计算方法（A）

实验（四） 实现共轭梯度法

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2022年5月26日
学　　院	信息学院
课程名称	计算方法（A）

实验（四） 实现共轭梯度法

目录

1 原理与实现 **1**

2 运行结果 **1**

3 基准测试 **1**

3.1 运行时间 1

3.2 与高斯-赛德尔迭代法的比较 2

4 结论 **2**

A 附录：代码 **3**

A.1 共轭梯度实现 3

1 原理与实现

共轭梯度法是一种把求方程组的解的问题转化为一个求二次函数极值点的问题的方法。构造二次函数(1)

$$\phi(x) = \frac{1}{2}(\mathbf{Ax}, \mathbf{x}) - (\mathbf{b}, \mathbf{x}) \tag{1}$$

它的梯度是(2)

$$\nabla\phi(x) = \mathbf{Ax} - \mathbf{b} \tag{2}$$

因此求极值点可以求出对应方程组的解。共轭梯度法是指在每一步梯度下降时选择的搜索方向 $p^{(0)}, \cdots, p^{(n)}$是$A-$共轭向量组，选取的修正方向和幅度是(3)

$$\begin{aligned} \mathbf{p}^{(k)} &= \mathbf{r}^{(k)} + \beta_{k-1}\mathbf{p}^{(k-1)} \\ \beta_{k-1} &= -\frac{(\mathbf{r}^{(k)}, \mathbf{Ap^{(k-1)}})}{(\mathbf{p}^{(k)}, \mathbf{Ap^{(k-1)}})} \\ \alpha_{k-1} &= \frac{(\mathbf{r}^{(k)}, \mathbf{r}^{(k)})}{(\mathbf{p}^{(k)}, \mathbf{Ap^{(k)}})} \end{aligned} \tag{3}$$

本程序在*cg/conj_grad.py*中实现了共轭梯度法。同时为了比较性能，也实现了高斯赛德尔迭代法。通过使用**Numpy**矩阵A和向量b构造一个*Equation*类，可以实现使用共轭梯度法和高斯赛德尔法解方程组，具有良好的可扩展性。

本程序依赖以下一个第三方库：

- **Numpy**: 用于加速线性代数运算

2 运行结果

解方程组(4)，可得结果$[3, 1, 5.773\,159\,73 \times 10^{-15}]^{\mathsf{T}}$。正确结果是$[3, 1, 0]^{\mathsf{T}}$。由于$5.773\,159\,73 \times 10^{-15} \approx 0$，可见共轭梯度法作为一种优化算法，已经给出了足够正确的答案。

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \tag{4}$$

3 基准测试

为了测试程序的性能，本节在以下条件下测量运行时间，并与高斯赛德尔法进行性能上的比较：

- 系数矩阵A： n维系数矩阵A的生成方法是$A_n = P_n + R_n R_n^{\mathsf{T}}$，其中$P_n$是$n$维Pascal矩阵。 Pascal矩阵是元素是组合数的矩阵，是一个正定对称的矩阵。R_n是一个n维随机矩阵，每个元素在$[0, 1)$间均匀分布。这样的生成方式可以保证矩阵A_n大概率是一个正定对称的矩阵，并且与直接使用Pascal矩阵相比，可以排除**Numpy**库对特殊矩阵运算的优化。
- 向量b：向量b使用了一个n维随机向量。
- 测量方法：通过测量5次运行时间取平均值排除随机因素的干扰。

3.1 运行时间

首先测量问题规模n变化时运行时间的变化。如图 3.1。可以发现总体来说除了问题规模较小的部分之外，

运行时间基本不随问题规模扩大而增加。这说明迭代次数较为稳定。当n较小时有显著变化的原因可能是因为预处理的时间影响相对迭代的时间而言在较小的问题规模下显得较为显著。

