第三章程序作业

彭榆烜

2024 年 11 月 4 日

目录

1 问题分析 **3**
1.1 问题描述 . . . 3
1.2 问题处理 . . . 3

2 不同算法实现 **5**
2.1 Jacobi 方法 . . . 5
2.1.1 算法原理 . . . 5
2.1.2 计算结果 . . . 6
2.2 Gauss-Seidel 法 . . . 6
2.2.1 算法原理 . . . 6
2.2.2 计算结果 . . . 8
2.3 SOR 法 . . . 8
2.3.1 算法原理 . . . 8
2.3.2 计算结果 . . . 10
2.4 SSOR 法 . . . 10
2.4.1 算法原理 . . . 10
2.4.2 计算结果 . . . 12
2.5 块 Jacobi 方法 . . . 12
2.5.1 算法原理 . . . 12
2.5.2 计算结果 . . . 14
2.6 块 Gauss-Seidel 方法 . . . 14
2.6.1 算法原理 . . . 14
2.6.2 计算结果 . . . 16
2.7 块 SOR 方法 . . . 16
2.7.1 算法原理 . . . 16
2.7.2 计算结果 . . . 18
2.8 块 SSOR 方法 . . . 18
2.8.1 算法原理 . . . 18
2.8.2 计算结果 . . . 20
2.9 最速下降法 . . . 20
2.9.1 算法原理 . . . 20
2.9.2 计算结果 . . . 23

2.10 共轭梯度法 23
2.10.1 算法原理 23
2.10.2 计算结果 26

3 不同算法对比 **27**

4 代码 **28**

1 问题分析

1.1 问题描述

数值求解正方形域上的泊松（Poisson）方程边值问题。问题对应的数学模型如下 [1]：

$$
\begin{cases}
-(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) = f(x,y), 0 < x, y < 1 & (1) \\
u(0,y) = u(1,y) = u(x,0) = u(x,1) = 0 & (2)
\end{cases}
$$

其中式 (2) 为边界条件。

1.2 问题处理

对问题的正方形求解域进行空间离散，用平行坐标轴的直线,

$$
\begin{gathered}
x = x_i = h, y = y_j = jh, h = \tfrac{1}{N+1} \\
i, j = 0, 1, \cdots, N, N+1
\end{gathered} \tag{3}
$$

将求解域划分为网格，如下图所示。

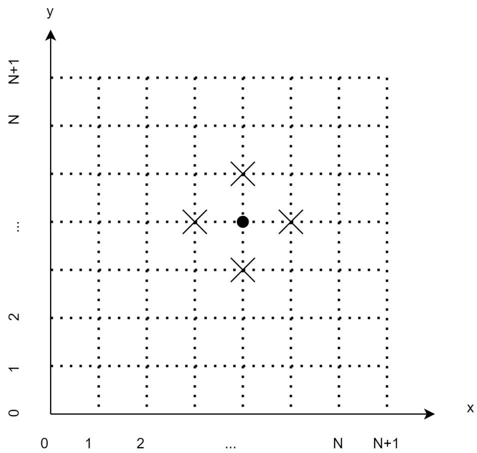


图 1: 离散坐标

对泊松方程的的微分部分进行离散

$$
\frac{\partial^2 u}{\partial x^2}(x_i, y_j) = \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} + O(h^2) \tag{4}
$$

$$
\frac{\partial^2 u}{\partial y^2}(x_i, y_j) = \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2} + O(h^2) \tag{5}
$$

得到每个点的有限差分方程：

$$
4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{ij} \ (1 \le i, j \le N) \tag{6}
$$

在边界处有：

$$
u_{0,j} = u_{N+1,j} = u_{i,0} = u_{i,N+1} = 0 \ (1 \le i, j \le N) \tag{7}
$$

对非边界点进行编号，顺序为对直线 $y = y_i$; 从下往上，对固定的一条线上的网点从左往右依次编号，对应的 u 和 f 也依次编号。最后得差分方程组 $(1 \leq i, j \leq N)$:

$$\begin{cases} 4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{ij} & (8) \\ u_{0,j} = u_{N+1,j} = u_{i,0} = u_{i,N+1} = 0 & (9) \end{cases}$$

2 不同算法实现

2.1 Jacobi 方法

2.1.1 算法原理

由上述问题分析，原问题转化成求解 $Au = f$ 的方程，对于此方程，可以构造 Jacobi 迭代的分量形式为：

$$x_i^{(k+1)} = (b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)})/a_{ii}, i = 1, 2, \cdots, n \tag{10}$$

对应的点更新的 Jacobi 迭代公式为：

$$u_{i,j}^{(k+1)} = (u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)} + h^2 f_{ij})/4 \tag{11}$$

代码实现至少有两种思路，第一种是先生成一个三维数组，第二种是在迭代循环内部先存储先前迭代步数对应的解，用于当前迭代步数的求解，两者在相同条件下的迭代次数是相同的。这里以第二种为例，式 (11) 对应的 Python 代码如下：

Listing 1: Jacobi

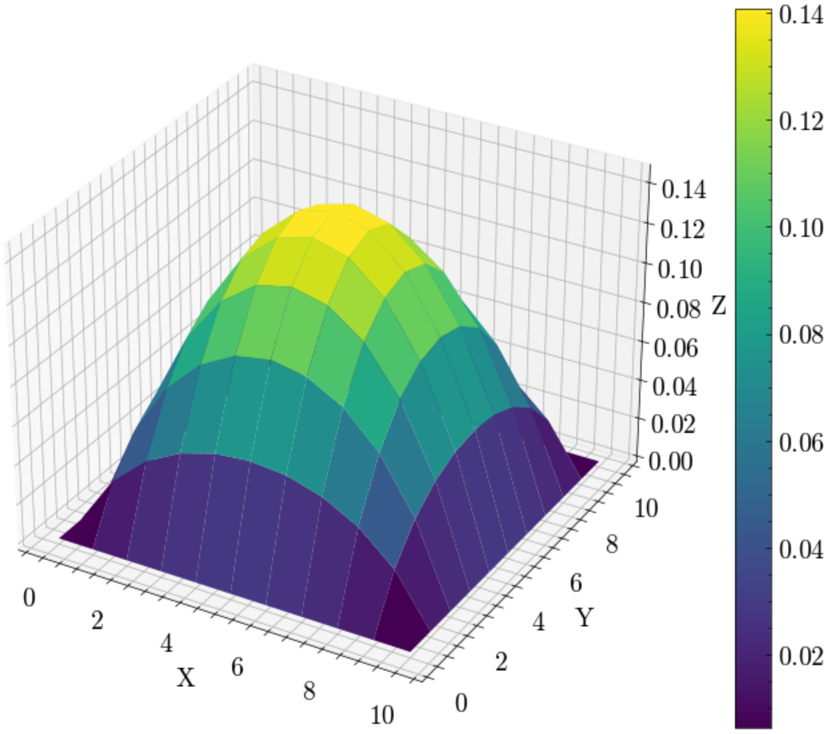
```
1  def J(n=9, max_iter=1000, tol=1e-5):
2      u = np.zeros([n + 2, n + 2])
3      h = 1 / (n + 1)
4      f = np.full([n + 2, n + 2], h**2 * 2)
5      for k in range(max_iter):
6          e = 0.0
7          uo = u.copy() # 存储上一个迭代步数的解
8          for j in range(1, n + 1):
9              for i in range(1, n + 1):
10                 uol = u[i, j].copy()
11                 u[i, j] = (
12                     uo[i - 1, j] + u[i + 1, j] + uo[i, j - 1] + u[i, j + 1] + f[i, j]
13                 ) / 4
14                 e = e + np.abs(u[i, j] - uol)
15         if e / n**2 < tol:
16             break
17     return u, k + 1
```

