《多重网格法解 *Poisson* 方程》项目文档

张志心 3210106357 计科 2106

日期： Apr. 9. 2024

目录

摘　要

该程序包为一维/二维正方形区域Poisson 方程的求解器，采用多重网格法，支持Dirichlet，Neumann，以及混合边值条件问题。

该程序的时间复杂度为 $O(n)$，其中 n 为网格的格点数。

该程序包进行了充分的测试，对于一维/二维 Poisson 方程形式的边值问题，达到了二阶的收敛阶。

1　用户文档

该部分介绍用户使用此程序包求解一维/二维 Poisson 方程的方法。

1.1　问题输入格式与规约

用户需使用 JSON 格式描述待求解问题（二维 Poisson 方程），问题的数学形式为：

$$-\Delta u = f, \text{in } \Omega$$

对于一维问题，$\Omega = [x_l, x_r]$，对于二维问题，$\Omega = [x_l, x_r] \times [y_l, y_r](x_r - x_l = y_r - y_l)$。

对于边值条件，有如下几种类型：

1. Dirichlet 边值条件：$u = g, \text{on } \partial\Omega$;
2. Nuemann 边值条件：$\boldsymbol{n} \cdot \nabla u = g, \text{on } \partial\Omega$;
3. mixed 边值条件：$\begin{cases} u = g_1 & \text{on } X_1 \\ \boldsymbol{n} \cdot \nabla u = g_2 & \text{on } X_2 \end{cases}, \left(X_1 \cap X_2 = \emptyset, X_1 \cup X_2 = \partial\Omega\right)$。

1.1.1　手动构造输入数据

JSON 具体格式如下：

- `Dimension`：问题定义域的维数。
- `Domain_Border`：问题定义域外边界，格式为 $[x_l, x_r]$（一维）或 $[x_l, x_r, y_l, y_r]$（二维），若用户未定义，则默认为 $[0, 1]$ 或 $[0, 1]^2$。对于二维情况，程序会检查输入外边界是否是一个合法的正方形。
- `Domain_Type`：问题定义域的类型，包括 `Regular`, `Irregular`。
- `Center`：若问题定义域是非规则类型，则需要提供内部挖去圆形的信息，圆心坐标格式为 $[x_c, y_c]$。
- `R`：同上，圆形的半径为 R，程序会检查圆形是否位于外边界正方形的**内部**。
- `Grid_n`：用户自定义网格的大小，程序会检查 $2 \le Grid\_n \le 248$，程序离散化的格点为 $(x_i, y_j) = (x_l + ih, y_l + jh), h = \frac{1}{Grid\_n}, i, j = 0, 1, \ldots, Grid\_n$。
- `BC_Type`：Poisson 方程的边值条件的类型，包括 `Dirichlet`, `Neumann`，`mixed`。
- `Cycle_Type`：迭代方法，包括 V-Cycle（`V`）和 FMG（`FMG`）。
- `Accuracy`：迭代终止的残差条件。
- `Interpolation_opt`：插值算子采用的方法，包括 `linear` 和 `quadratic` 两种。
- `Restriction_opt`：限制算子采用的方法，包括 `injection` 和 `full_weight` 两种。
- `Max_Iteration`：最大迭代次数。

- `initial_guess`：迭代的初始猜测，是一个函数，默认为 0。
- `f`：描述 $-\Delta u = f, \text{in } \Omega$，用 C++ 数学表达式的格式给出，表达式解析的具体方法见 `/include/function_generator/ExecCode2.hpp`。
- `g`：描述 Dirichlet 边值条件（$u = g, \text{on } \partial\Omega$）或 Neumann 边值条件（$\boldsymbol{n} \cdot \nabla u = g, \text{on } \partial\Omega$），注意混合边值条件不使用该项来描述，输入为一个 `JSON class`，包含：
  - down : $y = y_l$ 上边值条件表达式；
  - left : $x = x_l$ 上边值条件表达式；
  - right : $x = x_r$ 上边值条件表达式；
  - up : $y = y_r$ 上边值条件表达式.

  如果上述各部分的边值条件的表达式相同，用户可以使用 all 来描述。
- `mixed_g`：仅用于描述混合边值条件，若非混合边值条件，则不需要该项。输入格式为一个 `JSON class`（与 g 类似），每个方向的键值为一个二元组，即 {边值条件的类型, 边值条件的表达式}。
- `Need_Error`：表示是否需要进行解误差分析，为 boolean 类型（`true` 或 `false`）；
- `answer`：若 `Need_Error = true`，需要提供问题的真解，输入格式同 f。

下面给出部分输入数据作为示例：

1. 一维 Dirichlet 边值条件：

```
{
 "Accuracy" : 1.0000000000000001e-10,
 "BC_Type" : "Dirichlet",
 "Cycle_Type" : "FMG",
 "Dimension" : 1,
 "Domain_Border" : [ 0.0, 1.0 ],
 "Domain_Type" : "Regular",
 "Grid_n" : 16,
 "Interpolation_opt" : "linear",
 "Max_Iteration" : 50,
 "Need_Error" : true,
 "Restriction_opt" : "full_weight",
 "answer" : "exp(sin(x))",
 "initial_guess" : "0",
 "f" : "(sin(x)-cos(x)*cos(x))*exp(sin(x))",
 "g" : {
    "left" : "1",
    "right" : "exp(sin(1))"
 },
 "mixed_g" : null
}
```

