为

$$e_j \approx \frac{\sum_{k \in C_i} a_{jk} e_k}{\sum_{k \in C_i} a_{jk}}. \tag{3.5}$$

将 (6.5) 代入 (6.4)，得

$$\omega_{ij} = -\frac{a_{ij} + \sum\limits_{m \in D_i^s} \left(\frac{a_{im} a_{mj}}{\sum\limits_{k \in C_i} a_{mk}} \right)}{a_{ii} + \sum\limits_{n \in D_i^w} a_{in}} \tag{3.6}$$

本小节内容均来源于 William L. Briggs 的书，我们实现了上述过程，同时对反复使用的求和进行了预处理优化，能在 $O(N_i)$ 的时间里完成单个 ω_{ij} 的计算。

3.4 构建限制算子与粗网格矩阵 A^{2h}

事实上，有了插值算子，我们可以直接由对称性和 Garlerkin 条件构建 I_h^{2h} 与 A^{2h}，如下：

$$I_h^{2h} := (I_{2h}^h)^T \tag{3.7}$$

$$A^{2h} := I_h^{2h} A^h I_{2h}^h \tag{3.8}$$

3.5 V-Cycle

实际上，代数多重网格的 V-Cycle 与几何多重网格基本一致，只不过使用的 A^h, I_h^{2h}, I_{2h}^h 都是事先构建好的。为方便表述，记Rh为 I_h^{2h}，Ph为 I_{2h}^h，下面以递归形式描述。

```
VC(h, x, f)
    if size of grid <= 16
        return direct_solve(Ah, f)
    pre-smoothing for v1 times
    e = VC(2h, zeros, Rh*(f-Ah*x))
    x = x + Ph * e
    post-smoothing for v2 times
    return x
```

有了 V-Cycle，自然可以定义 FMG-Cycle.

3.6 纯 Neumann 条件或周期条件下的求解

对于纯 Neumann 条件或周期条件下的二维 Poisson 方程，我们知道，在解存在的前提下，任何两个解都相差一个常数。此时直接使用多重网格求解也是可以的，但是解会发生“漂移”，即平均值越来越大。这对数值方法很不友好，实际测试时，一个 512×512 的二维网格所需的 FMG 迭代次数可以高达 30 次。为此，我们人为添加“平均值为 0”的额外条件，并定义下面这个操作为**“均值标准化”**。

$$\mathbf{v} \leftarrow \mathbf{v} - \left(\frac{1}{n} \sum_{i=1}^{n} v_i \right) \mathbf{e}.$$

我们在第一层网格中，每进行完一次 smoothing 都对当前解执行一次均值标准化，实测效果极佳，可以将 FMG 迭代次数降低到 5 次以内。这个做法可以在文献 [1] 的第 113 页找到。

在 INSE 的近似投影算子中，我们将会用到本节的做法。

第 4 章 对流扩散方程的四阶 MOL 方法

4.1 设计要点

4.2 测试用例一

4.3 测试用例二

第 5 章 INSE 的四阶近似投影方法

5.1 设计要点

5.2 测试用例三

第 6 章 总结

参考文献

[1] William L. Briggs. *A Multigrid Tutorial*. A Multigrid Tutorial, 1987.