其中，k 为迭代次数，代码中 14 行的 $f[i,j]$ 为二维数组（矩阵）中的一个分量，对应于式 (11) 中的 $h^2 f_{ij}$。

2.1.2 计算结果

表 1: Algorithm: J

Algorithm	**Iterations**	**Time (s)**
J	121	0.030681




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02559032 0.04118415 0.05070034 0.05588843 0.05754048 0.05588843 0.05070034 0.04118415 0.02559032 0.        ]
 [0.         0.04118415 0.0684593  0.08574715 0.09533452 0.09840778 0.09533452 0.08574715 0.0684593  0.04118415 0.        ]
 [0.         0.05070034 0.08574715 0.10852945 0.12133581 0.12546486 0.12133581 0.10852945 0.08574715 0.05070034 0.        ]
 [0.         0.05588843 0.09533452 0.12133581 0.13607105 0.14083953 0.13607105 0.12133581 0.09533452 0.05588843 0.        ]
 [0.         0.05754048 0.09840778 0.12546486 0.14083953 0.14582115 0.14083953 0.12546486 0.09840778 0.05754048 0.        ]
 [0.         0.05588843 0.09533452 0.12133581 0.13607105 0.14083953 0.13607105 0.12133581 0.09533452 0.05588843 0.        ]
 [0.         0.05070034 0.08574715 0.10852945 0.12133581 0.12546486 0.12133581 0.10852945 0.08574715 0.05070034 0.        ]
 [0.         0.04118415 0.0684593  0.08574715 0.09533452 0.09840778 0.09533452 0.08574715 0.0684593  0.04118415 0.        ]
 [0.         0.02559032 0.04118415 0.05070034 0.05588843 0.05754048 0.05588843 0.05070034 0.04118415 0.02559032 0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 2: Jacobi 方法计算结果

2.2 Gauss-Seidel 法

2.2.1 算法原理

Gauss-Seidel 方法在前面 Jacobi 方法的基础上做了修改，在本次迭代的环境下，使用本次迭代计算得到的向前的解进行计算，而不是上次迭代对应的解，写成点更新的迭代格式：

$$u_{i,j}^{(k+1)} = (u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k+1)} + u_{i,j+1}^{(k)} + h^2 f_{ij})/4 \tag{12}$$

由于 $u_{i-1,j}^{(k+1)}$ 和 $u_{i,j-1}^{(k+1)}$ 为当前迭代下的前值（已经计算出来的值，已更新），而 $u_{i+1,j}^{(k)}$ 和 $u_{i,j+1}^{(k)}$ 是上一迭代下的值，但未更新，所以在代码中这四个量都是当前迭代下的表达（即表达式相同），对应的代码如下：

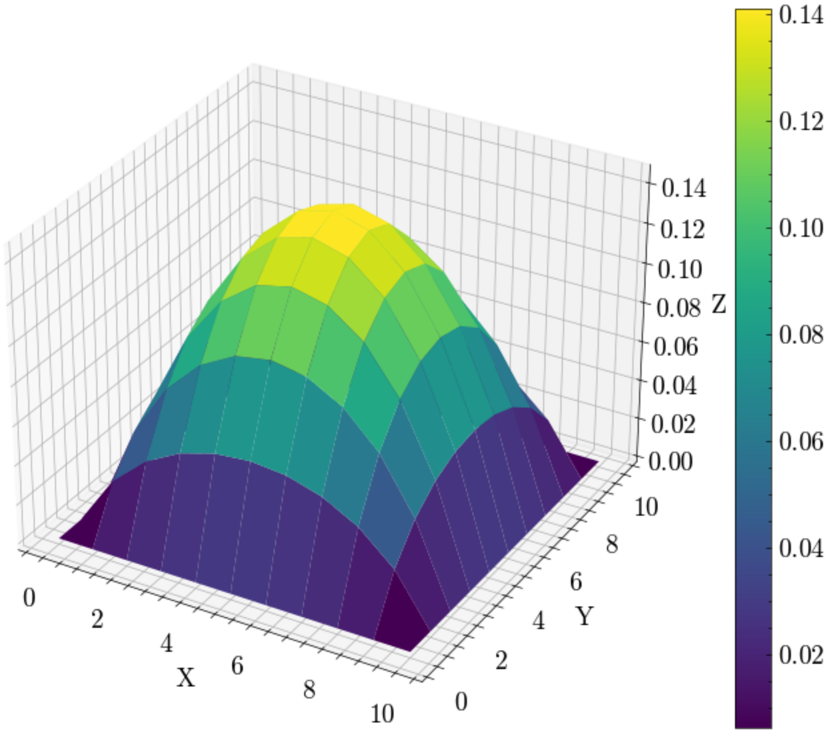
Listing 2: Gauss-Seidel

```
def GS(n=9, max_iter=1000, tol=1e-5):
    u = np.zeros([n + 2, n + 2])
    h = 1 / (n + 1)
    f = np.full([n + 2, n + 2], h**2 * 2)
    for k in range(max_iter):
        e = 0.0
        for j in range(1, n + 1):
            for i in range(1, n + 1):
                uo = u[i, j].copy()
                u[i, j] = (
                    u[i - 1, j] + u[i + 1, j] + u[i, j - 1] + u[i, j + 1] + f[i, j]
                ) / 4
                e = e + np.abs(u[i, j] - uo)
        if e / n**2 < tol:
            break
    return u, k + 1
```

2.2.2 计算结果

表 2: Algorithm: GS

Algorithm	Iterations	Time (s)
GS	68	0.017398




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02560001 0.04120503 0.05073226 0.05592953 0.05758726 0.05593615 0.05074354 0.04121734 0.02560867 0.        ]
 [0.         0.04120503 0.06850342 0.08581366 0.09541915 0.09850322 0.09543112 0.08583406 0.0685257  0.04122069 0.        ]
 [0.         0.05073226 0.08581366 0.10862854 0.12146075 0.12560467 0.12147643 0.10865524 0.08584282 0.05075276 0.        ]
 [0.         0.05592953 0.09541915 0.12146075 0.13622736 0.14101334 0.13624488 0.12149061 0.09545175 0.05595245 0.        ]
 [0.         0.05758726 0.09850322 0.12560467 0.14101334 0.14601333 0.14103086 0.12563453 0.09853583 0.05761018 0.        ]
 [0.         0.05593615 0.09543112 0.12147643 0.13624488 0.14103086 0.13626073 0.12150343 0.09546061 0.05595688 0.        ]
 [0.         0.05074354 0.08583406 0.10865524 0.12149061 0.12563453 0.12150343 0.10867709 0.08585791 0.05076031 0.        ]
 [0.         0.04121734 0.0685257  0.08584282 0.09545175 0.09853583 0.09546061 0.08585791 0.06854218 0.04122893 0.        ]
 [0.         0.02560867 0.04122069 0.05075276 0.05595245 0.05761018 0.05595688 0.05076031 0.04122893 0.02561447 0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 3: Gauss-Seidel 方法计算结果

2.3 SOR 法

2.3.1 算法原理

Gauss-Seidel 迭代法虽然计算简单，但是在实际计算中，因为其迭代矩阵的谱半径常接近 1，导致收敛很慢。为了克服这个缺点，引进一个加速因子 ω（又称松弛因子）对 Guss-Seidel 方法进行修正加速。这种方法最早是 1950 年由 Young 和 Frankel 针对偏微分方程数值解中离散的线性方程组提出来的，称为逐次超松弛迭代法（Successive Over Relaxation Method），简称 SOR 方法。

代码实现至少有种思路，一种是先通过 Gauss-Seidel 方法计算出 $\bar{x}_i^{(k+1)}$，然后用松弛因子 ω 对 $\bar{x}_i^{(k+1)}$ 和 $x_i^{(k)}$ 进行线性组合；另一种是通过分量形式对 x 进行一次更新。下面展示第二种思路：

n 阶方程的 SOR 方法的分量形式为：

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{a_i}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i}^{n} a_{ij}x_j^{(k)}) \ (i = 1, 2, \cdots, n) \tag{13}$$

转换成对应的点更新的 SOR 迭代格式为:

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \frac{\omega}{4}\left[f_{i,j} - \left((-1)u_{i-1,j}^{(k+1)} + (-1)u_{i-1,j}^{(k+1)}\right) - \left(4u_{i,j}^{(k)} + (-1)u_{i+1,j}^{(k)} + (-1)u_{i,j+1}^{(k)}\right)\right] \tag{14}$$