2. 二维 Neumann 边值条件：

```
{
  "Accuracy" : 1e-10,
  "BC_Type" : "Neumann",
```

```
  "Cycle_Type" : "FMG",
  "Dimension" : 2,
  "Domain_Border" : [ 0.0, 1.0, 0.0, 1.0 ],
  "Domain_Type" : "Regular",
  "Grid_n" : 32,
  "Interpolation_opt" : "quadratic",
  "Max_Iteration" : 50,
  "Need_Error" : true,
  "Restriction_opt" : "full_weight",
  "answer" : "exp(sin(x)+y)",
  "f" : "-(1-sin(x)+cos(x)*cos(x))*exp(sin(x)+y)",
  "g" : {
     "down" : "-exp(sin(x))",
     "left" : "-exp(y)",
     "right" : "cos(1)*exp(sin(1)+y)",
     "up" : "exp(sin(x)+1)"
  }
}
```

3. 二维混合边值条件:

```
{
  "Accuracy" : 1e-10,
  "BC_Type" : "mixed",
  "Cycle_Type" : "FMG",
  "Dimension" : 2,
  "Domain_Border" : [ 0.0, 1.0, 0.0, 1.0 ],
  "Domain_Type" : "Regular",
  "Grid_n" : 32,
  "Interpolation_opt" : "linear",
  "Max_Iteration" : 50,
  "Need_Error" : true,
  "initial_guess" : "0",
  "Restriction_opt" : "injection",
  "answer" : "exp(sin(x)+y)",
  "f" : "-(1-sin(x)+cos(x)*cos(x))*exp(sin(x)+y)",
  "mixed_g" : {
     "down" : [ "Dirichlet", "exp(sin(x)+y)" ],
     "left" : [ "Dirichlet", "exp(sin(x)+y)" ],
     "right" : [ "Neumann", "cos(1)*exp(sin(1)+y)" ],
     "up" : [ "Neumann", "exp(sin(x)+1)" ]
  }
}
```

1.1.2 自动构造输入数据

程序包提供了交互式输入数据，数据自检，并自动构造 JSON 文件的方法，方法如下：

```
make dataGen
./dataGen
```

1.1.3 数据自检

程序会输入数据进行严格的检查，当输入数据不合法时，程序会报出异常并终止，返回 -1。用户也可以自己测试数据是否满足条件，具体方法如下：假设输入 JSON 数据的路径为 `std::string file`，

```
Multigrid_BVP_Problem new_Problem;
deserialize_Json(new_Problem, file);
try {
    new_Problem._self_checked();
} catch (char const * e) {
    cerr << e << "\n";
    // do anything you can
}
```

该方法也用于用户自主创建问题实例，关于问题类的相关定义在 2.1 中说明。

1.2 问题求解

1.2.1 相关函数

程序调用 `Multigrid_BVPsolver::solveProblem(std::string FILE, bool print)` 来求解问题，该函数是对求解流程的进一步封装。使用方法为：假设输入 JSON 数据的路径为 `std::string file`，

```
Square_BVPsolver solver;
solver.solveProblem(file, /*print*/ 0);
```

若 print = 1，则程序会在 `stderr` 中输出问题的描述和求解过程，若 print = 0，则程序只会输出最后的误差分析（若 Need_error = 1）。

用户也可以使用原始接口，具体如下：

```
Multigrid_BVPsolver::readProblem(const std::string file); // 读取数据并检查
Multigrid_BVPsolver::printProblem() const; // 输出问题描述
Multigrid_BVPsolver::solveProblem(bool print = 0); // 在readProblem之后调用，求解当前问题。
Multigrid_BVPsolver::solveProblem(const Multigrid_BVP_Problem &_prob); // 求解已有的问题
Multigrid_BVPsolver::Summary(); // 在solveProblem之后调用，输出当前问题和求解误差分析。
Multigrid_BVPsolver::saveResults(std::string file = "res.csv"); // 输出问题求解结果。
```

1.2.2 自动求解

该程序包提供了 `test.cpp` 用于直接求解用户问题，在根目录下编译求解程序：

```
make all
```

求解方法：

```
usage: ./test [-v|--verbose] <input JSON file>
```

其中 -v 或 –verbose 选项将会提供更多的求解信息（包括问题描述，求解过程，误差分析等），否则，将只提供求解的误差分析。

用户可以一次性求解一个或多个输入数据：

```
./test --verbose 1.json 2.json 3.json
```

2 设计文档

该部分说明此程序包的相关组件的逻辑结构。

2.1 Multigrid_BVP_Problem 类及其序列化方法

`Multigrid_BVP_Problem` 类用于描述本程序包用于解决的问题实例、初始化问题的各项参数、检查问题是否满足规约条件。

```
class Multigrid_BVP_Problem{
  public:
      std::string BC_Type; /* boundary condition : mixed, Dirichlet, Neumann*/
      std::string Domain_Type; /* regular(default) irregular*/
      vector<NUM> Domain_Border; /* [xl, xr, yl, yr] (default : [0,1]x[0,1])*/
      NUM xl, xr, yl, yr, h;
      void getDomain();
      int Grid_n; /* Grid_n x Grid_n */
      std::string f; /* - \Delta u (in Domain)*/
      func F;

      // calculate error
      bool Need_Error;
      std::string answer;
      func Answer;

      std::string initial_guess;
      func Initial_guess;

      // Boundary Condition
      // Dirichlet / Neumann
      map<std::string, std::string> g; /* left, right, up, down -> function*/
                                       /* 0,    1,     2,  3*/
      vector<func> G;
      // mixed
```

```
        map<std::string, pair<std::string, std::string>> mixed_g; /* left, right, up, down ->
            function*/
                                                                  /* 0,    1,     2,  3 */
        vector<std::string> _bc_types;

        int Dimension;
        std::string Cycle_Type; // FMG or V
        std::string Restriction_opt; // full_weight, linear
        std::string Interpolation_opt; // lenear, quadratic
        int Max_Iteration;
        NUM Accuracy;

        Multigrid_BVP_Problem() :
            Domain_Border({0, 1, 0, 1}), Domain_Type("Regular"),
            Max_Iteration(20), Accuracy(1e-10), Need_Error(0) {}

        void _self_checked ();
        void print() const;
    };
```

