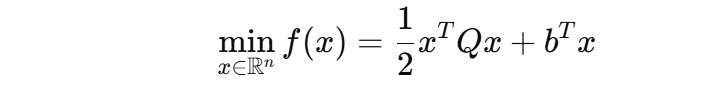
**数学建模第二次作业**

**第一部分（解决优化问题）**

**题目概述：**

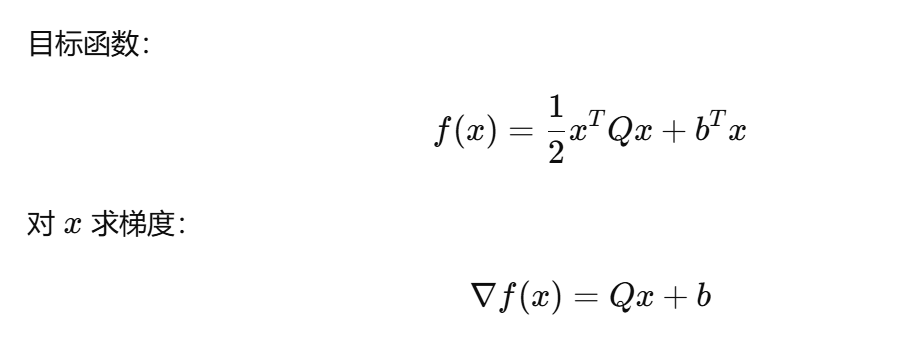
|  |
| --- |
| 要解决的优化问题是一个标准的二次型优化问题： |



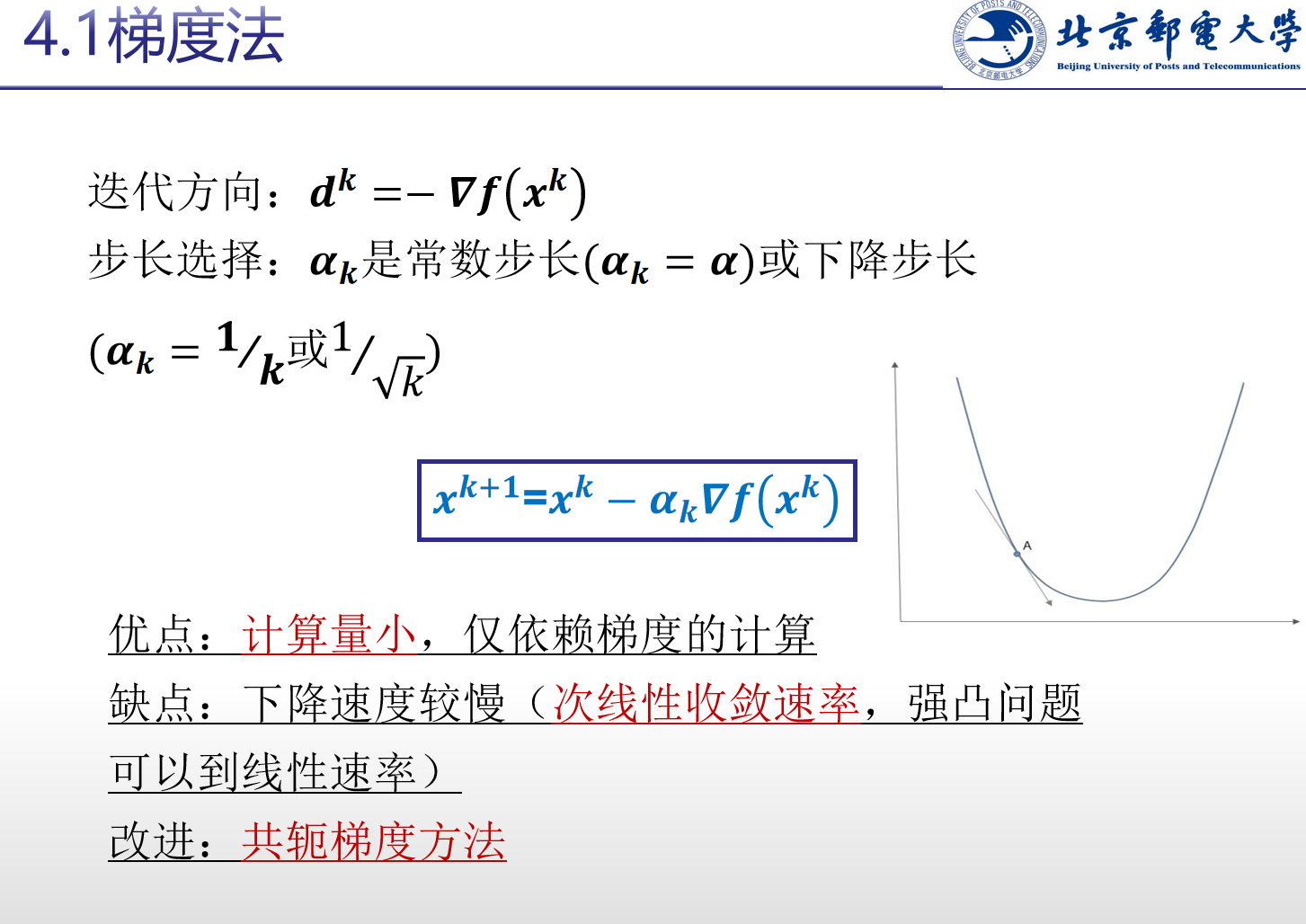
|  |
| --- |
| 其中，矩阵 QQQ 是通过生成服从标准正态分布的列向量 vvv 和服从均匀分布的向量 ϵ\epsilonϵ，根据公式 **Q=vvT+diag(ϵ)Q = v v^T + \text{diag}(\epsilon)Q=vvT+diag(ϵ)** 来生成。向量 **b** 也是从标准正态分布生成的。 |