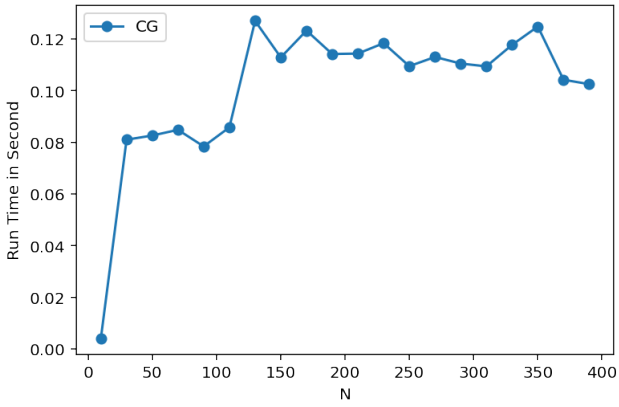

图 1: 平均运行时间

3.2 与高斯-赛德尔迭代法的比较

按前述测试条件，同时使用高斯赛德尔法和共轭梯度法解方程组，并记录运行时间，可以得到图 3.2。可以发现，在问题规模较小的情况下，高斯赛德尔法与共轭梯度法运行效率相近。随着问题规模的扩大，高

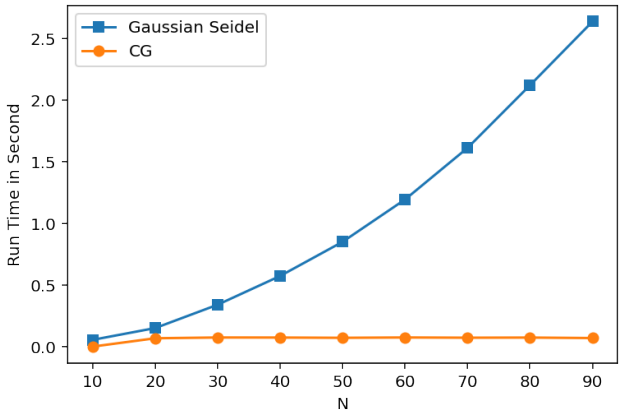

图 2: 平均运行时间比较

斯赛德尔法的运行时间会比共轭梯度法的运行时间更长，呈指数级增长。这表明高斯赛德尔迭代法的迭代次数较为不稳定，并且快速增长，难以解决规模较大的问题。

4 结论

通过实际实现和测试可以发现，在系数矩阵正定时共轭梯度法是一个既方便实现，又高效的方程组解法。它的迭代次数较少，运行速度较快。

A 附录：代码

A.1 共轭梯度实现

代码 1: 共轭梯度实现

```python
import numpy as np


class Equation:
    def __init__(self, coeff, vals):
        self.a = np.asmatrix(coeff, dtype=np. float)
        self.b = np.asarray(vals, dtype=np. float)
        self.solve = self._solve_cg
        assert self.a.shape[1] == self.b.shape[0]

    def use_naive(self):
        self.solve = self._solve_naive
        return self

    def use_conjugate_gradient(self):
        self.solve = self._solve_cg
        return self

    def _solve_naive(self, x0=None, max_iter=512):
        x = x0
        if x0 is None:
            x = np.zeros_like(self.b, dtype=np. float)

        n = self.a.shape[0]
         iter = 0
        prev_x = x.copy()
        while  iter < max_iter:
            for j in  range(0, n):
                d = self.b[j]

                for i in  range(0, n):
                    if j != i:
                        d -= self.a[j, i] * x[i]
                x[j] = d / self.a[j, j]

            if np.allclose(x, prev_x, rtol=1e-5, atol=1e-5, equal_nan=True):
                return x
            prev_x = x.copy()
             iter += 1

        return x

    def _solve_cg(self, x0=None, max_iter=512):
        if x0 is None:
            x0 = np.zeros_like(self.b, dtype=np. float)

        a = np.asmatrix(self.a, dtype=np. float)
        b = np.asmatrix(self.b, dtype=np. float).transpose()
        x = np.asmatrix(x0, dtype=np. float).transpose()
```

```
50
51          r = b - a * x
52          p = r
53
54          zeros = np.zeros_like(x)
55
56           iter = 0
57          while  iter <= max_iter:
58              if np.allclose(r, zeros):
59                  break
60
61              ap = a * p
62
63              if np.allclose(ap, zeros):
64                  break
65
66              alpha = ((r.transpose() * r) / (p.transpose() * ap))[0, 0]
67
68              x += alpha * p
69
70              r1 = r - alpha * ap
71
72              beta = ((r1.transpose() * r1) / (r.transpose() * r))[0, 0]
73
74              p = r1 + beta * p
75
76              r = r1
77               iter += 1
78
79          return x
```