该类实例可以直接序列化为 JSON 文档，或由 JSON 文档里反序列化得到。此类用户自定义类的序列化方法在 `/include/serialization/serialize_json.hpp` 中实现。采用编译预处理方式展开该方法：

```
REGISTER_SERIALIZATION_JSON(BC_Type, Domain_Type, Domain_Border, Dimension,
                            Restriction_opt, Interpolation_opt, Cycle_Type, Grid_n,
                            f, g, mixed_g, Need_Error, answer, Max_Iteration, Accuracy);
```

序列化：

```
  serialize_Json(const Multigrid_BVP_Problem& prob, std::string filename);
```

反序列化：

```
  deserialize_Json(Multigrid_BVP_Problem& prob, std::string filename);
```

2.2 Multigrid_BVP_Solver 类

`Multigrid_BVP_Solver` 类用于求解 Poisson 方程的边值问题，其中保存有一个 `Multigrid_BVP_Problem prob` 表示当前求解的问题，它可以调用两个类，`class Regular_Multigrid_BVPsolver;` `class Irregular_Multigrid_BVPsolver;`，为两种方程的求解器（均为模板类，模板参数为 `int dim`，表示求解问题的维数）。求解器类中均包含 `void solve();` 和 void ErrorAnalysis(); 两个成员函数，用于求解问题实例和误差分析。最终求解结果将保存在 `results` 中。

```
  class Multigrid_BVPsolver {
    Multigrid_BVP_Problem prob;
    RES<1> results1;
    RES<2> results2;
public:
```

```
    void readProblem(const std::string file);
    void printProblem() const;
    void solveProblem(bool print = 0);
    void solveProblem(const Multigrid_BVP_Problem &_prob);
    void solveProblem(std::string File, bool print = 0);
    void saveResults(std::string file = "res.csv") const;
    NUM norm1, norm2, normi;
    void Summary();
};
```

2.3 template<int dim> Regular_Multigrid_BVPsolver 类

2.3.1 限制算子

对于 n 和一个 $(n+1)^d$ 维的向量，计算其在粗网格上的值（为一个 $(n/2+1)^d$ 维向量）。

一维情况：

```
Vec<NUM> Regular_Multigrid_BVPsolver<1>::Restriction(const Vec<NUM>& x, const int N) {
      Vec<NUM> ret;

      ret.resize((N>>1)+1);
      if(prob.Restriction_opt == "injection") {
          for(int i = 0; i < ret.size; ++i) {
              ret[i] = x[i<<1];
          }
      } else { // prob.Restriction_opt == "full_weight"
          ret[0] = x[0]; ret[N>>1] = x[N];
          for(int i = 1; i < ret.size-1; ++i) {
              ret[i] = 0.25*(x[(i<<1)-1]+2*x[i<<1]+x[i<<1|1]);
          }
      }
      return ret;
  }
```

二维情况：

```
Vec<NUM> Regular_Multigrid_BVPsolver<2>::Restriction(const Vec<NUM>& x, const int N) {
      Vec<NUM> ret;

      ret.resize((N>>1)+1);
      if(prob.Restriction_opt == "injection") {
          for(int i = 0; i < ret.size; ++i) {
              ret[i] = x[i<<1];
          }
      } else { // prob.Restriction_opt == "full_weight"
          ret[0] = x[0]; ret[N>>1] = x[N];
          for(int i = 1; i < ret.size-1; ++i) {
              ret[i] = 0.25*(x[(i<<1)-1]+2*x[i<<1]+x[i<<1|1]);
```

```
            }
        }
        return ret;
    }
```

2.3.2 插值算子

对于 n 和一个 $(n+1)^d$ 维的向量，计算其在细网格上的值（为一个 $(2n+1)^d$ 维向量）。

一维情况：

考虑二次插值的方法：对于网格上的点，直接取原先的值；对于其他节点，若其位于线段两侧，采用最近的三个粗网格上点进行插值，否则，采用最近的对称的四个点进行插值。

```
Vec<NUM> Regular_Multigrid_BVPsolver<1>::Interpolation(const Vec<NUM>& x, const int N) {
        Vec<NUM> ret;

        ret.resize(N<<1|1);
        if(prob.Interpolation_opt == "linear") {
            for(int i = 0; i < ret.size; ++i) {
                if(i&1) ret[i] = 0.5*(x[i>>1] + x[(i>>1)+1]);
                else ret[i] = x[i>>1];
            }
        } else { // prob.Interpolation_opt == "quadratic"
            for(int i = 0; i < ret.size; ++i) {
                if(i&1) {
                    int j = i/2;
                    if(i == 1) ret[i] = (3*x[j]+6*x[j+1]-x[j+2])/8.0;
                    else if(i == (N<<1)-1) ret[i] = (3*x[j+1]+6*x[j]-x[j-1])/8.0;
                    else ret[i] = (-x[j-1]+9*x[j]+9*x[j+1]-x[j+2])/16.0;
                }
                else ret[i] = x[i>>1];
            }
        }
        return ret;
}
```

二维情况：

对于网格上的点，直接取原先的值；对于网格线上但不在网格上的点，采用与一维二次插值相同的方法；对于不在网格线上的点，考虑它四周的四个节点，设其左上角为 (I, J)，考虑以下四组插值点：

- (I, J+1), (I+1, J), (I+1, J+1), (I, J+2), (I+1, J+2);
- (I, J), (I+1, J+1), (I, J+1), (I-1, J), (I-1, J+1);
- (I, J+1), (I+1, J), (I, J), (I, J-1), (I+1, J-1);
- (I, J), (I+1, J+1), (I+1, J), (I+2, J), (I+2, J+1).

