

Title

Alphabetium

2023 年 4 月 2 日

1 用定义函数的形式写出jacobi方法

```
1  import numpy as np
2
3  def jacobi(A, b, x0, max_iter=1000, eps=1e-8):
4      """
5      Jacobi method for solving linear equations  $Ax = b$ .
6      A: coefficient matrix
7      b: constant vector
8      x0: initial approximation
9      max_iter: maximum number of iterations
10     eps: tolerance for stopping criterion
11     """
12     n = A.shape[0] # dimension of the problem
13     x = x0.copy() # make a copy of the initial approximation
14     for k in range(max_iter):
15         x1 = np.zeros_like(x)
16         for i in range(n):
17             x1[i] = (b[i] - np.dot(A[i, :], x[:i]) - np.dot(A[i, i
+1:], x[i+1:])) / A[i, i]
```

```

18         if np.linalg.norm(x1 - x) < eps:
19             break
20         x = x1
21     return x

```

Listing 1: 0.1s

Q:详细解释`np.dot(A[i, :i], x[:i])`

A:

`np.dot(A[i, :i], x[:i])` 是计算向量内积的 numpy 函数。在这里，它用于计算 A 矩阵第 i 行的前 i 个元素和当前迭代向量 x 的前 i 个分量的内积。

具体来说， $A[i, :i]$ 是 A 矩阵第 i 行的前 i 个元素， $x[:i]$ 是当前迭代向量 x 的前 i 个分量，因此 `np.dot(A[i, :i], x[:i])` 就是这两个向量的内积。它的计算公式为：

$$\sum_{j=0}^{i-1} A_{ij}x_j$$

其中， A_{ij} 是矩阵 A 的第 i 行第 j 列的元素， x_j 是向量 x 的第 j 个分量。

Q:详细解释`np.dot(A[i, i+1:], x[i+1:])`

A:

`np.dot(A[i, i+1:], x[i+1:])` 同样是计算向量内积的 numpy 函数。在这里，它用于计算 A 矩阵第 i 行的后 $n - i - 1$ 个元素和当前迭代向量 x 的后 $n - i - 1$ 个分量的内积。

具体来说， $A[i, i+1:]$ 是 A 矩阵第 i 行的后 $n - i - 1$ 个元素， $x[i+1:]$ 是当前迭代向量 x 的后 $n - i - 1$ 个分量，因此 `np.dot(A[i, i+1:], x[i+1:])` 就是这两个向量的内积。它的计算公式为：

$$\sum_{j=i+1}^{n-1} A_{ij}x_j$$

其中， A_{ij} 是矩阵 A 的第 i 行第 j 列的元素， x_j 是向量 x 的第 j 个分量。

Q:你可以帮我看我的代码哪里有问题吗？

```

1  def Jacobi(A, b, x_0, eps , max_iter):
2
3      n = len(A)
4
5      x = x_0.copy()
6
7      for i in range(max_iter):
8
9          x1 = np.zeros(n)
10
11         for j in range(n):
12
13             s1 = np.dot(A[j,:j],x1[:j])
14
15             x1[j] = (b[j] - s1) / A[j, j]
16
17         if np.allclose(x, x1, rtol=eps):
18
19             return x1
20
21         x = x1
22
23     return x

```

A:

您的 Jacobi 方法代码看起来没有明显的问题，但是我注意到在计算 s_1 的时候，您的代码中写成了 `np.dot(A[j,:j],x1[:j])`，这里应该是 `np.dot(A[j,:j],x[:j])`，因为 x_1 还没有被更新。

另外，您使用了 `np.allclose(x, x1, rtol=eps)` 来判断两个向量是否相等。这里需要注意的是，`allclose` 函数比较的是两个向量之间的相对误差，而不是绝对误差。这意味着，当两个向量的分量值都很小的时候，相对误差可能会比绝对误差更重要，因为分母比分子大很多。如果您更关心绝对误差，可以使用 `np.linalg.norm(x1 - x, ord=np.inf)` ; `eps` 来代替 `allclose` 函数。

Q: 我们注意到jacobi方法的数学表示是 $x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$ ，看起来只需要减去一个 $\sum_{j \neq i} a_{ij} x_j^{(k)}$ 也就是我算法里面的a1，那为何我只减去了a1没有办法达成，还需要再减去`np.dot(A[j, j+1:], x[j+1:])`才可以正常使用？

A:

您提到的Jacobi方法数学公式中确实只有一个 $\sum_{j \neq i} a_{ij} x_j^{(k)}$ ，但是在Python代码实现时，我们需要将其分解成两部分，即 $a_1 = \sum_{j=1}^{i-1} a_{ij} x_j^{(k)}$ 和 $a_2 = \sum_{j=i+1}^n a_{ij} x_j^{(k)}$ 。

这是因为在Python中，使用numpy进行矩阵运算时，使用的是向量化运算，即一次性计算整个向量，而不是逐个元素计算。因此，我们需要将矩阵的某些部分单独取出来进行运算。

具体来说， $\text{np.dot}(A[j, :i], x[:i])$ 对应的是 a_1 ，它计算的是矩阵A第j行前i个元素和向量x前i个元素的点积，对应的就是公式中的 $\sum_{j=1}^{i-1} a_{ij} x_j^{(k)}$ ； $\text{np.dot}(A[j, i+1:], x[i+1:])$ 对应的是 a_2 ，它计算的是矩阵A第j行i+1到n个元素和向量x的i+1到n个元素的点积，对应的就是公式中的 $\sum_{j=i+1}^n a_{ij} x_j^{(k)}$ 。

因此，在实现Jacobi方法的Python代码中，需要将这两个部分分别计算并相加，才能得到正确的结果。

2 123

一根光纤中的模式频率可以使用波动方程计算：

$$\nabla^2 \vec{E} - \frac{1}{c^2} \frac{\partial^2 \vec{E}}{\partial t^2} = 0$$

其中， \vec{E} 是电场， c 是真空中光速， ∇^2 是拉普拉斯算子。

假设电场可分为径向和方位角两个部分，则我们可以写成：

$$\vec{E}(r, \theta, z, t) = \vec{E}(r) e^{-i\beta z} e^{i\omega t} \vec{u}(\theta)$$

其中 r 是距离光纤轴线的径向距离， θ 是方位角度数， z 是沿着光纤的距离， β 是传播常数， ω 是角频率， $\vec{u}(\theta)$ 是方位角单位矢量。

将这个表达式代入波动方程并简化后得到：

$$\frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial \vec{E}}{\partial r} \right) + (\beta^2 - \frac{\omega^2}{c^2} n^2(r)) \text{vec} E = 0$$

其中 $n(r)$ 表示在径向距离 r 处的折射率。

该方程可以通过适当的边界条件求解，例如，在芯层-包层边界处电场及其导数的连续性。

模式频率与其角频率 ω 有关：

$$\omega = \frac{2\pi c}{\lambda}$$

其中， λ 是光在光纤中的波长。

因此，模式频率可以表示为：

$$f = \frac{\omega}{2\pi} = \frac{c}{\lambda}$$

其中 f 是模式的频率。

总之，在光纤中的模式频率与其波长有关，并且可以使用波动方程和适当的边界条件进行计算。该频率由光纤的折射率分布以及芯区域大小和形状决定。