**梯度下降法**

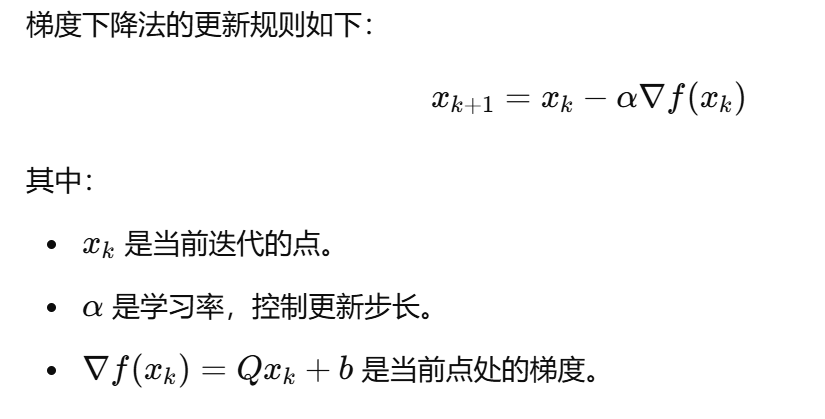
**1.明确梯度下降法**



**2.根据ppt讲解思考解题过程**



根据老师的ppt可以得知



**3.结合上述知识，给出伪代码**

**步骤 1：初始化**

* 随机生成初始点
* 设置学习率 （类似步长）alphaα 和收敛阈值 ϵ。

**步骤 2：计算梯度**

* 计算 ∇f(xk)=Qxk+b\nabla f(x\_k) = Qx\_k + b∇f(xk)=Qxk+b。

**步骤 3：更新点**

* 根据更新规则，不断更新迭代。

**步骤 4：检查终止条件**

* 迭代次数超过最大值，则终止。

**4.具体代码解释与过程说明**

1. **生成数据**：

* 使用题目中给定的生成方式生成 Q和 b。

|  |
| --- |
| Python def generateData(dimension):  *"""生成对称半正定矩阵 matrixQ 和向量 vectorB"""* randomV = np.random.randn(dimension)  epsilon = np.random.uniform(0, 1, dimension)  matrixQ = np.outer(randomV, randomV) + np.diag(epsilon) # 生成 matrixQ  vectorB = np.random.randn(dimension) # 生成 vectorB  return matrixQ, vectorB |

1. **初始化及参数设计**：

* solutionX = np.**random**.randn(dimension) # 随机初始化 solutionX
* *learningRate - 学习率，默认为0.01,****类似于ppt上的步长***  
  *tolerance - 收敛的容忍度，默认为1e-6（****不同取值结果不同，后续会补充说明****）*  
  *maxIterations -* ***最大迭代次数****，默认为1000*
* 为了*方便计算和观测结果，设置****维数dimension*** *= 5*