对于每一组点，如果其在网格内，则对其进行一个插值，最后该点上的值是所有可以取到的插值组得到的值的平均值。

```
Vec<NUM> Regular_Multigrid_BVPsolver<2>::Interpolation(const Vec<NUM>& x, const int N) {
      Vec<NUM> ret;
      int N2 = N<<1;
      ret.resize((N2+1)*(N2+1));
      auto ID = [](int i, int j, int N) -> int {
          return j * (N+1) + i;
      }; // for 2d function
      if(prob.Interpolation_opt == "linear") {
          for(int i = 0; i <= N2; ++i)
          for(int j = 0; j <= N2; ++j) {
              int I = i>>1, J = j>>1;
              if((i&1) && (j&1)) ret[ID(i, j, N2)] = 0.25 * (
                  x[ID(I, J, N)] + x[ID(I+1, J, N)] + x[ID(I+1, J+1, N)] + x[ID(I, J+1, N)])
                      ;
              else if(i&1) ret[ID(i, j, N2)] = 0.5 * (x[ID(I, J, N)] + x[ID(I+1, J, N)]);
              else if(j&1) ret[ID(i, j, N2)] = 0.5 * (x[ID(I, J, N)] + x[ID(I, J+1, N)]);
              else ret[ID(i, j, N2)] = x[ID(I, J, N)];
          }
      } else { // prob.Interpolation_opt == "quadratic"
          for(int i = 0; i <= N2; ++i)
          for(int j = 0; j <= N2; ++j) {
              if((~i&1)&&(~j&1)) ret[ID(i, j, N2)] = x[ID(i>>1, j>>1, N)];
              else if(~i&1) {
                  int I = i>>1, J = j>>1;
                  if(j == 1) ret[ID(i, j, N2)] = (3*x[ID(I, J, N)]+6*x[ID(I, J+1, N)]-x[ID(I
                      , J+2, N)])/8.0;
                  else if(j == N2-1) ret[ID(i, j, N2)] = (3*x[ID(I, J+1, N)]+6*x[ID(I, J, N)
                      ]-x[ID(I, J-1, N)])/8.0;
                  else ret[ID(i, j, N2)] = (-x[ID(I, J-1, N)]+9*x[ID(I, J, N)]+9*x[ID(I, J
                      +1, N)]-x[ID(I, J+2, N)])/16.0;
              } else if(~j&1) {
                  int I = i>>1, J = j>>1;
                  if(i == 1) ret[ID(i, j, N2)] = (3*x[ID(I, J, N)]+6*x[ID(I+1, J, N)]-x[ID(I
                      +2, J, N)])/8.0;
                  else if(i == N2-1) ret[ID(i, j, N2)] = (3*x[ID(I+1, J, N)]+6*x[ID(I, J, N)
                      ]-x[ID(I-1, J, N)])/8.0;
                  else ret[ID(i, j, N2)] = (-x[ID(I-1, J, N)]+9*x[ID(I, J, N)]+9*x[ID(I+1, J
                      , N)]-x[ID(I+2, J, N)])/16.0;
              } else {
                  int I = i>>1, J = j>>1, cnt = 0, id = ID(i, j, N2);
                  if(J + 2 <= N) { cnt += 8;
                      ret[id] += (4*x[ID(I,J+1,N)]+4*x[ID(I+1,J,N)]+2*x[ID(I+1,J+1,N)]-x[ID(
                          I,J+2,N)]-x[ID(I+1,J+2,N)]);
                  }
                  if(I - 1 >= 0) { cnt += 8;
                      ret[id] += (4*x[ID(I,J,N)]+4*x[ID(I+1,J+1,N)]+2*x[ID(I,J+1,N)]-x[ID(I
                          -1,J,N)]-x[ID(I-1,J+1,N)]);
```

```
                    }
                    if(J - 1 >= 0) { cnt += 8;
                        ret[id] += (4*x[ID(I,J+1,N)]+4*x[ID(I+1,J,N)]+2*x[ID(I,J,N)]-x[ID(I,J
                            -1,N)]-x[ID(I+1,J-1,N)]);
                    }
                    if(I + 2 <= N) { cnt += 8;
                        ret[id] += (4*x[ID(I,J,N)]+4*x[ID(I+1,J+1,N)]+2*x[ID(I+1,J,N)]-x[ID(I
                            +2,J,N)]-x[ID(I+2,J+1,N)]);
                    }
                    ret[id] /= (NUM)cnt;
                }
            }
        }
        return ret;
    }
```

2.3.3 V-Cycle

V-cycles 的形式如下：

$$\boldsymbol{v}^h \leftarrow \mathrm{VC}(\boldsymbol{v}^h, \boldsymbol{f}^h, \nu_1, \nu_2),$$

(V-1) 对 $A^h\boldsymbol{u}^h = \boldsymbol{f}$ 在 Ω^h 上松弛 ν_1 次；

(V-2) 如果 Ω^h 是粗网格，转 4，否则：

$$\boldsymbol{f}^{2h} \leftarrow I_h^{2h}(\boldsymbol{f}^h - A^h\boldsymbol{v}^h),$$

$$\boldsymbol{v}^{2h} \leftarrow 0,\ \boldsymbol{v}^{2h} \leftarrow \mathrm{VC}^{2h}(\boldsymbol{v}^{2h}, \boldsymbol{f}^{2h}, \nu_1, \nu_2).$$

(V-3) 校正：$\boldsymbol{v}^h \leftarrow \boldsymbol{v}^h + I_{2h}^h\boldsymbol{v}^{2h}$；