移项化简得:

$$u_{i,j}^{(k+1)} = \frac{\omega}{4}\left[f_{i,j} + u_{i-1,j}^{(k+1)} + u_{i,j-1}^{(k+1)} + \left(\frac{4}{\omega} - 4\right)u_{i,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right] \tag{15}$$

通过遍历的方式，取得迭代次数最小值对应的 ω 值为 1.55。对应的代码如下:

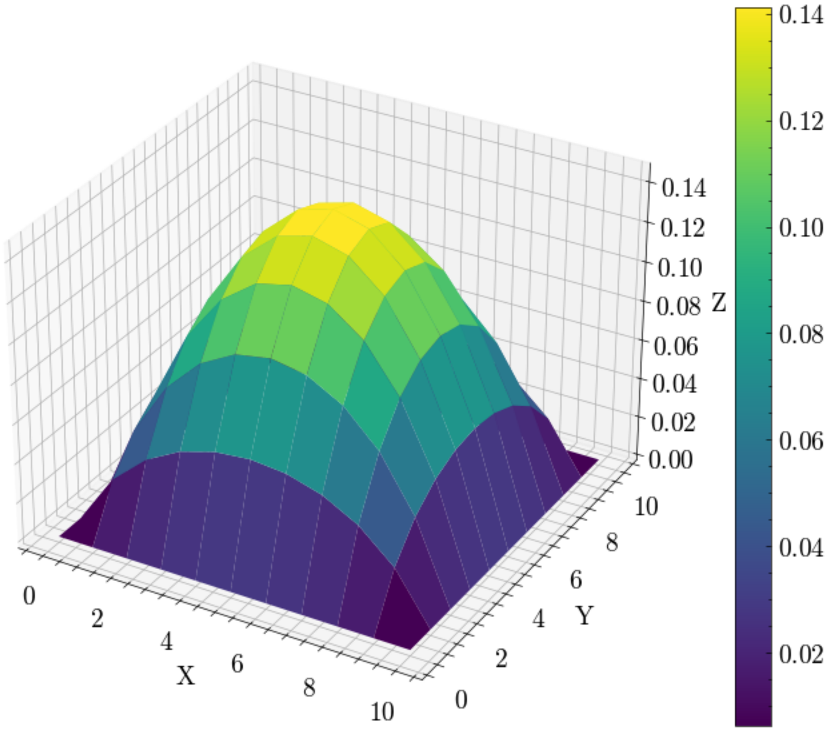
Listing 3: SOR

```
def SOR(n=9, max_iter=1000, tol=1e-5, w=1.55):
    u = np.zeros([n + 2, n + 2])
    h = 1 / (n + 1)
    f = np.full([n + 2, n + 2], h**2 * 2)
    for k in range(max_iter):
        e = 0.0
        for j in range(1, n + 1):
            for i in range(1, n + 1):
                uo = u[i, j].copy()
                u[i, j] = (
                    (
                        (4 / w - 4) * u[i, j]
                        + u[i - 1, j]
                        + u[i + 1, j]
                        + u[i, j - 1]
                        + u[i, j + 1]
                        + f[i, j]
                    )
                    * w
                    / 4
                )
                e = e + np.abs(u[i, j] - uo)
        if e / n**2 < tol:
            break
    return u, k + 1
```

2.3.2 计算结果

表 3: Algorithm: SOR

Algorithm	Iterations	Time (s)
SOR	17	0.004839




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02561227 0.04122925 0.05076644 0.05597027 0.05763033 0.05597704 0.05077806 0.0412421  0.02562147 0.        ]
 [0.         0.04122925 0.06855065 0.08587958 0.09549706 0.098585   0.09550831 0.08589889 0.06857201 0.04124454 0.        ]
 [0.         0.05076644 0.08587958 0.10871976 0.1215678  0.12571637 0.12158133 0.10874298 0.08590527 0.05078483 0.        ]
 [0.         0.05597027 0.09549706 0.1215678  0.13635224 0.141143   0.13636613 0.12159165 0.09552344 0.05598915 0.        ]
 [0.         0.05763033 0.098585   0.12571637 0.141143   0.14614738 0.14115575 0.12573828 0.09860924 0.05764768 0.        ]
 [0.         0.05597704 0.09550831 0.12158133 0.13636613 0.14115575 0.13637673 0.12159952 0.09552844 0.05599145 0.        ]
 [0.         0.05077806 0.08589889 0.10874298 0.12159165 0.12573828 0.12159952 0.1087565  0.08591385 0.05078877 0.        ]
 [0.         0.0412421  0.06857201 0.08590527 0.09552344 0.09860924 0.09552844 0.08591385 0.06858151 0.0412489  0.        ]
 [0.         0.02562147 0.04124454 0.05078483 0.05598915 0.05764768 0.05599145 0.05078877 0.0412489  0.02562459 0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 4: SOR 方法计算结果

2.4 SSOR 法

2.4.1 算法原理

对称超松弛迭代法（SSOR 法）的每一步迭代由向前迭代的 SOR 方法和向后迭代的 SOR 方法两步组成，并引入 $x^{(k+\frac{1}{2})}$ 表示第 k 次迭代后的一个中间变量。

先使用向前迭代的 SOR 方法，计算 $x_i^{(k+\frac{1}{2})}$：

$$x_i^{(k+\frac{1}{2})} = x_i^{(k)} + \frac{\omega}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+\frac{1}{2})} - \sum_{j=i}^{n} a_{ij}x_j^{(k)})\ (i = 1, 2, \cdots, n) \tag{16}$$

然后使用向后迭代的 SOR 方法，计算 $x_i^{(k+1)}$：

$$x_i^{(k+1)} = x_i^{(k+\frac{1}{2})} + \frac{\omega}{a_{ii}}(b_i - \sum_{j=1}^{i} a_{ij}x_j^{(k+\frac{1}{2})} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k+1)})\ (i = n, n-1, \cdots, 2, 1) \tag{17}$$

代码中，向前迭代的 SOR 算法与原 SOR 算法的代码相似，只需要新建一个二维数组 um（数组形状与 u 相同）来存储中间值；对于向后迭代的 SOR 算法部分，建立一个新的从后往前的循环计算 $u[i, j]$。通过遍历的方式，取得迭代次数最小值对应的 ω 值为 1.55，具体代码如下：

Listing 4: SSOR

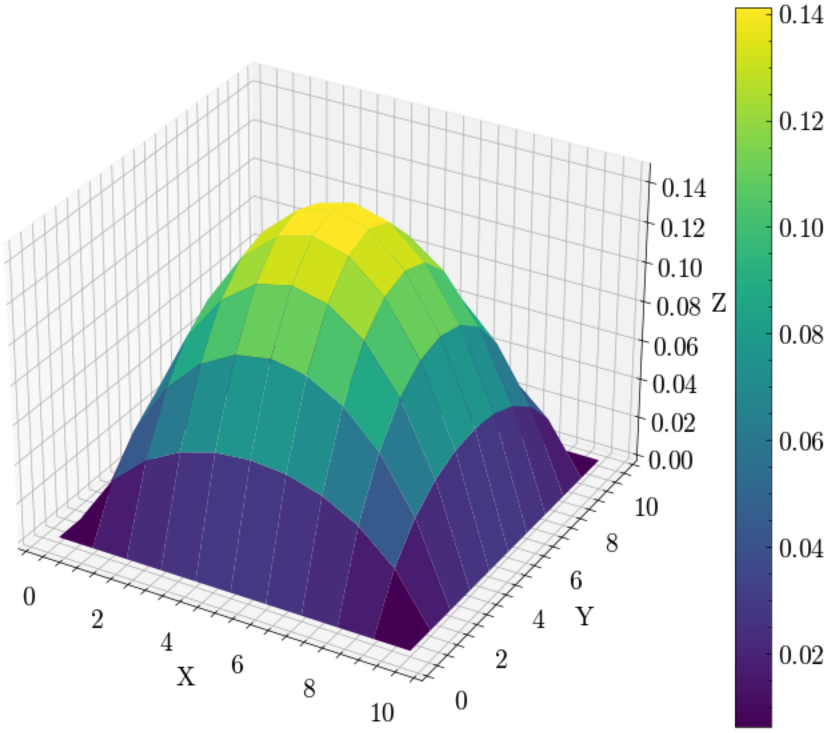
```
def SSOR(n=9, max_iter=1000, tol=1e-5, w=1.55):
    u = np.zeros([n + 2, n + 2])
    um = np.zeros([n + 2, n + 2])
    h = 1 / (n + 1)
    f = np.full([n + 2, n + 2], h**2 * 2)
    for k in range(max_iter):
        e1 = 0.0
        e2 = 0.0
        for j in range(1, n + 1):
            for i in range(1, n + 1):
                uo = um[i, j].copy()
                um[i, j] = (
                    (
                        (4 / w - 4) * um[i, j]
                        + um[i - 1, j]
                        + um[i + 1, j]
                        + um[i, j - 1]
                        + um[i, j + 1]
                        + f[i, j]
                    )
                    * w
                    / 4
                )
                e1 = e1 + np.abs(um[i, j] - uo)
        for j in range(n, 0, -1):
            for i in range(n, 0, -1):
                uo1 = um[i, j].copy()
                u[i, j] = (
                    (
                        (4 / w - 4) * um[i, j]
                        + um[i - 1, j]
                        + u[i + 1, j]
                        + um[i, j - 1]
                        + u[i, j + 1]
                        + f[i, j]
                    )
                    * w
                    / 4
                )
                e2 = e2 + np.abs(u[i, j] - uo1)
        if (e1 + e2) / n**2 < tol:
```

```python
            break
    return u, k + 1
```

