微分方程数值解 - 第七章上机作业

樊睿强基数学 2001 班

2023 年 3 月 24 日

摘要

本项目实现了对规则区域（以 $[0,1]^2$ 为例）和非规则区域（以 $[0,1]^2 \setminus D$ 为例）上 Poisson 方程的 Dirichlet 和 Neumann 边值问题的有限差分法求解。

1 规则区域的 Poisson 方程

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

1.1 差分格式

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

将求解未知函数 u 的问题转化为求解 u 在网格点 $u(ih, jh)$（$0 \le i \le n+1, 0 \le j \le n+1$，且 i, j 不同时为 0 或 $n+1$）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

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

其中 $1 \le i, j \le n$。

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。

1.2 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

$$u(ih, 0) = g(ih, 0), 1 \le i \le n, \tag{4}$$
$$u(ih, 1) = g(ih, 1), 1 \le i \le n, \tag{5}$$
$$u(0, jh) = g(0, jh), 1 \le j \le n, \tag{6}$$
$$u(1, jh) = g(1, jh), 1 \le j \le n. \tag{7}$$

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 $O(h^2)$。

1.3 Neumann 边界条件

$$\frac{\partial u}{\partial n}|_{\partial\Omega} = g. \tag{8}$$

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：

$$\frac{1}{2h}(-3u(ih, 0) + 4u(ih, h) - u(ih, 2h)) = g(ih, 0), \tag{9}$$
$$\frac{1}{2h}(-3u(ih, 1) + 4u(ih, 1-h) - u(ih, 1-2h)) = g(ih, 1), \tag{10}$$
$$\frac{1}{2h}(-3u(0, jh) + 4u(h, jh) - u(2h, jh)) = g(0, jh), \tag{11}$$
$$\frac{1}{2h}(-3u(1, jh) + 4u(1-h, jh) - u(1-2h, jh)) = g(1, jh). \tag{12}$$

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。

为解决这个问题，可直接将矩阵对角线上的某个系数加 1 破坏奇异性。

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 $O(h^2)$。

2 非规则区域的 Poisson 方程

$$-u_{xx} - u_{yy} = f, u \in \Omega = [0,1]^2 \setminus D. \tag{13}$$

约定其中 D 是一个完全包含在 $[0,1]^2$ 中的圆，且其边界与 $[0,1]^2$ 的距离至少为 $2h$。

2.1 差分格式

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

和规则区域相比，不规则区域主要有以下两个问题：

- 部分网格点可能不在区域内，我们不关心这些点的值（或者说这些点的值是“非法的”），所以也不能把它们引入方程。
- 非规则边界与网格的交点不一定是网格点，我们要将非规则边界条件离散到这些点上。以这些点来代替那些不在区域内的的点来对区域内部的点的 Poisson 方程进行差分。

我们分别考虑 u_{xx} 和 u_{yy} 的差分格式。

设 $(ih, jh) = (x, y)$ 是一个内部格点。

因为区域非规则，所以 $(x-h, y)$ 和 $(x+h, y)$ 不一定在区域内部。设其左右两侧的格点分别为 $(x-\alpha h, y)$ 和 $(x+\beta h, y)$。

考虑用这三个格点的值对 u_{xx} 进行插值。我们设

$$u_{xx}(x,y) = au(x,y) + b(x-\alpha h, y) + c(x+\beta h, y) + o(h^2). \tag{14}$$

则

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & -\alpha h & \beta h \\ 0 & \dfrac{\alpha^2 h^2}{2} & \dfrac{\beta^2 h^2}{2} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{15}$$

解得

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{2}{(\alpha+\beta)\alpha\beta h^2} \begin{bmatrix} \alpha+\beta \\ -\beta \\ -\alpha \end{bmatrix}. \tag{16}$$

同理可以插值 u_{yy}。

这样就建立了内部网格点的 Poisson 方程的差分格式。

2.2 Dirichlet 边界条件

Dirichlet 边界条件比较简单，直接对所有边界上的离散点列出 $u(x,y) = g(x,y)$ 即可。

根据教材 Exercise 7.63，上述差分格式的在非正则点（即周围有非网格点）处的 LTE 为 $O(h)$，其他点处 LTE 为 $O(h^2)$。但总体误差是 $O(h^2)$。

2.3 Neumann 边界条件

非规则 Neumann 边界条件的处理是整个项目中最困难的部分。

为保证边界点处 $O(h^2)$ 的 LTE，我们必须设计 $\dfrac{\partial u}{\partial n} = u_x \cos\alpha + u_y \sin\alpha$ 的二阶差分格式。