(V-4) 对 $A^h\boldsymbol{u}^h = \boldsymbol{f}^h$ 松弛 ν_2 次。

```
void VC(Vec<NUM> &vh, const Vec<NUM> &fh, const int nu1, const int nu2, const int N) {
    int id = calid[N];
    cal[id].GoIter(vh, fh, nu1);
    if(N > 2) {
        Vec<NUM> f2h = Restriction((fh - cal[id].A * vh), N);
        Vec<NUM> v2h(prob.Dimension==1?N/2+1:(N/2+1)*(N/2+1));
        VC(v2h, f2h, nu1, nu2, N/2);
        vh = vh + Interpolation(v2h, N/2);
    }
    cal[id].GoIter(vh, fh, nu2);
}
```

2.3.4 FMG Cycle

Full Multigrid V-Cycle 形式入下：

$$\mathrm{v}^h \leftarrow \mathtt{FMG}^h(\mathrm{f}^h, \nu_1, \nu_2),$$

(F-1) 若 Ω^h 是粗网格，设置 $\mathrm{v}^h \leftarrow 0$，转 3，否则

$$\mathrm{f}^{2h} \leftarrow I_h^{2h}\mathrm{f}^{2h},$$

$$\mathrm{v}^{2h} \leftarrow \mathtt{FMG}^{2h}(\mathrm{f}^{2h}, \nu_1, \nu_2);$$

(F-2) 校正：$\mathrm{v}^h \leftarrow \mathrm{v}^h + I_{2h}^h \mathrm{v}^{2h}$;

(F-3) 执行一个 V 循环使用初始值 v^h：$\mathrm{v}^h \leftarrow \mathtt{VC}^h(\mathrm{v}^h, \mathrm{f}^h, \nu_1, \nu_2)$。

```
void FMG(Vec<NUM> &vh, const Vec<NUM> &fh, const int nu1, const int nu2, const int N) {
    int id = calid[N];
    if(N > 2) {
        Vec<NUM> f2h = Restriction(fh, N);
        Vec<NUM> v2h(prob.Dimension==1?N/2+1:(N/2+1)*(N/2+1));
        FMG(v2h, f2h, nu1, nu2, N/2);
        vh = Interpolation(v2h, N/2);
    }
    VC(vh, fh, nu1, nu2, N);
}
```

2.3.5 求解过程

首先根据 `initial_guess` 设置初始 u 的值，每次迭代，使用当前的残差进行求解，并把结果加入 u 中，若当前加入的向量的范数小于设置的值，则退出。最后的 b 就是当前的残差。

```
// 设置初始 Guess 向量 u
b = b - cal[0].A * u;

for(int iter = 0; iter < prob.Max_Iteration; ++iter) {
    Vec<NUM> v(len);
    if(prob.Cycle_Type == "V") {
        VC(v, b, 5, 5, n);
    } else {
        FMG(v, b, 5, 5, n);
    }
    u = u + v;
    b = b - cal[0].A * v;
    if(Norm2Vec(v) < prob.Accuracy) break;
}
```

2.4 其他组件

2.4.1 SparseMat 稀疏矩阵类

本项目中涉及到的矩阵 A 都是稀疏矩阵（即对于 $O(n^2)$ 级别的矩阵，非零项为 $O(n)$ 个）。为了使矩阵乘向量的复杂度为 $O(n)$，我们优化了矩阵的存储方式，使用 *classSparseMat*。见 (`/include/linear.hpp`)。具体的，我们使用 `MatLine` 来表示矩阵的一行，我们使用的 Hash 的方式，将一行中，列下标模

5 余数相同的列保存在一起（这样是为了方便矩阵的随机访问），在矩阵与向量的乘法中，直接依次取出矩阵中的非零元素与向量中的相应位置作乘法，并把其贡献加入结果向量中。

```
template<class Type>
class SparseMat {
    class MatLine { // 表示矩阵的一行
        int len;
        Type zero;
    public:
        vector<pair<int,Type>> a[5];
        MatLine(int _len = 0) : len(_len), zero(0) {}
        Type& operator [] (const int& pos);
        const Type& operator [] (const int& pos) const;
        void setv(int pos, Type v);
        Type v(int pos);
        Type indot(const Vec<Type>& b) const;
        MatLine operator * (const Type& x) const;
        MatLine operator - () const;
        MatLine operator / (const Type& x) const;
        friend ostream& operator << (ostream& o, const MatLine &line);
        operator Vec<Type>() const;
    };
    vector<MatLine> a;
public:
    int d, d2; // the dimension of column vector and row vector.
    typedef Vec<Type> VecT;
    typedef Mat<Type, Type> MatT;
    SparseMat() : d(0), d2(0), a(0) {}
    SparseMat(const int& _d) : d(_d), d2(_d), a(_d, _d) {}
    SparseMat(const int& _d, const int& _d2) : d(_d), d2(_d2), a(_d, _d2) {}

    MatLine& operator[] (const int &x) { return a[x]; }
    const MatLine& operator[] (const int &x) const { return a[x]; }
    auto begin() const -> decltype(a.begin()) {
        return a.begin();
    }
    auto begin() -> decltype(a.begin()) {
        return a.begin();
    }
    auto end() const -> decltype(a.end()) {
        return a.end();
    }
    auto end() -> decltype(a.end()) {
        return a.end();
    }

    VecT operator * (const VecT& y) const;
    SparseMat<Type> operator * (const Type &x) const ;
```

```
    SparseMat<Type> operator - () const ;
    friend SparseMat<Type> operator * (const Type &x, const SparseMat<Type> &y);
    SparseMat<Type> operator / (const Type& x) const;
    operator MatT() const;
    void setv(int i, int j, Type v) { a[i].setv(j, v); }
    void v(int i, int j) { return a[i].v(j); }
};
```