1. **梯度计算**：

* 每次迭代中，根据 Q和 b 计算当前点的梯度 ∇f

1. **更新规则**：

* 使用预设的学习率 （α）进行更新。

1. **收敛判断**：

* 检查梯度的范数是否小于给定的阈值 ϵ\epsilonϵ，如果满足条件，算法停止，输出当前解。

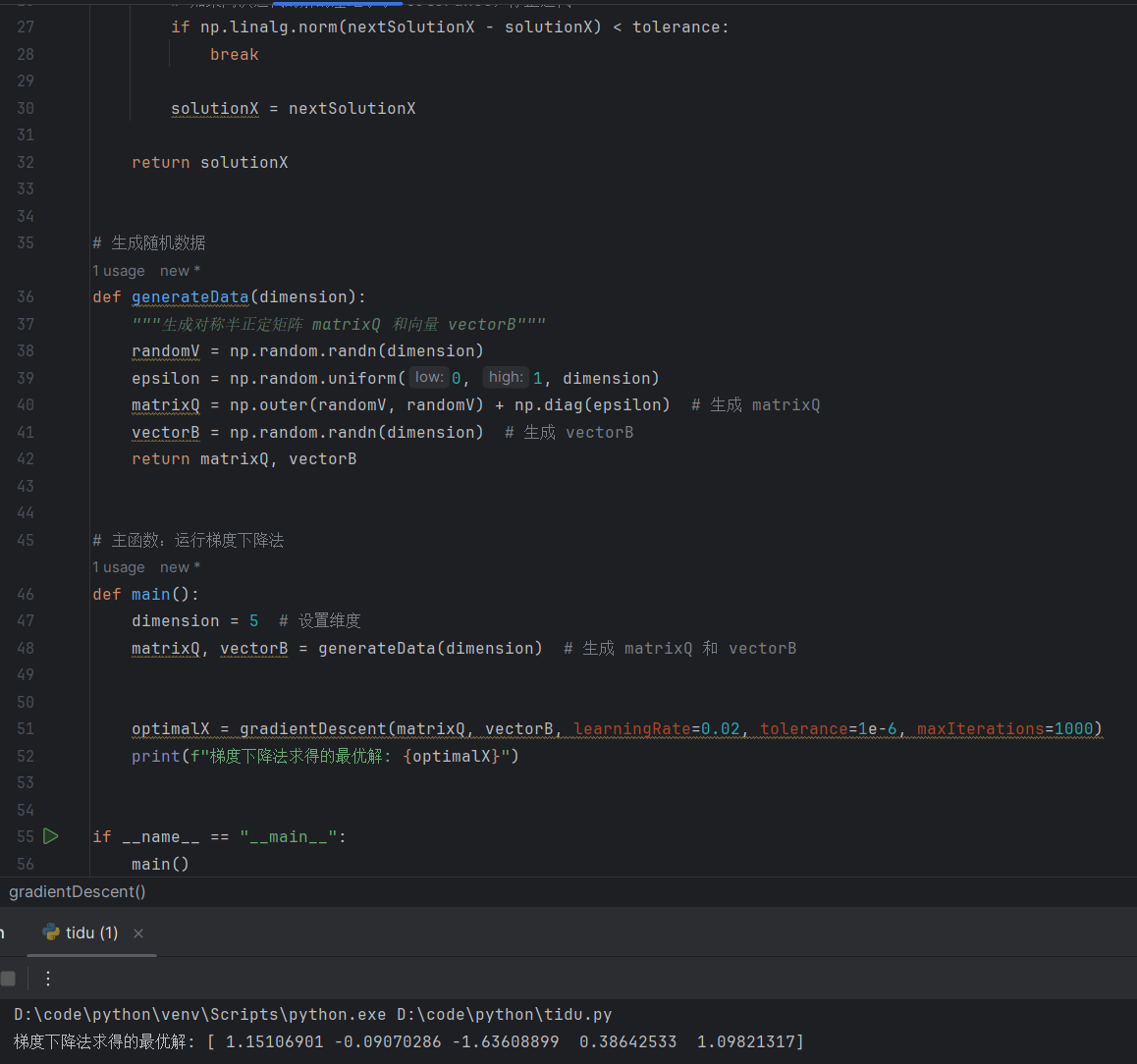
**3，4，5代码展示如下**

|  |
| --- |
| Python def gradientDescent(matrixQ, vectorB, learningRate=0.01, tolerance=1e-6, maxIterations=1000):   dimension = len(vectorB) # 获取维度  solutionX = np.random.randn(dimension) # 随机初始化 solutionX   for iteration in range(maxIterations):  gradient = np.dot(matrixQ, solutionX) + vectorB # 计算梯度 ∇f(solutionX) = Qx + b  nextSolutionX = solutionX - learningRate \* gradient # 梯度下降更新公式 solutionX\_{t+1} = solutionX\_t - learningRate \* ∇f(solutionX)   # 如果两次迭代的解的差距小于 tolerance，停止迭代  if np.linalg.norm(nextSolutionX - solutionX) < tolerance:  break   solutionX = nextSolutionX   return solutionX |