2.4.2 计算结果

表 4: Algorithm: SSOR

Algorithm	Iterations	Time (s)
SSOR	18	0.010465




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02562586 0.04124584 0.05078364 0.05598621 0.05764372 0.05598718 0.0507848  0.04124577 0.02562278 0.        ]
 [0.         0.04124584 0.06857146 0.0859015  0.09551762 0.09860246 0.09552169 0.08590791 0.06857701 0.04124637 0.        ]
 [0.         0.05078364 0.0859015  0.10874306 0.1215898  0.12573517 0.12159582 0.10875282 0.08591077 0.05078686 0.        ]
 [0.         0.05598621 0.09551762 0.1215898  0.13637311 0.14116091 0.13638001 0.12160112 0.09552877 0.05599115 0.        ]
 [0.         0.05764372 0.09860246 0.12573517 0.14116091 0.14616282 0.14116776 0.12574651 0.09861389 0.05764943 0.        ]
 [0.         0.05598718 0.09552169 0.12159582 0.13638001 0.14116776 0.13638611 0.12160598 0.09553212 0.05599285 0.        ]
 [0.         0.0507848  0.08590791 0.10875282 0.12160112 0.12574651 0.12160598 0.10876098 0.08591641 0.05078975 0.        ]
 [0.         0.04124577 0.06857701 0.08591077 0.09552877 0.09861389 0.09553212 0.08591641 0.06858299 0.04124948 0.        ]
 [0.         0.02562278 0.04124637 0.05078686 0.05599115 0.05764943 0.05599285 0.05078975 0.04124948 0.02562482 0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 5: SSOR 方法计算结果

2.5 块 Jacobi 方法

2.5.1 算法原理

如果把每条线上的网格点看成一组，就能将对应的矩阵进行分块，化简得块 Jacobi 方法的迭代格式为：

$$A_{ij}v_j^{(k+1)} = v_{j-1}^{(k)} + v_{j+1}^{(k)} + b_j \tag{18}$$

对应的代码如下:

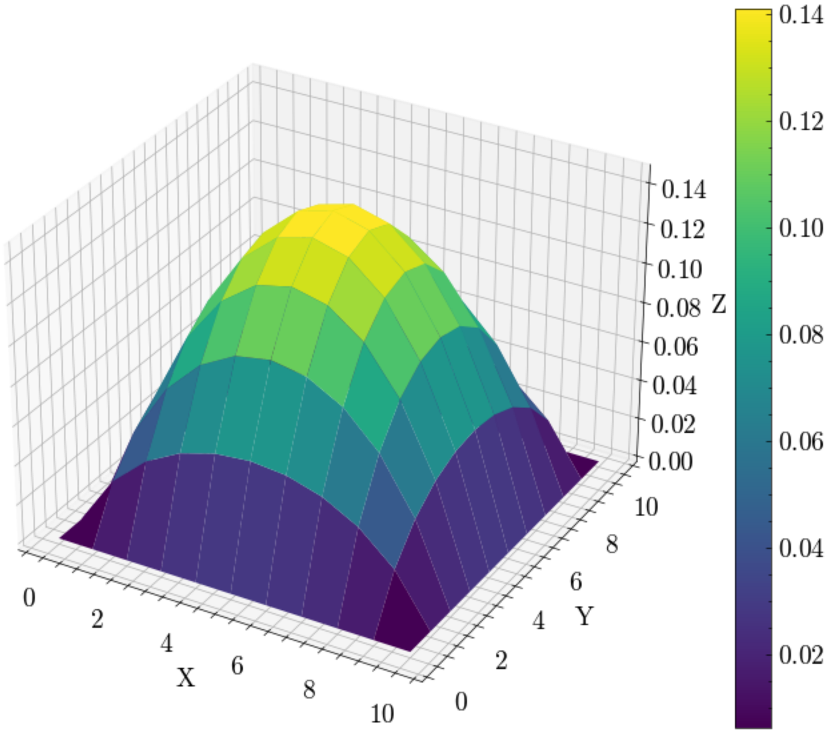
Listing 5: BJacobi

```
def BJ(n=9, max_iter=1000, tol=1e-5):
    u = np.zeros([n + 2, n + 2])
    h = 1 / (n + 1)
    f = np.full([n + 2, n + 2], h**2 * 2)
    a = -1 * np.ones(n)
    b = 4 * np.ones(n)
    c = -1 * np.ones(n)
    d = np.zeros(n)
    for k in range(max_iter):
        e = 0.0
        u_old = u.copy()
        for j in range(1, n + 1):
            u_o_n = u[:, j].copy()
            d = f[1 : n + 1, j] + u_old[1 : n + 1, j - 1] + u[1 : n + 1, j + 1]
            x = zg(a, b, c, d)
            u[1 : n + 1, j] = x
            e = e + np.linalg.norm(u_o_n - u[:, j], 1)
        if e / n**2 < tol:
            break
    return u, k + 1
```

2.5.2 计算结果

表 5: Algorithm: BJ

Algorithm	Iterations	Time (s)
BJ	69	0.014893




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02560818 0.04121804 0.0507471  0.05594326 0.05759828 0.05594326 0.0507471  0.04121804 0.02560818 0.        ]
 [0.         0.04121812 0.06852376 0.0858361  0.09543882 0.09851773 0.09543882 0.0858361  0.06852376 0.04121812 0.        ]
 [0.         0.0507471  0.08583588 0.10865188 0.12147938 0.12561619 0.12147938 0.10865188 0.08583588 0.0507471  0.        ]
 [0.         0.0559434  0.09543882 0.12147974 0.13623981 0.14101743 0.13623981 0.12147974 0.09543882 0.0559434  0.        ]
 [0.         0.05759828 0.09851746 0.12561619 0.14101699 0.14600819 0.14101699 0.12561619 0.09851746 0.05759828 0.        ]
 [0.         0.0559434  0.09543882 0.12147974 0.13623981 0.14101743 0.13623981 0.12147974 0.09543882 0.0559434  0.        ]
 [0.         0.0507471  0.08583588 0.10865188 0.12147938 0.12561619 0.12147938 0.10865188 0.08583588 0.0507471  0.        ]
 [0.         0.04121812 0.06852376 0.0858361  0.09543882 0.09851773 0.09543882 0.0858361  0.06852376 0.04121812 0.        ]
 [0.         0.02560818 0.04121804 0.0507471  0.05594326 0.05759828 0.05594326 0.0507471  0.04121804 0.02560818 0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 6: 块 Jacobi 方法计算结果