即我们要满足 $u(x,y) = L_h(x,y) + o(h^2)$。

注意到 $u(x,y)$ 关于 x, y 的二阶泰勒展开有 6 项，因此我们要设计一个 6 个点的差分格式。

我们以非规则边界点为 $(x,y) = (ih, y), x > x_O, y > y_O$ 为例（共 8 种情况，需分别讨论）。

设 (ih, jh) 为在 (ih, y) 且距离 (ih, y) 最近的网格点。设 $(ih, jh) = (x, y + \theta h)$。则

$$\begin{aligned}\frac{\partial u}{\partial n}(x,y) &= c_0u(x,y) + c_1u(x,y+\theta h) + c_2u(x,y+(1+\theta)h)\\ &+ c_3u(x+h,y+\theta h) + c_4u(x+h,y+(1+\theta)h)\\ &+ c_5u(x+2h,y+\theta h) + o(h^2).\end{aligned} \tag{17}$$

则有

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1\\ 0 & 0 & 0 & h & h & 2h\\ 0 & \theta h & (1+\theta)h & \theta h & (1+\theta)h & \theta h\\ 0 & 0 & 0 & \frac{h^2}{2} & \frac{h^2}{2} & 2h^2\\ 0 & \frac{\theta^2h^2}{2} & \frac{(1+\theta h)^2h^2}{2} & \frac{\theta^2h^2}{2} & \frac{(1+\theta h)^2h^2}{2} & \frac{\theta^2h^2}{2}\\ 0 & 0 & 0 & \theta h^2 & (1+\theta)h^2 & 2\theta h^2 \end{bmatrix} \begin{bmatrix} c_0\\ c_1\\ c_2\\ c_3\\ c_4\\ c_5 \end{bmatrix} = \begin{bmatrix} 0\\ \cos\alpha\\ \sin\alpha\\ 0\\ 0\\ 0 \end{bmatrix}. \tag{18}$$

这个方程手算求解比较困难，求解八个方程计算量过大，因此在程序设计时，采用构造矩阵并运行高斯消元法求解 6 阶线性方程组的方法求出系数。

3 程序设计细节

3.1 二元函数的实现

类似第一章一元函数的实现，设计 `Function_2D` 抽象类，实现二元函数的求函数值和求偏导数值。详见文件目录中 `function.h`。

```
#ifndef FUNCTION
#define FUNCTION
const double delta = 1e-8;
template <class type>
class Function{
public:
    virtual type operator ()(const type& x) const = 0;
    virtual type d(const type& x, const int& k = 1) const {
        if (k == 1) return ((*this)(x+delta) - (*this)(x-delta)
            ) / (2*delta);
```

```
10          if (k == 2) return ((*this)(x+delta) - 2*(*this)(x) +
                (*this)(x-delta)) / (delta*delta);
11          throw 0;
12      }
13  };
14  template <class type>
15  class Function_2D{
16  public:
17      virtual type operator ()(const type& x, const type& y)
            const = 0;
18      virtual type partial(const type& x, const type& y, const
            int& i, const int& j) const {
19          if (i == 1 && j == 0) return ((*this)(x+delta, y) - (*
                this)(x-delta, y)) / (2*delta);
20          if (i == 2 && j == 0) return ((*this)(x+delta, y) -
                2*(*this)(x, y) + (*this)(x-delta, y))/ (delta*
                delta);
21          if (i == 0 && j == 1) return ((*this)(x, y+delta) - (*
                this)(x, y-delta)) / (2*delta);
22          if (i == 0 && j == 2) return ((*this)(x, y+delta) -
                2*(*this)(x, y) + (*this)(x, y-delta))/ (delta*
                delta);
23          throw 0;
24      }
25  };
26  #endif
```

3.2 平面几何的程序实现

本题需要一些平面向量和圆的基本运算。例如向量的加减，圆和水平竖直直线的交点、判断点是否在圆内等。

因为机器误差，两个相等的点可能在一系列运算之后相差一个机器精度级别的小量。因此我们定义 `sgn` 函数，当两个实数相差不超过 10^{-12} 时，就认为它们是相等的（这其实是一个危险的操作，因为模糊的 `<` 运算可能会导致序关系混乱。本次作业运算比较简单，相等的点误差不会太大，所以不会出现这种情况）。同理当一个点和圆周的距离不超过 10^{-12} 时，就认为它在圆上。详见文件目录中 `geo_2D.h`。

```cpp
#ifndef GEO2D
#define GEO2D
#include <bits/stdc++.h>
#include "function.h"
using namespace std;

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();
}

```

```
38 struct circle {
39     pnt o;
40     double r;
41     circle(double x=0, double y=0, double r=0) : o(x,y), r(r)
           {}
42     double Y1(const double& x) const {return o.y - sqrt(max
           (0.0, r*r - (x-o.x)*(x-o.x)));}
43     double Y2(const double& x) const {return o.y + sqrt(max
           (0.0, r*r - (x-o.x)*(x-o.x)));}
44     double X1(const double& y) const {return o.x - sqrt(max
           (0.0, r*r - (y-o.y)*(y-o.y)));}
45     double X2(const double& y) const {return o.x + sqrt(max
           (0.0, r*r - (y-o.y)*(y-o.y)));}
46     double arg(const pnt& p) const {return atan2(p.x-o.x, p.y-o
           .y);}
47 };
48
49 inline int pnt_to_circle(const pnt &p, const circle &c) {
50     return sgn(dis(p, c.o) - c.r);
51 }
52 #endif
```