**主函数设计**

|  |
| --- |
| Python def main():  dimension = 5 # 设置维度  matrixQ, vectorB = generateData(dimension) # 生成 matrixQ 和 vectorB    optimalX = gradientDescent(matrixQ, vectorB, learningRate=0.01, tolerance=1e-6, maxIterations=1000)  print(f"梯度下降法求得的最优解: {optimalX}")   if \_\_name\_\_ == "\_\_main\_\_":  main() |

**5.输出结果展示**



**6思维拓展题目延伸**

**1.关于学习率.收敛容忍度和 最大迭代次数参数的确定**

如何确定 学习率(learningRate).收敛容忍度(tolerance)和 最大迭代次数(maxlterations)，可以遵循以下原则，但具体的最佳值往往需要通过实验来调整，因为它们高度依赖于问题的性质。

1.**学习率**控制了每次更新时步长的大小，过大或过小都会影响优化的效果。

过大的学习率:更新步长太大，可能会导致在搜索过程中跳过最优解，甚至让算法发散。

过小的学习率:更新步长太小，虽然可以保证不跳过最优解，但会导致收敛速度非常慢。

在高维问题中，学习率通常需要设定得较小，以避免步长过大。

**2.收敛容忍度**(tolerance)

容忍度定义了算法停止的条件，即当两个连续迭代结果的差距小于这个值时，认为已经收敛。