2.6 块 Gauss-Seidel 方法

2.6.1 算法原理

块 Gauss-Seidel 方法与块 Jacobi 方法相似，不同之处在于将当前迭代次数下得到的 v_{j-1} 用于后续计算。迭代格式如下：

$$A_{ij}v_j^{(k+1)} = v_{j-1}^{(k+1)} + v_{j+1}^{(k)} + b_j \tag{19}$$

相应的代码如下：

Listing 6: BG-S

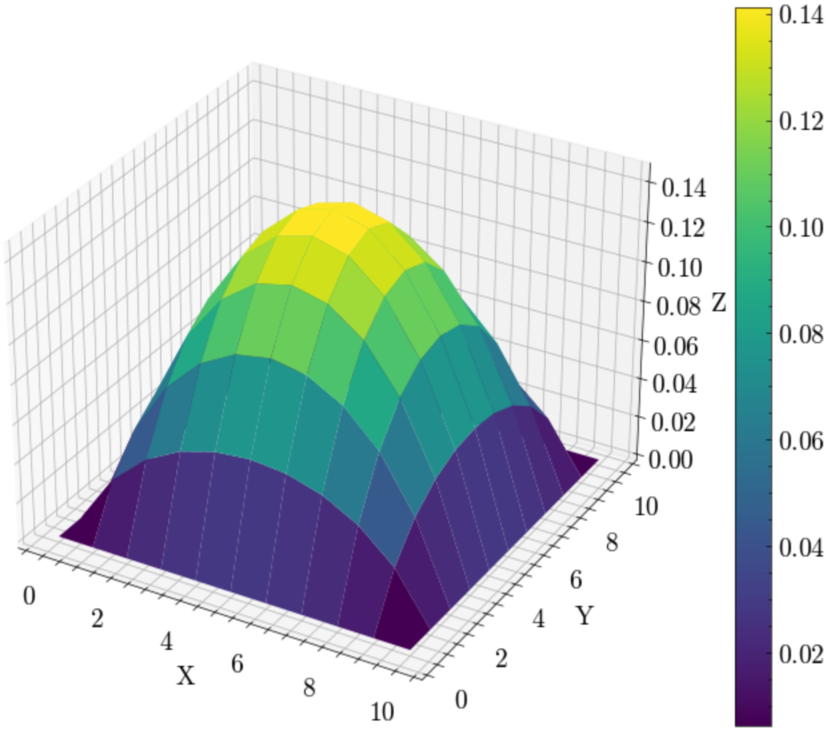
```python
def BGS(n=9, max_iter=1000, tol=1e-5):
```

```
    u = np.zeros([n + 2, n + 2])
    h = 1 / (n + 1)
    f = np.zeros([n + 2, n + 2])
    f[1 : n + 1, 1 : n + 1] = h**2 * 2
    a = -1 * np.ones(n)
    b = 4 * np.ones(n)
    c = -1 * np.ones(n)
    d = np.zeros(n)
    for k in range(max_iter):
        e = 0.0
        for j in range(1, n + 1):
            u_old = u[:, j].copy()
            d = f[1 : n + 1, j] + u[1 : n + 1, j - 1] + u[1 : n + 1, j + 1]
            x = zg(a, b, c, d)
            u[1 : n + 1, j] = x
            e = e + np.linalg.norm(u_old - u[:, j], 1)
        if e / n**2 < tol:
            break
    return u, k + 1
```

2.6.2 计算结果

表 6: Algorithm: BGS

Algorithm	Iterations	Time (s)
BGS	39	0.008421




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02561435 0.04123196 0.05076876 0.05597167 0.05763067 0.0559765  0.05077703 0.04124104 0.02562078 0.        ]
 [0.         0.04122985 0.06855024 0.08587731 0.09549285 0.09857934 0.09550205 0.08589304 0.06856751 0.0412421  0.        ]
 [0.         0.05076324 0.08587232 0.1087086  0.12155374 0.12570098 0.1215664  0.10873024 0.0858961  0.0507801  0.        ]
 [0.         0.05596238 0.09548166 0.12154642 0.13632723 0.14111711 0.13634211 0.12157186 0.09550961 0.05598219 0.        ]
 [0.         0.05761823 0.09856249 0.1256863  0.1411089  0.14611301 0.14112455 0.12571305 0.09859189 0.05763907 0.        ]
 [0.         0.05596238 0.09548166 0.12154642 0.13632723 0.14111711 0.13634211 0.12157186 0.09550961 0.05598219 0.        ]
 [0.         0.05076324 0.08587232 0.1087086  0.12155374 0.12570098 0.1215664  0.10873024 0.0858961  0.0507801  0.        ]
 [0.         0.04122985 0.06855024 0.08587731 0.09549285 0.09857934 0.09550205 0.08589304 0.06856751 0.0412421  0.        ]
 [0.         0.02561435 0.04123196 0.05076876 0.05597167 0.05763067 0.0559765  0.05077703 0.04124104 0.02562078 0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 7: 块 Gauss-Seidel 方法计算结果

2.7 块 SOR 方法

2.7.1 算法原理

块 SOR 方法在块 Gauss-Seidel 方法的基础上，引入松弛因子 *omega* 加速收敛，先通过块 Gauss-Seidel 方法计算出 $\bar{u}_i^{(k+1)}$，然后用松弛因子 ω 对 $\bar{u}_i^{(k+1)}$ 和 $u_i^{(k)}$ 进行线性组合。通过遍历的方式，取得迭代次数最小值对应的 ω 值为 1.43，具体的代码如下：

Listing 7: BSOR

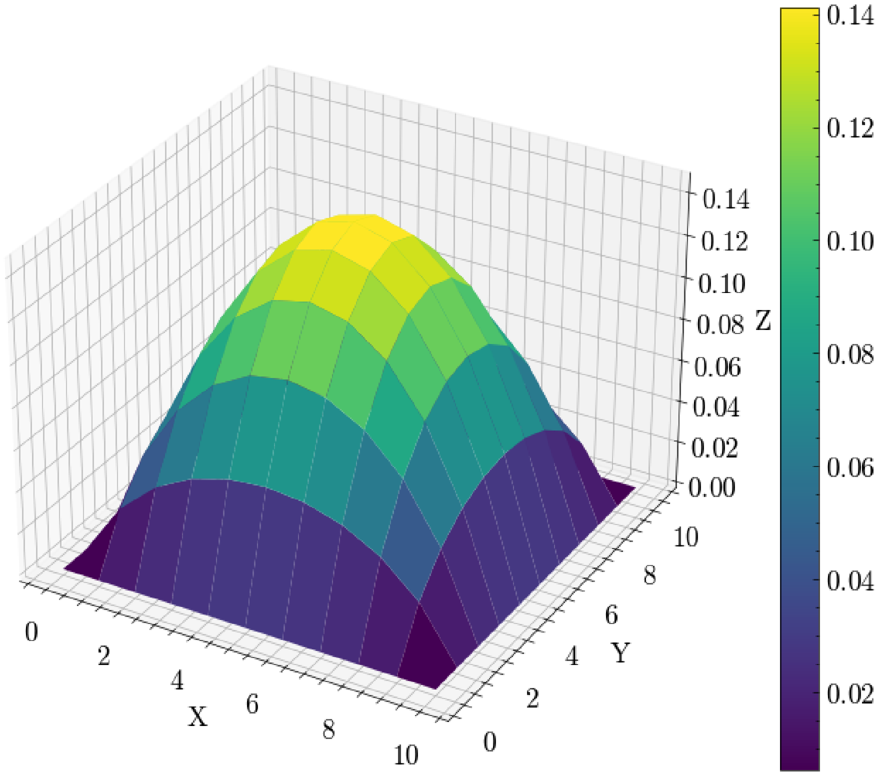
```
def BSOR(n=9, max_iter=1000, tol=1e-5, w=1.43):
    u = np.zeros([n + 2, n + 2])
    h = 1 / (n + 1)
```

```
    f = np.zeros([n + 2, n + 2])
    f[1 : n + 1, 1 : n + 1] = h**2 * 2
    a = -1 * np.ones(n)
    b = 4 * np.ones(n)
    c = -1 * np.ones(n)
    d = np.zeros(n)
    for k in range(max_iter):
        e = 0.0
        for j in range(1, n + 1):
            u_old = u[:, j].copy()
            d = f[1 : n + 1, j] + u[1 : n + 1, j - 1] + u[1 : n + 1, j + 1]
            x = zg(a, b, c, d)
            u[1 : n + 1, j] = w * x + (1 - w) * u_old[1 : n + 1]
            e = e + np.linalg.norm(u_old - u[:, j], 1)
        if e / n**2 < tol:
            break
    return u, k + 1
```