2.4.2 带权 Jacobi 迭代

定义 $D, -L, -U$ 为 A 的对角，下三角，上三角部分（$A = D - L - U$）。定义：

$$\begin{cases} T_J = D^{-1}(L+U), \\ \boldsymbol{c} = D^{-1}\boldsymbol{b}. \end{cases}$$

带权 Jacobi 迭代为如下形式的不动点迭代：

$$\boldsymbol{x}_* = T_j \boldsymbol{x}^{(k)} + \boldsymbol{c},$$

$$\boldsymbol{x}^{k+1} = (1-\omega)\boldsymbol{x}^{(k)} + \omega \boldsymbol{x}_* = [(1-\omega)I + \omega T_J]\boldsymbol{x}^{(k)} + \boldsymbol{c}.$$

这里都取 $\omega = \frac{2}{3}$。我们使用 `Jacobi_Iteration::GoIter(vec<NUM> &x, const Vec<NUM>&b, int iter = 1)` 表示从 `x` 开始，对于方程 $Ax = b$，迭代 `iter` 次。因为问题中对于每种网格的 A 矩阵总是固定的，但是右端项总是不相同，因此考虑将 b 作为参数输入，而把 A 作为类的成员。

```
class Jacobi_Iteration {
    SparseMat<NUM> T, nD;
    Vec<NUM> c;
    NUM w;
    int n;
public:
    SparseMat<NUM> A;
    Jacobi_Iteration(const SparseMat<NUM>&_A, const int& _n, const NUM& _w = 2.0/3.0) :
                    A(_A), w(_w), n(_n), T(-_A), nD(_n+1) {
                       for(int i = 0; i <= n; ++i) {
                           T[i][i] = 0;
                           nD[i][i] = 1.0 / _A[i][i];
                           T[i] = T[i] * nD[i][i];
                       }
                    }
    void GoIter(Vec<NUM> &x, const Vec<NUM>& b, int iter = 1) {
        c = nD * b;
        Vec<NUM> y;
        for(int i = 0; i < iter; ++i) {
            y = T * x + c;
            x = w * y + (1-w) * x;
        }
    }
};
```

2.4.3 function 类及字符串解析方法

求解问题中需要用到的 2 元函数为 `/include/function.hpp` 中的 `Function2D` 类的派生类，因为涉及到从 JSON 文档中读取函数的表达式，项目调用了 `/include/function_generator/ExecCode2.hpp` 中的 `FromString` 类，该类用于对一个字符串建立表达式树，并在之后求解的时候可以快速根据表达式树进行求值。对于多元函数，可以调用 `FromString<double, double>::cal2(const map<string, double>& pars)` 进行求值。

```
class func2D : Function2D<double, double> {
  FromString<double, double> getvalue;
public:
  func2D (const std::string &_str) : getvalue(_str) {}
  func2D () : getvalue() {}
  void set(const std::string &_str) { getvalue = FromString<double, double>(_str); }
  double operator() (const double &x, const double &y) const {
      double xx = x, yy = y;
      return getvalue.cal2({{"x", xx}, {"y", yy}});
  }
};
```

3 样例测试及结果

该部分说明此程序包在样例测试中的求解表现。

3.1 测试说明

本程序包对于四个 Poisson 方程（三个一维方程，一个二维方程）关于不同的边值条件提供了若干测试数据，测试数据位于目录 `/data` 下。

在根目录下编译并求解所有样例：

```
make all # compile
make run
```

该指令将会编译 `/test.cpp`，并且调用 `test` 按照字典序依次运行所有的样例（`/data` 下的所有 JSON 文件），并且保存样例的误差分析结果至 `/res/test_analysis.txt`。测试输出（离散网格点值的求解结果）将会以表格的形式保存至 `/res/<Input File Name>.csv`。

对测试输出进行绘图：

```
make plot
```

该指令将会对保存在 `/res/` 目录下的所有 csv 输出文件进行绘图，并且将结果保存在 `/res/` 下，命名与原文件相同，类型为 png。

所有测试共花费 1 min 左右，程序包提供了一份测试结果的备份，位于 `/res_bac` 目录，另有一份所有样例的误差报告 `terminal.out`，便于用户直接查看。

因报告篇幅有限，下只展示部分测试结果的收敛性分析。

3.2 测试样例 1

测试样例为 `data/1a-*.json`，求解结果见 `terminal.out`。该部分指定 Accuracy 为 1e-8。

问题：

$$u(x) = \exp(\sin(x)), \Omega = [0, 1]$$

3.2.1 Dirichlet 边值条件

$$\begin{cases} -\Delta u = \exp(\sin(x))(\sin(x) - \cos^2(x)), \text{in } \Omega \\ u = \exp(\sin(x)), \text{on } \partial\Omega \end{cases}$$

3.2.2 Neumann 边值条件

$$\begin{cases} -\Delta u = \exp(\sin(x))(\sin(x) - \cos^2(x)), \text{in } \Omega \\ \frac{\partial u}{\partial \boldsymbol{n}} = -1, x = 0 \\ \frac{\partial u}{\partial \boldsymbol{n}} = \exp(\sin(1))\cos(1), x = 1 \end{cases}$$

3.2.3 混合边值条件

$$\begin{cases} -\Delta u = \exp(\sin(x))(\sin(x) - \cos^2(x)), \text{in } \Omega \\ \frac{\partial u}{\partial \boldsymbol{n}} = -1, x = 0 \\ u = \exp(\sin(1)), x = 1 \end{cases}$$