较小的容忍度(例如 1e-8):可以获得更精确的解，但可能会导致算法迭代时间过长。

较大的容忍度(例如 1e-4):收敛速度更快，但可能导致解的精度不高。

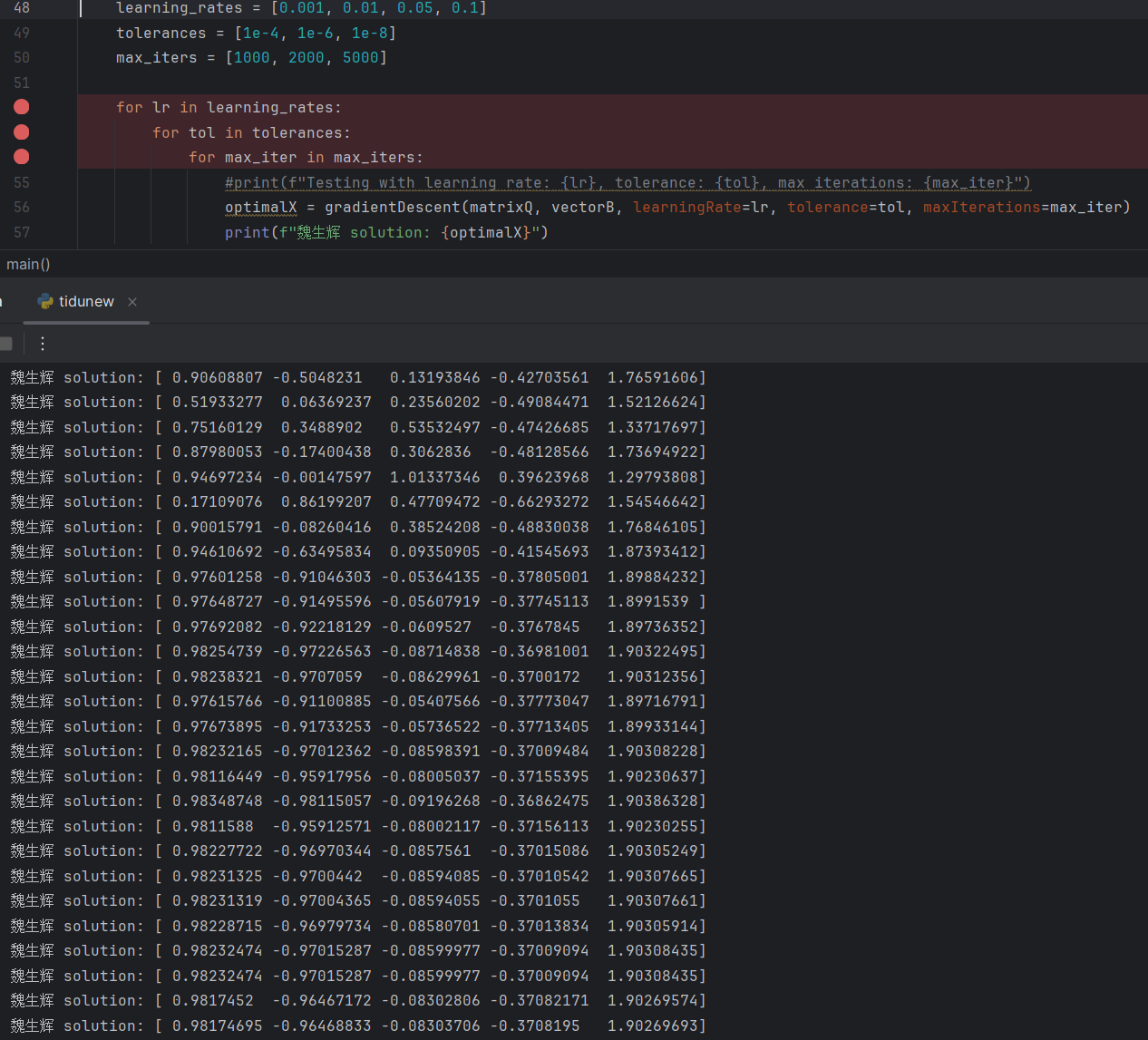
**3.最大迭代次数(maxlterations)**

最大迭代次数限制了算法执行的步数。如果在规定的迭代次数内没有收敛，算法将强制停止。这个参数主要用来避免算法陷入无穷迭代。

**2.不同参数下的结果展示**

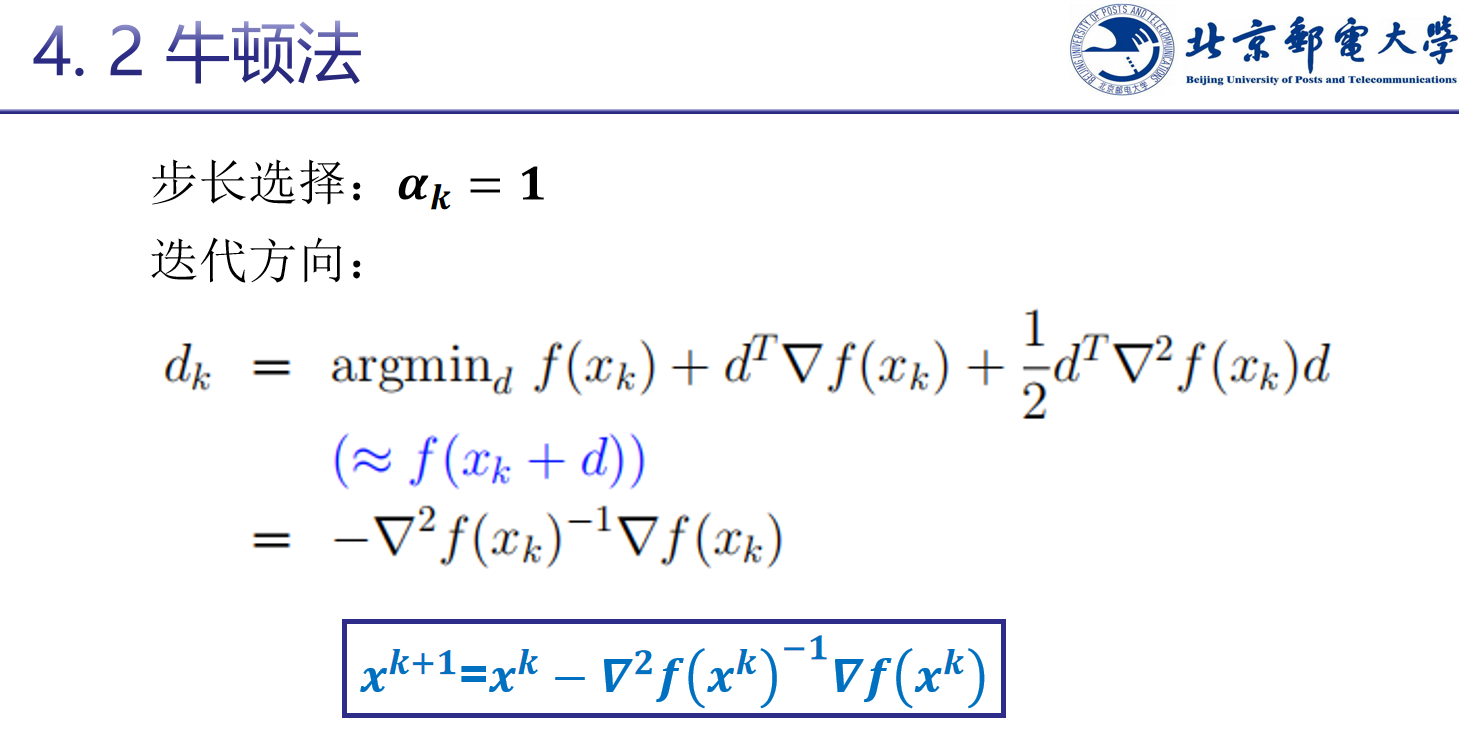
通过三个嵌套循环

|  |
| --- |
| Python learning\_rates = [0.001, 0.01, 0.05, 0.1] tolerances = [1e-4, 1e-6, 1e-8] max\_iters = [1000, 2000, 5000]  for lr in learning\_rates:  for tol in tolerances:  for max\_iter in max\_iters:  #print(f"Testing with learning rate: {lr}, tolerance: {tol}, max iterations: {max\_iter}")  optimalX = gradientDescent(matrixQ, vectorB, learningRate=lr, tolerance=tol, maxIterations=max\_iter)  print(f"魏生辉 solution: {optimalX}") |