3.3 数值代数部分

自编函数库。主要用到了矩阵类`Matrix`、列向量类`Colvec`和列主元高斯消元`Gauss_Improved_Solve`。详见 `Matrix.h`。由于代码过长，不在此给出完整代码。

3.4 BVPSolver 抽象类

定义 BVPSolver 抽象类，所有 BVPSolver 都是它的派生。

```
1 #ifndef BVPSOLVER
2 #define BVPSOLVER
3 #include <bits/stdc++.h>
4 #include "function.h"
5 using namespace std;
6
```

```
class BVPSolver {
protected:
    virtual void ID_Generator() = 0;
    virtual void Matrix_Generator() = 0;
public:
    virtual void Solve() = 0;
    virtual void Summary(Function_2D<double>&, bool detail) =
        0;
};
#endif
```

BVPSolver 的基本设计思路：

成员变量 `int n`、`Function_2D<double>&f,&g` 和 `string cond`（边界条件）由调用者给出，$h = \dfrac{1}{n+1}$。

成员变量 `map<pnt,int> id` 是网格点以及边界离散点的编号。

成员变量 `Matrix<double> coef Colvec<double> rhs` 分别表示离散后的线性方程组的系数矩阵和右端向量，`Colvec<double> sol`为线性方程组的解。

成员函数 `ID_Generator()` 计算所有网格点和边界离散点并生成编号。

成员函数 `Matrix_Generator()` 生成系数矩阵和右端项。

成员函数 `Solver()` 先生成编号，再生成系数矩阵和右端项，最后调用高斯消元法解方程。

成员函数 `Summary(Function_2D<double>&, bool detail = 0)`输入真值，输出 BVP 的区域、边界条件、网格密度和误差。`detail=1` 则输出所有离散点及对应的值。

很明显，最难的部分在于 `Matrix_Generator()` 函数。

3.5 规则区域 BVPSolver

构造函数的参数为`n,f,g,cond`。分别是每行（每列）网格数量、右端函数、边界函数和边界条件。`cond`是一个长为 4 的字符串，依次为正方形的下、左、上、右边界的边界条件。

按前面的设计思路，先将除四个角外的所有网格点进行标号。

定义辅助函数`Laplace_Normal_Discretor(pnt p0, pnt p1, pnt p2, pnt p3, pnt p4)`，建立一个 Poisson 方程在一个内部点`p0`处的离散方程。`p1,p2,p3,p4`是它左右上下的四个相邻的网格点。

再定义辅助函数 `Dirichlet_Normal_Discretor(pnt p0)` 和 `Neumann_Normal_Discretor(pnt p0,` 分别是建立 Dirichlet 和 Neumann 边界条件在边界点`p0`处的离散方程。`p1`和`p2`是与边界点差 h 和 $2h$ 的内部点。

这样我们就能解决规则区域的任意（同一边边界条件类型相同的）混合问题。

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

class Normal_BVPSolver : public BVPSolver{
private:
    int n, d;
    double h;
    Function_2D<double> &f, &g;
    Matrix<double> coef;
    Colvec<double> rhs;
    Colvec<double> sol;
    map<pnt, int> id;
    vector<pnt> ps;
    string cond;
    void ins(pnt p) {
        if(!id.count(p)) id[p] = d++, ps.push_back(p);
    }
    void ID_Generator() {
        for (int i = 0; i <= n+1; ++ i)
            for (int j = 0; j <= n+1; ++ j)
                if (i!=0&&i!=n+1 || j!=0&&j!=n+1)
                    ins(pnt(i*h,j*h));
    }
    void Laplace_Normal_Discretor(pnt p0, pnt p1, pnt p2, pnt
        p3, pnt p4) {
        int i0 = id[p0], i1 = id[p1], i2 = id[p2], i3 = id[p3],
            i4 = id[p4];
        coef[i0][i0] += 4 / (h*h);
        coef[i0][i1] -= 1 / (h*h);
```