2.7.2 计算结果

表 7: Algorithm: BSOR

Algorithm	Iterations	Time (s)
BSOR	13	0.003174




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02562236 0.04124732 0.05078943 0.05599481 0.05765369 0.05599696 0.05079317 0.04125184 0.025626   0.        ]
 [0.         0.04124494 0.06857949 0.08591664 0.09553688 0.09862312 0.09554097 0.08592374 0.06858807 0.04125202 0.        ]
 [0.         0.05078399 0.08591262 0.10876279 0.12161437 0.12576126 0.12161997 0.1087725  0.0859244  0.05079375 0.        ]
 [0.         0.05598687 0.0955291  0.12161018 0.13639853 0.14118799 0.13640509 0.12162153 0.09554288 0.05599824 0.        ]
 [0.         0.05764404 0.09861243 0.12575335 0.14118389 0.14618754 0.14119078 0.12576528 0.09862687 0.05765594 0.        ]
 [0.         0.05598687 0.0955291  0.12161018 0.13639853 0.14118799 0.13640509 0.12162153 0.09554288 0.05599824 0.        ]
 [0.         0.05078399 0.08591262 0.10876279 0.12161437 0.12576126 0.12161997 0.1087725  0.0859244  0.05079375 0.        ]
 [0.         0.04124494 0.06857949 0.08591664 0.09553688 0.09862312 0.09554097 0.08592374 0.06858807 0.04125202 0.        ]
 [0.         0.02562236 0.04124732 0.05078943 0.05599481 0.05765369 0.05599696 0.05079317 0.04125184 0.025626   0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 8: 块 SOR 方法计算结果

2.8 块 SSOR 方法

2.8.1 算法原理

。通过遍历的方式，取得迭代次数最小值对应的 ω 值为 1.43,

Listing 8: BSSOR

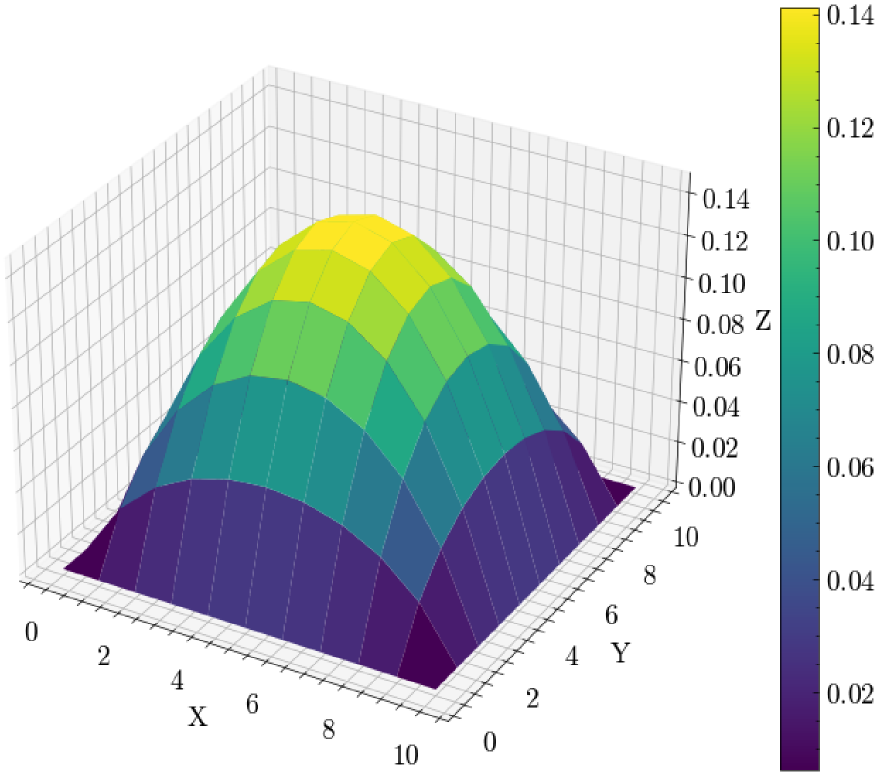
```
def BSSOR(n=9, max_iter=1000, tol=1e-5, w=1.43):
    u = np.zeros([n + 2, n + 2])
    um = np.zeros([n + 2, n + 2])
```

```python
    h = 1 / (n + 1)
    f = np.zeros([n + 2, n + 2])
    f[1 : n + 1, 1 : n + 1] = h**2 * 2
    a = -1 * np.ones(n)
    b = 4 * np.ones(n)
    c = -1 * np.ones(n)
    for k in range(max_iter):
        e1 = 0.0
        e2 = 0.0
        for j in range(1, n + 1):
            u_old = um[:, j].copy()
            d1 = f[1 : n + 1, j] + um[1 : n + 1, j - 1] + um[1 : n + 1, j + 1]
            x = zg(a, b, c, d1)
            um[1 : n + 1, j] = w * x + (1 - w) * u_old[1 : n + 1]
            e1 = e1 + np.linalg.norm(u_old - um[:, j], 1)
        for j in range(n, 0, -1):
            um_old = um[:, j].copy()
            d2 = f[1 : n + 1, j] + um[1 : n + 1, j - 1] + u[1 : n + 1, j + 1]
            x2 = zg(a, b, c, d2)
            u[1 : n + 1, j] = w * x2 + (1 - w) * um_old[1 : n + 1]
            e2 = e2 + np.linalg.norm(um_old - u[:, j], 1)

        if (e1 + e2) / n**2 < tol:
            break
    return u, k + 1
```

2.8.2 计算结果

表 8: Algorithm: BSSOR

Algorithm	Iterations	Time (s)
BSSOR	13	0.006439




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02562566 0.0412493  0.05079047 0.05599532 0.05765369 0.05599681 0.05079298 0.04125169 0.02562591 0.        ]
 [0.         0.04125142 0.06858318 0.08591861 0.09553784 0.09862313 0.09554068 0.08592337 0.06858777 0.04125185 0.        ]
 [0.         0.05079293 0.08591765 0.10876544 0.12161569 0.12576127 0.12161958 0.10877199 0.08592398 0.05079352 0.        ]
 [0.         0.05599724 0.09553496 0.12161327 0.13640007 0.141188   0.13640463 0.12162095 0.09554239 0.05599797 0.        ]
 [0.         0.05765487 0.09861855 0.1257566  0.1411855  0.14618754 0.1411903  0.12576466 0.09862635 0.05765566 0.        ]
 [0.         0.05599724 0.09553496 0.12161327 0.13640007 0.141188   0.13640463 0.12162095 0.09554239 0.05599797 0.        ]
 [0.         0.05079293 0.08591765 0.10876544 0.12161569 0.12576127 0.12161958 0.10877199 0.08592398 0.05079352 0.        ]
 [0.         0.04125142 0.06858318 0.08591861 0.09553784 0.09862313 0.09554068 0.08592337 0.06858777 0.04125185 0.        ]
 [0.         0.02562566 0.0412493  0.05079047 0.05599532 0.05765369 0.05599681 0.05079298 0.04125169 0.02562591 0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 9: 块 SSOR 方法计算结果