3.2.4 V-Cycle 和一阶算子的测试

一维 Dirichlet 边界条件，V-Cycle，linear 插值，injection 限制：

网格大小	残差	误差	运行时间 (ms)	收敛阶
32	1.00e-10	4.94e-05	0.247	-
64	2.00e-10	1.24e-05	0.347	1.99
128	1.00e-09	3.09e-06	0.551	2.00
256	3.80e-09	7.73e-07	0.910	2.00
512	1.54e-08	1.93e-07	1.472	2.00

一维 Neumann 边界条件，V-Cycle，linear 插值，injection 限制：

网格大小	残差	误差	运行时间 (ms)	收敛阶
32	2.22e-03	4.08e-04	0.542	-
64	6.91e-04	1.02e-04	0.752	2.00

128	2.15e-04	2.54e-05	1.078	2.00
256	6.66e-05	6.35e-06	1.725	2.00
512	1.59e-05	1.59e-06	2.911	2.00

一维 Mixed 边界条件，V-Cycle，linear 插值，injection 限制：

网格大小	残差	误差	运行时间 (ms)	收敛阶
32	1.26e-08	1.63e-04	0.441	-
64	7.80e-09	4.34e-05	0.634	1.91
128	1.80e-08	1.12e-05	0.885	1.95
256	3.88e-08	2.84e-06	1.413	1.98
512	2.04e-08	7.16e-07	2.536	1.99

3.2.5 FMG-Cycle 和二阶算子的测试

一维 Neumann 边界条件，FMG，quadratic 插值，Full-weight 限制：

网格大小	残差	误差	运行时间 (ms)	收敛阶
32	2.25e-03	4.07e-04	1.115	-
64	7.06e-04	1.02e-04	1.799	2.00
128	2.20e-04	2.54e-05	2.606	2.00
256	6.86e-05	6.34e-06	4.124	2.00
512	2.13e-05	1.58e-07	6.643	2.00

3.2.6 总结

多重网格算法具有二阶的收敛阶。且在 Dirichlet 和 Mixed 边界条件下均可使残差降到 10^{-10}；在 Neumann 边界条件下残差收敛到定值，即最小二乘问题 $\min \|Au - f\|$ 的解的残差。

在一维问题中，网格较粗的情况下，FMG-Cycle 和高阶算子的收敛速度优势并不显著。又因为 V-Cycle 和低阶算子的计算量明显小于高阶算子，所以第一小节的测试结果明显优于第二小节。

在最细的网格上，将 Accuracy 逐渐降至 1^{-16}，设置最大迭代次数为 100 （足够大）。执行命令：

```
./test data/1a-DFqf-512.json data/1a-DFqf-512-2.json data/1a-DFqf-512-3.json
       data/1a-DFqf-512-4.json data/1a-DFqf-512-5.json
```

得到结果如下：

网格大小	Accuracy	残差	误差	运行时间 (ms)
512	1e-8	3e-10	1.932e-7	4.255
512	1e-10	3e-10	1.932e-7	3.791
512	1e-12	<1e-10	1.932e-7	3.771

512	1e-14	<1e-10	1.932e-7	3.279
512	1e-16	<1e-10	1.932e-7	3.643

可以发现，虽然残差仍然在下降，但是误差保持不变，这是因为，当 $n = 512$ 时，由于 $h^2 = \frac{1}{262144} \approx 4e-6$，此时误差主要来自于截断误差，所以减小对残差的限制无法进一步减少总体误差。

3.3 测试样例 2

测试样例为 `data/1b-*.json`，求解结果见 `terminal.out`。该部分指定 Accuracy 为 1e-12。

问题：

$$u(x) = \ln(1+x^2), \Omega = [0,1]$$

3.3.1 Dirichlet 边值条件

$$\begin{cases} -\Delta u = -2(1-x^2)/(1+x^2)^2, \text{in } \Omega \\ u = \log(1+x^2), \text{on } \partial\Omega \end{cases}$$

3.3.2 Neumann 边值条件

$$\begin{cases} -\Delta u = -2(1-x^2)/(1+x^2)^2, \text{in } \Omega \\ \frac{\partial u}{\partial \boldsymbol{n}} = 0, x = 0 \\ \frac{\partial u}{\partial \boldsymbol{n}} = 1, x = 1 \end{cases}$$

3.3.3 混合边值条件

$$\begin{cases} -\Delta u = -2(1-x^2)/(1+x^2)^2, \text{in } \Omega \\ \frac{\partial u}{\partial \boldsymbol{n}} = 0, x = 0 \\ u = \ln(2), x = 1 \end{cases}$$

3.3.4 V-Cycle 和一阶算子的测试

一维 Mixed 边界条件，V-Cycle，linear 插值，injection 限制：

网格大小	残差	误差	运行时间 (ms)	收敛阶
32	<1e-10	7.57e-05	1.115	-
64	<1e-10	2.95e-05	1.799	1.36
128	<1e-10	8.76e-06	2.606	1.75
256	<1e-10	2.37e-06	4.124	1.89

512	<1e-10	6.14e-07	6.643	1.95

3.4 测试样例 3

测试样例为 `data/1c-*.json`，求解结果见 `terminal.out`。该部分指定 Accuracy 为 1e-12。

问题：

$$u(x) = \sin^2(x), \Omega = [0, 1]$$

3.4.1 Dirichlet 边值条件

$$\begin{cases} -\Delta u = -2\cos(2x), \text{in } \Omega \\ u = \sin^2(x), \text{on } \partial\Omega \end{cases}$$

3.4.2 Neumann 边值条件

$$\begin{cases} -\Delta u = -2\cos(2x), \text{in } \Omega \\ \frac{\partial u}{\partial \boldsymbol{n}} = 0, x = 0 \\ \frac{\partial u}{\partial \boldsymbol{n}} = \sin(2), x = 1 \end{cases}$$