```
31          coef[i0][i2] -= 1 / (h*h);
32          coef[i0][i3] -= 1 / (h*h);
33          coef[i0][i4] -= 1 / (h*h);
34          rhs[i0] = f(p0.x, p0.y);
35      }
36      void Dirichlet_Normal_Discretor(pnt p0) {
37          int i0 = id[p0];
38          coef[i0][i0] = 1;
39          rhs[i0] = g(p0.x, p0.y);
40      }
41      void Neumann_Normal_Discretor(pnt p0, pnt p1, pnt p2) {
42          int i0 = id[p0], i1 = id[p1], i2 = id[p2];
43          coef[i0][i0] -= 1.5 / h;
44          coef[i0][i1] += 2 / h;
45          coef[i0][i2] -= 0.5 / h;
46          rhs[i0] = g(p0.x, p0.y);
47      }
48      void Matrix_Generator() {
49          coef = Matrix<double> (d, d);
50          rhs = Colvec<double> (d);
51          for (int i = 1; i <= n; ++ i)
52              for (int j = 1; j <= n; ++ j)
53                  Laplace_Normal_Discretor(pnt(i*h,j*h), pnt((i
                        -1)*h,j*h), pnt((i+1)*h,j*h), pnt(i*h,(j-1)
                        *h), pnt(i*h,(j+1)*h));
54          for (int i = 1; i <= n; ++ i)
55              if (cond[0] == 'D') Dirichlet_Normal_Discretor(pnt(
                    i*h, 0));
56              else Neumann_Normal_Discretor(pnt(i*h, 0), pnt(i*h,
                     h), pnt(i*h, 2*h));
57
58          for (int j = 1; j <= n; ++ j)
59              if (cond[1] == 'D') Dirichlet_Normal_Discretor(pnt
                    (0, j*h));
60              else Neumann_Normal_Discretor(pnt(0, j*h), pnt(h, j
                    *h), pnt(2*h, j*h));
61
62          for (int i = 1; i <= n; ++ i)
63              if (cond[2] == 'D') Dirichlet_Normal_Discretor(pnt(
```

```
                i*h, 1));
64              else Neumann_Normal_Discretor(pnt(i*h, 1), pnt(i*h,
                     1-h), pnt(i*h, 1-2*h));
65
66          for (int j = 1; j <= n; ++ j)
67              if (cond[3] == 'D') Dirichlet_Normal_Discretor(pnt
                    (1, j*h));
68              else Neumann_Normal_Discretor(pnt(1, j*h), pnt(1-h,
                     j*h), pnt(1-2*h, j*h));
69
70          if (cond == "NNNN") coef[0][0] += 1;
71      }
72  public:
73      Normal_BVPSolver(int n, Function_2D<double>& f, Function_2D
            <double>& g, const string& s) :
74          n(n), d(0), h(1.0/(n+1)), f(f), g(g), cond(s) {}
75      void Solve() {
76          ID_Generator();
77          Matrix_Generator();
78          sol = Gauss_Improved_Solve(coef, rhs);
79      }
80      void Summary(Function_2D<double>& u, bool detail = 0) {
81          cout << "Domain : Normal" << endl;
82          cout << "Condition : " << cond << endl;
83          cout << "n = " << n << ", h = " << h << endl;
84          cout << "Values :" << endl;
85          double C = 0;
86          if (cond == "NNNN") {
87              for (int i = 0; i < d; ++ i) C += sol[i] - u(ps[i].
                    x, ps[i].y);
88              C /= d;
89          }
90          double res1 = 0, res2 = 0, resm = 0;
91          int cnt = 0;
92          for (int i = 0; i <= n+1; ++ i)
93              for (int j = 0; j <= n+1; ++ j) if (i!=0&&i!=n+1 ||
                     j!=0&&j!=n+1) {
94                  ++cnt;
95                  double uij = sol[id[pnt(i*h,j*h)]], uij_real =
```

```
                    u(i*h, j*h);
96                  double eij = fabs(uij - C - uij_real);
97                  res1 += eij;
98                  res2 += eij*eij;
99                  resm = max(resm, eij);
100                 cout << "(" << i*h << ", " << j*h << "), " << "
                        Solution Value : " << uij << ", Real Value
                        : " << uij_real << endl;
101             }
102         res1 /= cnt, res2 /= cnt, res2 = sqrt(res2);
103         cout << "Solution Error In L_1 : " << res1 << endl;
104         cout << "Solution Error In L_2 : " << res2 << endl;
105         cout << "Solution Error In L_max : " << resm << endl;
106     }
107 };
```

3.6 非规则区域 BVPSolver

按前面的设计思路，先将所有在圆外的网格点以及圆和网格的交点标号。

定义辅助函数 `void Laplace_Irnormal_Discretor(pnt p0, pnt p1, pnt p2, pnt p3, pnt p4)` 建立一个 Poisson 方程在一个内部点`p0`处的离散方程。`p1,p2,p3,p4`是它左右上下的四个相邻的网格点或边界离散点。

对于正方形边界上的离散点，直接调用规则区域 BVPSolver 定义的两个边界离散函数即可。

对于圆边界上的离散点，如果是 Dirichlet 边界条件，仍调用前面定义的边界离散函数即可。

再定义辅助函数 `void Neumann_Irnormal_Discretor(pnt p0, pnt p1, pnt p2, pnt p3, pnt p` 建立一个 Neumann 边界条件在圆上边界点`p0`处的离散方程。`p1,p2,p3,p4,p5`如前文对应章节所述。

```
1 #include <bits/stdc++.h>
2 #include "function.h"
3 #include "geo_2D.h"
4 #include "../Matrix.h"
5 #include "BVPSolver.h"
6 using namespace std;
```