2.9 最速下降法

2.9.1 算法原理

若方程组 Ax=b 的系数矩阵 A 是 n 阶对称正定阵，则 x* 是方程组解的充要条件是，x* 是二次函数 $\varphi(x)$ 的极小值点，即

$$x^* = A^{-1}b \Leftrightarrow \varphi(x^*) = \min \varphi(x) \tag{20}$$

其中，$\varphi(x) = \frac{1}{2}(Ax, x) - (b, x) = \frac{1}{2}\sum_{i,j=1}^{n} x_i a_{ij} x_j - \sum_{i=1}^{n} b_i x_i$

二次函数 $\varphi(x)$ 在任意点处沿梯度方向增长最快，因此负梯度方向是下降最快的方向。在 $x^{(k+1)} = x^{(k)} + \alpha_k P^{(k)}$ 中应选

$$P^{(k)} = -grad(\varphi(x^{(k)})) = r^{(k)} = b - Ax^{(k)} \tag{21}$$

最优步长由下式计算：

$$\alpha_k = \frac{(r^{(k)}, r^{(k)})}{(Ar^{(k)}, r^{(k)})} \tag{22}$$

最速下降算法的计算流程如下：

1. 给定初值 x_0
2. 计算残值向量 $r^{(k)} = b - Ax^{(k)}$
3. 计算步长 $\alpha_k = (r^{(k)}, r^{(k)})/(Ar^{(k)}, r^{(k)})$
4. 计算 $x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)}$

把它们转换成以点更新的模式进行编程：

Listing 9: GD

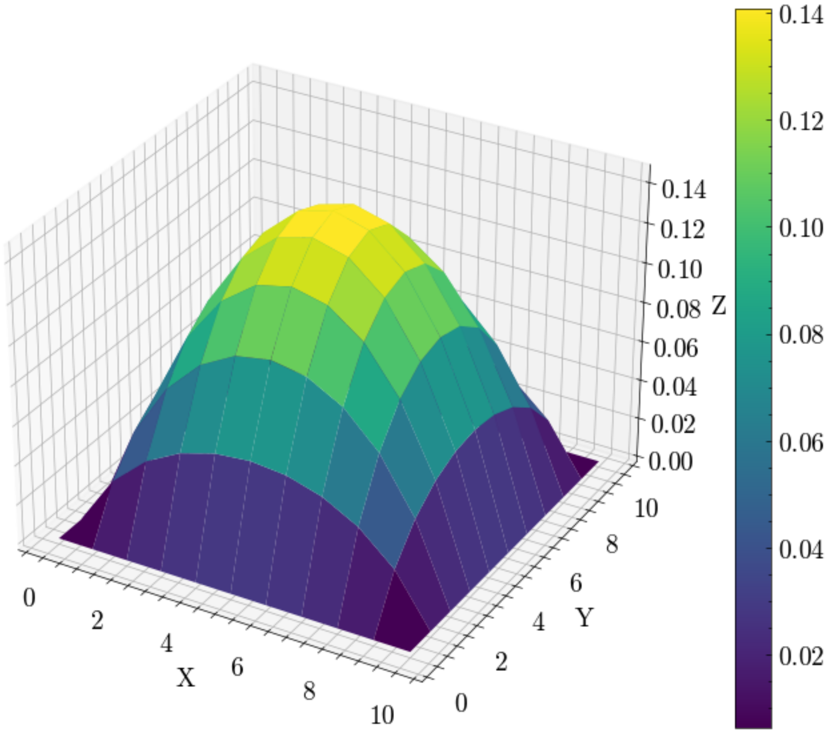
```
def GD(n=9, max_iter=1000, tol=1e-5):
    u = np.zeros([n + 2, n + 2])
    h = 1 / (n + 1)
    f = np.full([n + 2, n + 2], h**2 * 2)

    u[0, :] = 0
    u[-1, :] = 0
    u[:, 0] = 0
    u[:, -1] = 0
    for k in range(max_iter):
        r = np.zeros([n + 2, n + 2])

        # 计算残差
        for j in range(1, n + 1):
            for i in range(1, n + 1):
                r[i, j] = f[i, j] - (
                    4 * u[i, j] - u[i - 1, j] - u[i + 1, j] - u[i, j - 1] - u[i, j + 1]
                )

        # 计算步长 alpha_k
        alpha_k = np.sum(r**2) / np.sum(
            4 * r[1:-1, 1:-1] ** 2
            - r[1:-1, :-2] * r[1:-1, 1:-1]
            - r[1:-1, 2:] * r[1:-1, 1:-1]
            - r[:-2, 1:-1] * r[1:-1, 1:-1]
            - r[2:, 1:-1] * r[1:-1, 1:-1]
        )

        e = 0.0
```

```python
        # 更新解 u
        for j in range(1, n + 1):
            for i in range(1, n + 1):
                uo = u[i, j].copy()
                u[i, j] = u[i, j] + alpha_k * r[i, j]
                e = e + np.abs(u[i, j] - uo)

        # 检查收敛性
        if e / n**2 < tol:
            break

    return u, k + 1
```

2.9.2 计算结果

表 9: Algorithm: GD

Algorithm	Iterations	Time (s)
GD	108	0.035564




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.0255901  0.04118699 0.05069976 0.05589303 0.05753977 0.05589303 0.05069976 0.04118699 0.0255901  0.        ]
 [0.         0.04118699 0.0684585  0.08575459 0.09533323 0.09841698 0.09533323 0.08575459 0.0684585  0.04118699 0.        ]
 [0.         0.05069976 0.08575459 0.10852794 0.12134786 0.125463   0.12134786 0.10852794 0.08575459 0.05069976 0.        ]
 [0.         0.05589303 0.09533323 0.12134786 0.13606896 0.14085441 0.13606896 0.12134786 0.09533323 0.05589303 0.        ]
 [0.         0.05753977 0.09841698 0.125463   0.14085441 0.14581884 0.14085441 0.125463   0.09841698 0.05753977 0.        ]
 [0.         0.05589303 0.09533323 0.12134786 0.13606896 0.14085441 0.13606896 0.12134786 0.09533323 0.05589303 0.        ]
 [0.         0.05069976 0.08575459 0.10852794 0.12134786 0.125463   0.12134786 0.10852794 0.08575459 0.05069976 0.        ]
 [0.         0.04118699 0.0684585  0.08575459 0.09533323 0.09841698 0.09533323 0.08575459 0.0684585  0.04118699 0.        ]
 [0.         0.0255901  0.04118699 0.05069976 0.05589303 0.05753977 0.05589303 0.05069976 0.04118699 0.0255901  0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 10: 最速下降方法计算结果

2.10 共轭梯度法

2.10.1 算法原理

使用共轭梯度法求解对称正定的线性方程组 Ax=b 的一般流程如下。首先计算初值:

$$x^{(0)}, r^{(0)} = b - Ax^{(0)}, P^{(0)} = r^{(0)} \tag{23}$$

对 $k = 1, 2, 3, \cdots$:

$$
\begin{aligned}
\alpha_k &= \frac{(r^{(k)}, \mathbf{P}^{(k)})}{(\mathbf{P}^{(k)}, \mathbf{AP}^{(k)})} \\
x^{(k+1)} &= x^{(k)} + \alpha_k \mathbf{P}^{(k)} \\
\mathbf{P}^{(k+1)} &= r^{(k+1)} + \beta_k \mathbf{P}^{(k)} \\
\beta_k &= -\frac{(r^{(k+1)}, \mathbf{AP}^{(k)})}{(\mathbf{P}^{(k)}, \mathbf{AP}^{(k)})} \\
r^{(k+1)} &= b - \mathbf{A}x^{(k+1)}
\end{aligned}
\tag{24}
$$

把上述流程转换成点更新的代码:

Listing 10: CG

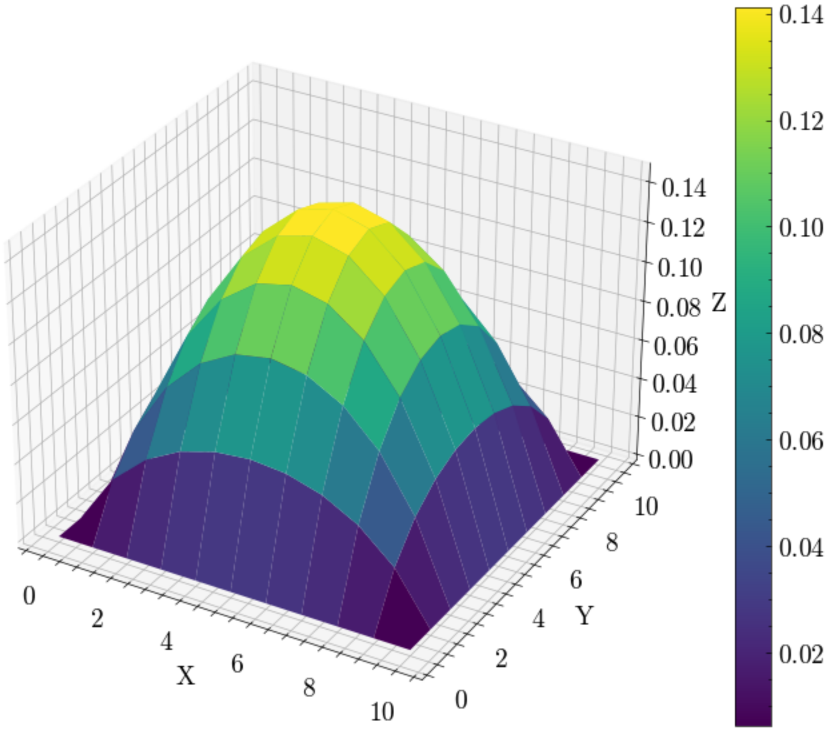
```
def CG(n=9, max_iter=1000, tol=1e-5):
    u = np.zeros([n + 2, n + 2])
    h = 1 / (n + 1)
    f = np.full([n + 2, n + 2], h**2 * 2)
    u[0, :] = 0
    u[-1, :] = 0
    u[:, 0] = 0
    u[:, -1] = 0

    r = np.zeros([n + 2, n + 2])
    for j in range(1, n + 1):
        for i in range(1, n + 1):
            r[i, j] = f[i, j] - (
                4 * u[i, j] - u[i - 1, j] - u[i + 1, j] - u[i, j - 1] - u[i, j + 1]
            )
    p = r.copy()

    for k in range(max_iter):

        # 计算步长 alpha_k
        alpha_k = np.sum(r**2) / np.sum(
            4 * p[1:-1, 1:-1] * p[1:-1, 1:-1]
            - p[1:-1, :-2] * p[1:-1, 1:-1]
            - p[1:-1, 2:] * p[1:-1, 1:-1]
            - p[:-2, 1:-1] * p[1:-1, 1:-1]
            - p[2:, 1:-1] * p[1:-1, 1:-1]
        )

        e = 0.0
        # 更新解 u
        for j in range(1, n + 1):
            for i in range(1, n + 1):
                uo = u[i, j].copy()
                u[i, j] = u[i, j] + alpha_k * p[i, j]
                e = e + np.abs(u[i, j] - uo)

        # 检查收敛性
```

```
        if e / n**2 < tol:
            break

        # 更新 r
        r_old = r.copy()
        for j in range(1, n + 1):
            for i in range(1, n + 1):
                r[i, j] = r[i, j] - alpha_k * (
                    4 * p[i, j] - p[i - 1, j] - p[i + 1, j] - p[i, j - 1] - p[i, j + 1]
                )

        # 计算 beta_k
        beta_k = np.sum(r**2) / np.sum(r_old**2)

        # 计算 p(k+1)
        for j in range(1, n + 1):
            for i in range(1, n + 1):
                p[i, j] = r[i, j] + beta_k * p[i, j]

    return u, k + 1
```

2.10.2 计算结果

表 10: Algorithm: CG

Algorithm	Iterations	Time (s)
CG	11	0.004252




```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]
 [0.         0.02562343 0.0412507  0.05079571 0.05600214 0.05765604 0.05600214 0.05079571 0.0412507  0.02562343 0.        ]
 [0.         0.0412507  0.0685883  0.08592628 0.09554331 0.09862969 0.09554331 0.08592628 0.0685883  0.0412507  0.        ]
 [0.         0.05079571 0.08592628 0.10877765 0.12162233 0.12576653 0.12162233 0.10877765 0.08592628 0.05079571 0.        ]
 [0.         0.05600214 0.09554331 0.12162233 0.13641444 0.14119822 0.13641444 0.12162233 0.09554331 0.05600214 0.        ]
 [0.         0.05765604 0.09862969 0.12576653 0.14119822 0.1461922  0.14119822 0.12576653 0.09862969 0.05765604 0.        ]
 [0.         0.05600214 0.09554331 0.12162233 0.13641444 0.14119822 0.13641444 0.12162233 0.09554331 0.05600214 0.        ]
 [0.         0.05079571 0.08592628 0.10877765 0.12162233 0.12576653 0.12162233 0.10877765 0.08592628 0.05079571 0.        ]
 [0.         0.0412507  0.0685883  0.08592628 0.09554331 0.09862969 0.09554331 0.08592628 0.0685883  0.0412507  0.        ]
 [0.         0.02562343 0.0412507  0.05079571 0.05600214 0.05765604 0.05600214 0.05079571 0.0412507  0.02562343 0.        ]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.        ]]
```

图 11: 共轭梯度方法计算结果

3　不同算法对比

当 SOR、SSOR、块 SOR、块 SSOR 算法分别取 ω 的最优值下，对这 10 个算法进行对比，对比的指标包括迭代次数和时间，计算算法时间时，采用多次取平均值的方式。得到的结果如下：

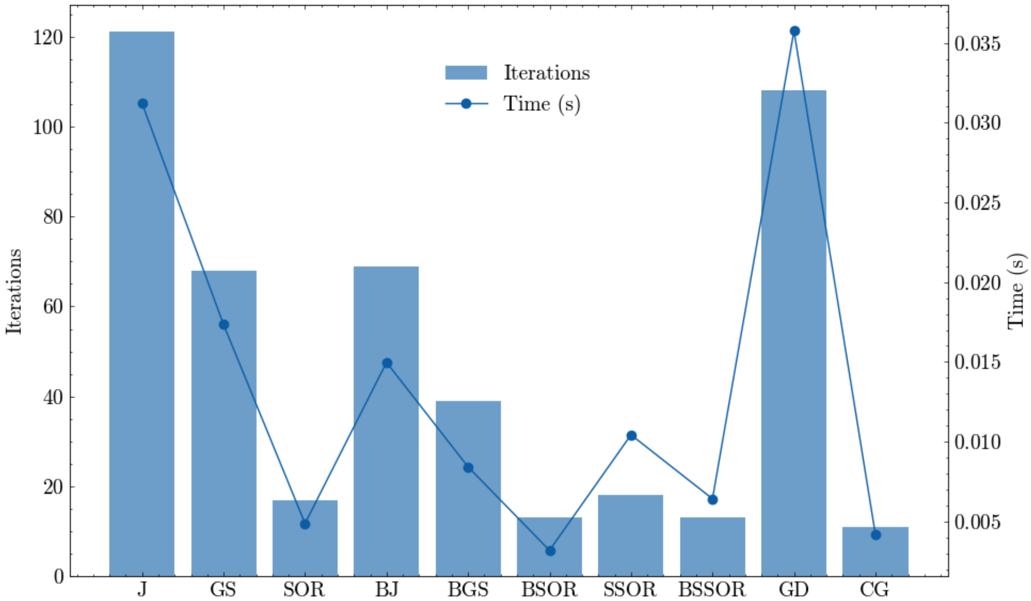


图 12: 不同算法对比

表 11: 算法迭代次数和时间

算法	迭代次数	时间 (s)
J	121	0.031305
GS	68	0.017478
SOR	17	0.004884
BJ	69	0.014909
BGS	39	0.008416
BSOR	13	0.003187
SSOR	18	0.010527
BSSOR	13	0.006372
GD	108	0.035893
CG	11	0.004155

4 代码

完整代码放在 GitHub 上，这是我的 GitHub 项目链接：https://github.com/cospeng/NA。

参考文献

[1] 张明. *应用数值分析 (第 4 版)*. 石油工业出版社, 4 edition, 8 2012.