**牛顿法**

**1明确牛顿法（根据ppt）**



**2. 步骤说明**

**步骤 1：生成矩阵 Q 和向量 b**

同梯度下降法。

**步骤 2：求解解析解**

由于我们知道牛顿法在这个问题上会一步得到最优解，因此我们直接使用公式：

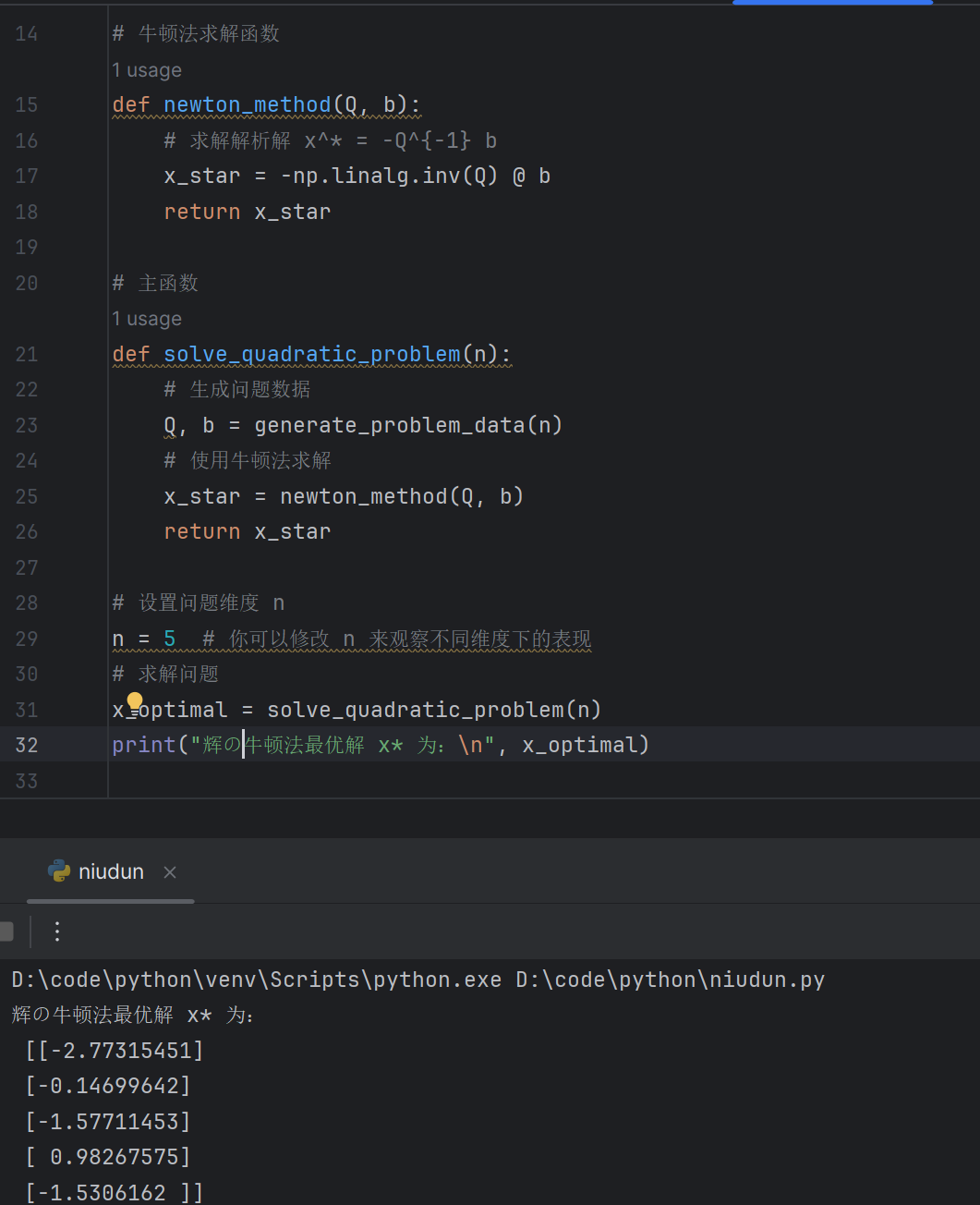
**步骤 3：终止条件**

在这种情形下，牛顿法无需迭代多次，只需要一步就能找到全局最优解，因此没有复杂的终止条件。

**3代码实现**

|  |
| --- |
| Python import numpy as np  # 生成矩阵 Q 和向量 b 的函数 def generate\_problem\_data(n):  # 生成随机向量 v 和 epsilon  v = np.random.normal(0, 1, (n, 1)) # 标准正态分布随机向量  epsilon = np.random.rand(n, 1) # [0, 1] 均匀分布随机向量  # 生成 Q = vv^T + diag(epsilon)  Q = v @ v.T + np.diagflat(epsilon)  # 生成 b  b = np.random.normal(0, 1, (n, 1)) # 标准正态分布随机向量  return Q, b  # 牛顿法求解函数 def newton\_method(Q, b):  # 求解解析解 x^\* = -Q^{-1} b  x\_star = -np.linalg.inv(Q) @ b  return x\_star  # 主函数 def solve(n):  # 生成问题数据  Q, b = generate\_problem\_data(n)  # 使用牛顿法求解  x\_star = newton\_method(Q, b)  return x\_star  # 设置问题维度 n n = 5 # ！！！！！！！可以修改 n 来观察不同维度下的表现 # 求解问题 x\_optimal = solve(n) print("辉の最优解 x\* 为：\n", x\_optimal) |

**4输出结果展示**



**Nelder-Mead单纯形法**

**1明确Nelder-Mead单纯形（根据ppt）**

fminsearch （Nelder-Mead单纯形法）

fminunc （拟牛顿方法）

**使用方法**：

[x, fval] = fminsearch/fminunc(fun, x0)