```
7
8  class Irnormal_BVPSolver : public BVPSolver{
9  private:
10     int n, d;
11     circle D;
12     double h;
13     Function_2D<double> &f, &g;
14     Matrix<double> coef;
15     Colvec<double> rhs;
16     Colvec<double> sol;
17     map<pnt, int> id;
18     vector<pnt> ps;
19     string cond;
20     void ins(pnt p) {
21         if(!id.count(p)) id[p] = d++, ps.push_back(p);
22     }
23     void ID_Generator() {
24         for (int i = 1; i <= n; ++ i)
25             for (int j = 1; j <= n; ++ j)
26                 if (pnt_to_circle(pnt(i*h, j*h), D) >= 0)
27                     ins(pnt(i*h, j*h));
28         for (int i = 1; i <= n ;++ i) {
29             ins(pnt(0, i*h));
30             ins(pnt(1, i*h));
31             ins(pnt(i*h, 0));
32             ins(pnt(i*h, 1));
33         }
34         for (int i = 1; i <= n; ++ i)
35             if (sgn(i*h - (D.o.x-D.r)) * sgn(i*h - (D.o.x+D.r))
                    <= 0) {
36                 ins(pnt(i*h, D.Y1(i*h)));
37                 ins(pnt(i*h, D.Y2(i*h)));
38             }
39         for (int j = 1; j <= n; ++ j)
40             if (sgn(j*h - (D.o.y-D.r)) * sgn(j*h - (D.o.y+D.r))
                    <= 0) {
41                 ins(pnt(D.X1(j*h), j*h));
42                 ins(pnt(D.X2(j*h), j*h));
43             }
```

```
44      }
45      void Laplace_Irnormal_Discretor(pnt p0, pnt p1, pnt p2, pnt
            p3, pnt p4) {
46          int i0 = id[p0], i1 = id[p1], i2 = id[p2], i3 = id[p3],
                i4 = id[p4];
47          double lt = (p0.x - p1.x) / h, rt = (p2.x - p0.x) / h;
48          double dt = (p0.y - p3.y) / h, ut = (p4.y - p0.y) / h;
49          coef[i0][i0] += (lt+rt) / ((lt+rt)*lt*rt*h*h/2);
50          coef[i0][i1] -= rt / ((lt+rt)*lt*rt*h*h/2);
51          coef[i0][i2] -= lt / ((lt+rt)*lt*rt*h*h/2);
52          coef[i0][i0] += (dt+ut) / ((dt+ut)*dt*ut*h*h/2);
53          coef[i0][i3] -= ut / ((dt+ut)*dt*ut*h*h/2);
54          coef[i0][i4] -= dt / ((dt+ut)*dt*ut*h*h/2);
55          rhs[i0] = f(p0.x, p0.y);
56      }
57      void Dirichlet_Discretor(pnt p0) {
58          int i0 = id[p0];
59          coef[i0][i0] = 1;
60          rhs[i0] = g(p0.x, p0.y);
61      }
62      void Neumann_Normal_Discretor(pnt p0, pnt p1, pnt p2) {
63          int i0 = id[p0], i1 = id[p1], i2 = id[p2];
64          coef[i0][i0] -= 1.5 / h;
65          coef[i0][i1] += 2 / h;
66          coef[i0][i2] -= 0.5 / h;
67          rhs[i0] = g(p0.x, p0.y);
68      }
69      void Neumann_Irnormal_Discretor(pnt p0, pnt p1, pnt p2, pnt
            p3, pnt p4, pnt p5) {
70          int i0 = id[p0], i1 = id[p1], i2 = id[p2], i3 = id[p3],
                i4 = id[p4], i5 = id[p5];
71          Matrix<double> c(6, 6);
72          Colvec<double> b(6);
73          pnt p[6] = {p0, p1, p2, p3, p4, p5};
74          for (int i = 0; i < 6; ++ i) {
75              c[0][i] = 1;
76              c[1][i] = p[i].x-p0.x;
77              c[2][i] = p[i].y-p0.y;
78              c[3][i] = (p[i].x-p0.x) * (p[i].x-p0.x) / 2;
```

```
79              c[4][i] = (p[i].y-p0.y) * (p[i].y-p0.y) / 2;
80              c[5][i] = (p[i].x-p0.x) * (p[i].y-p0.y);
81          }
82          double arg = D.arg(p0);
83          b[1] = cos(arg);
84          b[2] = sin(arg);
85          Colvec<double> w = Gauss_Improved_Solve(c, b);
86          coef[i0][i0] = w[0];
87          coef[i0][i1] = w[1];
88          coef[i0][i2] = w[2];
89          coef[i0][i3] = w[3];
90          coef[i0][i4] = w[4];
91          coef[i0][i5] = w[5];
92          rhs[i0] = g(p0.x, p0.y);
93      }
94      void Matrix_Generator() {
95          coef = Matrix<double> (d, d);
96          rhs = Colvec<double> (d);
97          for (int i = 1; i <= n; ++ i)
98              for (int j = 1; j <= n; ++ j) if (pnt_to_circle(pnt
                    (i*h, j*h), D) > 0){
99                  pnt p(i*h, j*h);
100                 pnt lp((i-1)*h, j*h);
101                 pnt rp((i+1)*h, j*h);
102                 pnt dp(i*h, (j-1)*h);
103                 pnt up(i*h, (j+1)*h);
104                 if (pnt_to_circle(lp, D) < 0) lp.x = D.X2(j*h);
105                 if (pnt_to_circle(rp, D) < 0) rp.x = D.X1(j*h);
106                 if (pnt_to_circle(dp, D) < 0) dp.y = D.Y2(i*h);
107                 if (pnt_to_circle(up, D) < 0) up.y = D.Y1(i*h);
108                 Laplace_Irnormal_Discretor(p, lp, rp, dp, up);
109             }
110
111         for (int i = 1; i <= n; ++ i)
112             if (cond[0] == 'D') Dirichlet_Discretor(pnt(i*h, 0)
                    );
113             else Neumann_Normal_Discretor(pnt(i*h, 0), pnt(i*h,
                     h), pnt(i*h, 2*h));
114
```