3.4.3 混合边值条件

$$\begin{cases} -\Delta u = \exp(\sin(x))(\sin(x) - \cos^2(x)), \text{in } \Omega \\ \frac{\partial u}{\partial \boldsymbol{n}} = 0, x = 0 \\ u = \sin^2(1), x = 1 \end{cases}$$

3.5 测试样例 4

测试样例为 `data/2-*.json`，求解结果见 `terminal.out`。该部分指定 Accuracy 为 1e-12。

$$u(x, y) = \exp(y + \sin x), \Omega = [0, 1]^2$$

3.5.1 Dirichlet 边值条件

$$\begin{cases} -\Delta u = -(1 - \sin x + \cos^2 x)\exp(\sin x + y), \text{in } \Omega \\ u = \exp(\sin x + y), \text{on } \partial\Omega \end{cases}$$

3.5.2 Neumann 边值条件

$$
\begin{cases}
-\Delta u = -(1 - \sin x + \cos^2 x)\exp(\sin x + y), \text{in } \Omega \\
\frac{\partial u}{\partial \boldsymbol{n}} = -\exp(\sin x), y = 0 \\
\frac{\partial u}{\partial \boldsymbol{n}} = -\exp(y), x = 0 \\
\frac{\partial u}{\partial \boldsymbol{n}} = \cos 1 \cdot \exp(\sin 1 + y), x = 1 \\
\frac{\partial u}{\partial \boldsymbol{n}} = \exp(\sin x + 1), y = 1
\end{cases}
$$

3.5.3 混合边值条件

$$
\begin{cases}
-\Delta u = -(1 - \sin x + \cos^2 x)\exp(\sin x + y), \text{in } \Omega \\
u = \exp(\sin x + y), y = 0 \\
u = \exp(\sin x + y), x = 0 \\
\frac{\partial u}{\partial \boldsymbol{n}} = \cos 1 \cdot \exp(\sin 1 + y), x = 1 \\
\frac{\partial u}{\partial \boldsymbol{n}} = \exp(\sin x + 1), y = 1
\end{cases}
$$

3.5.4 V-Cycle 和一阶算子的测试

二维 Dirichlet 边界条件，V-Cycle，linear 插值，injection 限制：

网格大小	残差	误差	运行时间 (ms)	收敛阶	运行时间收敛阶
32	<1e-10	3.45e-05	3.753	-	-
64	<1e-10	8.62e-06	15.811	2.00	2.07
128	<1e-10	2.16e-06	71.655	2.00	2.18
256	<1e-10	5.39e-07	303.294	2.00	2.08
512	<1e-10	1.35e-07	1289.425	2.00	2.09

二维 Neumann 边界条件，V-Cycle，linear 插值，injection 限制：

网格大小	残差	误差	运行时间 (ms)	收敛阶	运行时间收敛阶
32	2.89e-03	8.56e-04	5.557	-	-
64	9.71e-04	2.16e-04	18.187	1.99	1.71
128	3.49e-04	5.42e-05	105.937	1.99	2.54
256	1.95e-04	1.37e-05	471.695	1.98	2.15
512	2.86e-03	3.75e-06	1818.714	1.87	1.94

二维 Mixed 边界条件，V-Cycle，linear 插值，injection 限制：

网格大小	残差	误差	运行时间 (ms)	收敛阶	运行时间收敛阶
32	7.81e-07	1.06e-03	4.795	-	-
64	4.38e-06	2.63e-03	17.392	2.01	1.86
128	1.95e-05	6.58e-05	69.162	2.00	1.99
256	7.36e-05	1.70e-05	336.728	1.95	2.28
512	2.47e-04	5.52e-06	1598.827	1.62	2.24

3.5.5 FMG 和二阶算子的测试

测试函数 $u(x, y) = \exp(\sin(x) + y)$，二维 Mixed 边界条件，FMG，linear 插值，injection 限制。

网格大小	残差	误差	运行时间 (ms)	收敛阶	运行时间收敛阶
32	<1e-10	1.06e-03	5.499	-	-
64	<1e-10	2.63e-03	17.582	2.01	1.67
128	<1e-10	6.54e-05	84.448	2.01	2.26
256	<1e-10	1.63e-05	342.296	2.00	2.03
512	<1e-10	4.07e-06	1420.509	2.00	2.05

测试函数 $u(x, y) = \exp(\sin(x) + y)$，二维 Mixed 边界条件，FMG，quadratic 插值，Full-weight 限制。

网格大小	残差	误差	运行时间 (ms)	收敛阶	运行时间收敛阶
32	<1e-10	1.06e-03	5.341	-	-
64	<1e-10	2.63e-03	18.999	2.01	1.83
128	<1e-10	6.54e-05	65.258	2.01	1.78
256	<1e-10	1.63e-05	314.449	2.00	2.27
512	<1e-10	4.07e-06	1865.895	2.00	2.57

3.5.6 分析与总结

多重网格在二维问题中仍具有二阶的收敛阶，且时间复杂度基本上和网格格点数量线性相关。

二维问题中，FMG-Cycle 相比 V-Cycle 的优势可以体现：二维的 Neumann 和 mixed 边界条件下，$n = 128, 256, 512$ 时，V-Cycle 在循环 20 次后残差没有收敛，导致计算出的解收敛阶偏低，明显小于二阶。而换成 FMG-Cycle 后，同样迭代 20 次，残差收敛至 10^{-10} 以下，解的收敛阶达到了二阶。

参考文献

[1] 张庆海. “Notes on Numerical Analysis and Numerical Methods for Differential Equations”. In: (2024).