```
115         for (int j = 1; j <= n; ++ j)
116             if (cond[1] == 'D') Dirichlet_Discretor(pnt(0, j*h)
                    );
117             else Neumann_Normal_Discretor(pnt(0, j*h), pnt(h, j
                    *h), pnt(2*h, j*h));
118
119         for (int i = 1; i <= n; ++ i)
120             if (cond[2] == 'D') Dirichlet_Discretor(pnt(i*h, 1)
                    );
121             else Neumann_Normal_Discretor(pnt(i*h, 1), pnt(i*h,
                     1-h), pnt(i*h, 1-2*h));
122
123         for (int j = 1; j <= n; ++ j)
124             if (cond[3] == 'D') Dirichlet_Discretor(pnt(1, j*h)
                    );
125             else Neumann_Normal_Discretor(pnt(1, j*h), pnt(1-h,
                     j*h), pnt(1-2*h, j*h));
126
127         if (cond[4] == 'D') {
128             for (int i = 1; i <= n; ++ i)
129                 if (sgn(i*h - (D.o.x-D.r)) * sgn(i*h - (D.o.x+D
                        .r)) == -1) {
130                     Dirichlet_Discretor(pnt(i*h, D.Y1(i*h)));
131                     Dirichlet_Discretor(pnt(i*h, D.Y2(i*h)));
132                 }
133             for (int j = 1; j <= n; ++ j)
134                 if (sgn(j*h - (D.o.y-D.r)) * sgn(j*h - (D.o.y+D
                        .r)) == -1) {
135                     Dirichlet_Discretor(pnt(D.X1(j*h), j*h));
136                     Dirichlet_Discretor(pnt(D.X2(j*h), j*h));
137                 }
138         }
139         else {
140             for (int i = 1, j; i <= n; ++ i)
141                 if (sgn(i*h - (D.o.x-D.r)) * sgn(i*h - (D.o.x+D
                        .r)) <= 0) {
142                     pnt p(i*h, D.Y1(i*h));
143                     int j = int(p.y/h-1e-12);
144                     int op = sgn(i*h-D.o.x);
```

```
145                     if(op == 0) op = 1;
146                     pnt p1(i*h, j*h);
147                     pnt p2(i*h, (j-1)*h);
148                     pnt p3((i+op)*h, j*h);
149                     pnt p4((i+op)*h, (j-1)*h);
150                     pnt p5((i+2*op)*h, j*h);
151                     Neumann_Irnormal_Discretor(p, p1, p2, p3,
                            p4, p5);
152                 }
153             for (int i = 1; i <= n; ++ i)
154                 if (sgn(i*h - (D.o.x-D.r)) * sgn(i*h - (D.o.x+D
                        .r)) == -1) {
155                     pnt p(i*h, D.Y2(i*h));
156                     int j = int(p.y/h+1e-12) + 1;
157                     int op = sgn(i*h-D.o.x);
158                     if (op == 0) op = 1;
159                     pnt p1(i*h, j*h);
160                     pnt p2(i*h, (j+1)*h);
161                     pnt p3((i+op)*h, j*h);
162                     pnt p4((i+op)*h, (j+1)*h);
163                     pnt p5((i+2*op)*h, j*h);
164                     Neumann_Irnormal_Discretor(p, p1, p2, p3,
                            p4, p5);
165                 }
166             for (int j = 1; j <= n; ++ j)
167                 if (sgn(j*h - (D.o.y-D.r)) * sgn(j*h - (D.o.y+D
                        .r)) <= 0) {
168                     pnt p(D.X1(j*h), j*h);
169                     int i = int(p.x/h-1e-12);
170                     int op = sgn(j*h-D.o.y);
171                     if (op == 0) op = 1;
172                     pnt p1(i*h, j*h);
173                     pnt p2((i-1)*h, j*h);
174                     pnt p3(i*h, (j+op)*h);
175                     pnt p4((i-1)*h, (j+op)*h);
176                     pnt p5(i*h, (j+2*op)*h);
177                     Neumann_Irnormal_Discretor(p, p1, p2, p3,
                            p4, p5);
178                 }
```

```
179             for (int j = 1; j <= n; ++ j)
180                 if (sgn(j*h - (D.o.y-D.r)) * sgn(j*h - (D.o.y+D
                        .r)) == -1) {
181                     pnt p(D.X2(j*h), j*h);
182                     int i = int(p.x/h+1e-12) + 1;
183                     int op = sgn(j*h-D.o.y);
184                     if (op == 0) op = 1;
185                     pnt p1(i*h, j*h);
186                     pnt p2((i+1)*h, j*h);
187                     pnt p3(i*h, (j+op)*h);
188                     pnt p4((i+1)*h, (j+op)*h);
189                     pnt p5(i*h, (j+2*op)*h);
190                     Neumann_Irnormal_Discretor(p, p1, p2, p3,
                            p4, p5);
191                 }
192         }
193         if (cond == "NNNNN") coef[0][0] += 1;
194     }
195 public:
196     Irnormal_BVPSolver(int n, Function_2D<double>& f,
            Function_2D<double>& g, double x0, double y0, double r,
             const string& s) :
197         n(n), d(0), ps(0), h(1.0/(n+1)), f(f), g(g), D(x0, y0,
                r), cond(s) {
198             if (r <= h) {
199                 cerr << "Too␣Coarse␣Grid!" << endl;
200                 throw 1;
201             }
202             if (x0 - r <= 2*h || x0 + r >= 1-2*h || y0 - r <=
                    2*h || y0 + r >= 1-2*h) {
203                 cerr << "Circle␣Outside␣the␣Square!" << endl;
204                 throw 1;
205             }
206         }
207     void Solve() {
208         ID_Generator();
209         Matrix_Generator();
210         sol = Gauss_Improved_Solve(coef, rhs);
211     }
```

```
212     void Summary(Function_2D<double>& u, bool detail = 0) {
213         cout << "Domain : Irnormal" << endl;
214         cout << "Condition : " << cond << endl;
215         cout << "n = " << n << ", h = " << h << endl;
216         cout << "Circle : " << "O = (" << D.o.x << ", " << D.o.
                y << "), r = " << D.r << endl;
217         if (detail) cout << "Values :" << endl;
218         double C = 0;
219         for (int i = 0; i < d; ++ i) C += sol[i] - u(ps[i].x,
                ps[i].y);
220         C /= d;
221         double res1 = 0, res2 = 0, resm = 0;
222         int cnt = 0;
223         for (int i = 1; i <= n; ++ i)
224             for (int j = 1; j <= n; ++ j) if (pnt_to_circle(pnt
                    (i*h, j*h), D) > 0) {
225                 ++ cnt;
226                 double uij = sol[id[pnt(i*h,j*h)]], uij_real =
                        u(i*h, j*h);
227                 double eij = fabs(uij - C - uij_real);
228                 res1 += eij;
229                 res2 += eij*eij;
230                 resm = max(resm, eij);
231                 if (detail) cout << "(" << i*h << ", " << j*h
                        << "), " << "Solution Value : " << uij << "
                        , Real Value : " << uij_real << endl;
232             }
233         res1 /= cnt, res2 /= cnt, res2 = sqrt(res2);
234         cout << "Solution Error In L_1 : " << res1 << endl;
235         cout << "Solution Error In L_2 : " << res2 << endl;
236         cout << "Solution Error In L_max : " << resm << endl;
237     }
238 };
```

3.7 测试程序

使用如下三个函数测试：

$$u(x,y) = \exp\{y + \sin x\} \tag{19}$$

$$u(x,y) = \exp\{-(x^2 + y^2)\} \tag{20}$$

$$u(x,y) = \frac{1}{\sqrt{x^2 + y^2}} \tag{21}$$

根据拉普拉斯算子和法向导数的定义，定义函数 f 和 g。

```
#include<bits/stdc++.h>
#include "function.h"
#include "geo_2D.h"
#include "Normal_BVPSolver.h"
#include "Irnormal_BVPSolver.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 V : public Function_2D<double> {
public:
    double operator ()(const double& x, const double& y)
        const {
        return exp(-(x*x+y*y));
    }
```

```
26          double partial(const double& x, const double& y, const
                int& i, const int& j) const {
27              if (i == 1 && j == 0) return -2 * x * exp(-(x*x+y*y
                    ));
28              if (i == 2 && j == 0) return (4*x*x - 2) * exp(-(x*
                    x+y*y));
29              if (i == 0 && j == 1) return -2 * x * exp(-(x*x+y*y
                    ));
30              if (i == 0 && j == 2) return (4*y*y - 2) * exp(-(x*
                    x+y*y));
31              throw 0;
32          }
33      } v;
34
35      class W : public Function_2D<double> {
36      public:
37          double operator ()(const double& x, const double& y)
                const {
38              return 1/sqrt(x*x+y*y);
39          }
40          double partial(const double& x, const double& y, const
                int& i, const int& j) const {
41              if (i == 1 && j == 0) return -x / (x*x+y*y) / sqrt(
                    x*x+y*y);
42              if (i == 2 && j == 0) return (2*x*x-y*y) / (x*x+y*y
                    ) / (x*x+y*y) / sqrt(x*x+y*y);
43              if (i == 0 && j == 1) return -y / (x*x+y*y) / sqrt(
                    x*x+y*y);
44              if (i == 0 && j == 2) return (2*y*y-x*x) / (x*x+y*y
                    ) / (x*x+y*y) / sqrt(x*x+y*y);
45              throw 0;
46          }
47      } w;
48
49      class F : public Function_2D<double> {
50      public:
51          Function_2D<double>& u;
52          F(Function_2D<double>& u) : u(u) {}
53          double operator ()(const double& x, const double& y)
```

```
            const {
54          return -(u.partial(x, y, 2, 0) + u.partial(x, y, 0,
                2));
55      }
56  };
57
58  class G : public Function_2D<double> {
59  public:
60      Function_2D<double>& u;
61      string cond;
62      G(Function_2D<double>& u, const string& s) : u(u), cond
            (s) {}
63      double operator ()(const double& x, const double& y)
            const {
64          if (x == 0) return cond[0] == 'D' ? u(0, y) : u.
                partial(0, y, 1, 0);
65          if (x == 1) return cond[1] == 'D' ? u(1, y) : -u.
                partial(1, y, 1, 0);
66          if (y == 0) return cond[2] == 'D' ? u(x, 0) : u.
                partial(x, 0, 0, 1);
67          if (y == 1) return cond[3] == 'D' ? u(x, 1) : -u.
                partial(x, 1, 0, 1);
68          throw 0;
69      }
70  };
71
72  class G1 : public Function_2D<double> {
73  public:
74      Function_2D<double>& u;
75      circle D;
76      string cond;
77      G1(Function_2D<double>& u, double x0, double y0, double
             r, const string& s) : u(u), D(x0, y0, r), cond(s)
            {}
78      double operator ()(const double& x, const double& y)
            const {
79          if (x == 0) return cond[0] == 'D' ? u(0, y) : u.
                partial(0, y, 1, 0);
80          if (x == 1) return cond[1] == 'D' ? u(1, y) : -u.
```

```
                partial(1, y, 1, 0);
81              if (y == 0) return cond[2] == 'D' ? u(x, 0) : u.
                    partial(x, 0, 0, 1);
82              if (y == 1) return cond[3] == 'D' ? u(x, 1) : -u.
                    partial(x, 1, 0, 1);
83              if (pnt_to_circle(pnt(x,y), D) == 0) {
84                  if (cond[4] == 'D') return u(x, y);
85                  double arg = D.arg(pnt(x,y));
86                  return cos(arg) * u.partial(x, y, 1, 0) + sin(
                        arg) * u.partial(x, y, 0, 1);
87              }
88              throw 0;
89          }
90      };
91
92      int main(int argc, char** argv){
93          string cmd0 = argv[1], cmd1 = argv[2], cmd2 = argv[3];
94          Function_2D<double>& t = u;
95          if (cmd0 == "v") t = v;
96          if (cmd0 == "w") t = w;
97          int n = atoi(argv[4]);
98          F f(t);
99          if (cmd1 == "Normal") {
100             if (cmd2 == "Dirichlet") cmd2 = "DDDD";
101             if (cmd2 == "Neumann") cmd2 = "NNNN";
102             G g(t, cmd2);
103             Normal_BVPSolver M(n, f, g, cmd2);
104             M.Solve();
105             M.Summary(t);
106         }
107         else {
108             double x0 = atof(argv[5]);
109             double y0 = atof(argv[6]);
110             double r = atof(argv[7]);
111             if (cmd2 == "Dirichlet") cmd2 = "DDDDD";
112             if (cmd2 == "Neumann") cmd2 = "NNNNN";
113             G1 g(t, x0, y0, r, cmd2);
114             Irnormal_BVPSolver MI(n, f, g, x0, y0, r, cmd2);
115             MI.Solve();
```

```
116                 MI.Summary(t);
117             }
118         }
```

4 测试数据

对三个函数、两种区域、三种边界条件、$n = 8, 16, 32, 64$ 四种网格密度分别测试。

测试结果见 `text1.txt,text2.txt,text3.txt` 三个文档，每个文档存一个函数的测试结果。

执行 `make run` 即可得到全部结果。

通过对同种区域同种边界条件不同网格密度的比较可知，对于 Dirichlet 和 Mixed 边界条件，无论是规则区域还是非规则区域，n 每增大一倍，误差都降为原来的 $\frac{1}{4}$。即误差是二阶收敛的，比较符合理论。

但对于纯 Neumann 边界条件，n 每增大一倍，误差仅减小为原来的 $\frac{1}{3}$。即误差仅 1.6 阶收敛。这是不符合理论的。可能是因为 Neumann 边界条件得到的方程组接近奇异，导致解方程组本身的误差随着 n 的增大而增大。