fun为输入的目标函数，x0为初始点

x为输出的局部极小值点，fval为局部极小值

**2. 步骤说明**

**步骤 1：定义目标函数**

在此问题中，目标函数上述提到过多次。

**步骤 2：初始单纯形**

**步骤 3：使用 SciPy 库的实现**

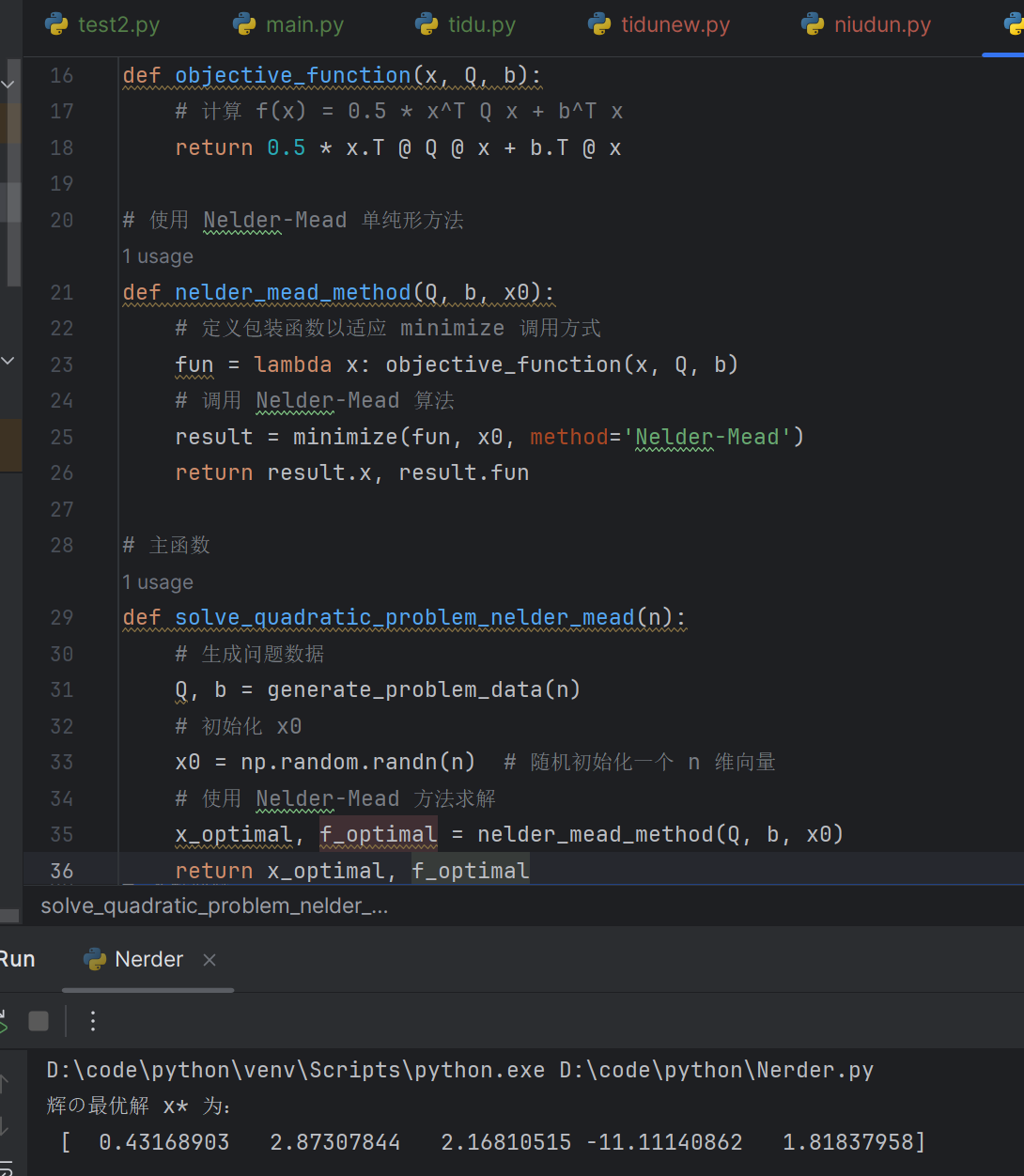
Nelder-Mead 方法已经集成在 Python 的 SciPy 库中。我们可以直接调用 scipy.optimize.minimize 函数并指定使用 Nelder-Mead 方法来求解问题。

**说明：**nelder\_mead\_method(Q, b, x0) 使用 SciPy 的 minimize 函数调用 Nelder-Mead 方法进行优化，初始点 x0x\_0x0 是一个随机生成的向量。

**3代码实现**

|  |
| --- |
| Python import numpy as np from scipy.optimize import minimize  # 生成矩阵 Q 和向量 b 的函数 def generate\_problem\_data(n):  # 生成随机向量 v 和 epsilon  v = np.random.normal(0, 1, (n, 1)) # 标准正态分布随机向量  epsilon = np.random.rand(n, 1) # [0, 1] 均匀分布随机向量  # 生成 Q = vv^T + diag(epsilon)  Q = v @ v.T + np.diagflat(epsilon)  # 生成 b  b = np.random.normal(0, 1, (n, 1)) # 标准正态分布随机向量  return Q, b  # 定义目标函数 def objective\_function(x, Q, b):  # 计算 f(x) = 0.5 \* x^T Q x + b^T x  return 0.5 \* x.T @ Q @ x + b.T @ x  # 使用 Nelder-Mead 单纯形方法 def nelder\_mead\_method(Q, b, x0):  # 定义包装函数以适应 minimize 调用方式  fun = lambda x: objective\_function(x, Q, b)  # ！！！！！！！！！！！！调用 Nelder-Mead 算法  result = minimize(fun, x0, method='Nelder-Mead')  return result.x, result.fun  # 主函数 def solve\_quadratic\_problem\_nelder\_mead(n):  # 生成问题数据  Q, b = generate\_problem\_data(n)  # 初始化 x0  x0 = np.random.randn(n) # 随机初始化一个 n 维向量  # 使用 Nelder-Mead 方法求解  x\_optimal, f\_optimal = nelder\_mead\_method(Q, b, x0)  return x\_optimal, f\_optimal  # 设置问题维度 n n = 5 # 你可以修改 n 来观察不同维度下的表现 # 求解问题 x\_optimal, f\_optimal = solve\_quadratic\_problem\_nelder\_mead(n) print("辉の最优解 x\* 为：\n", x\_optimal) |

**4输出结果展示**



**第二部分：比较三种方法在不同维度n下的速度**

**1基本思路（控制变量法！！！！）**

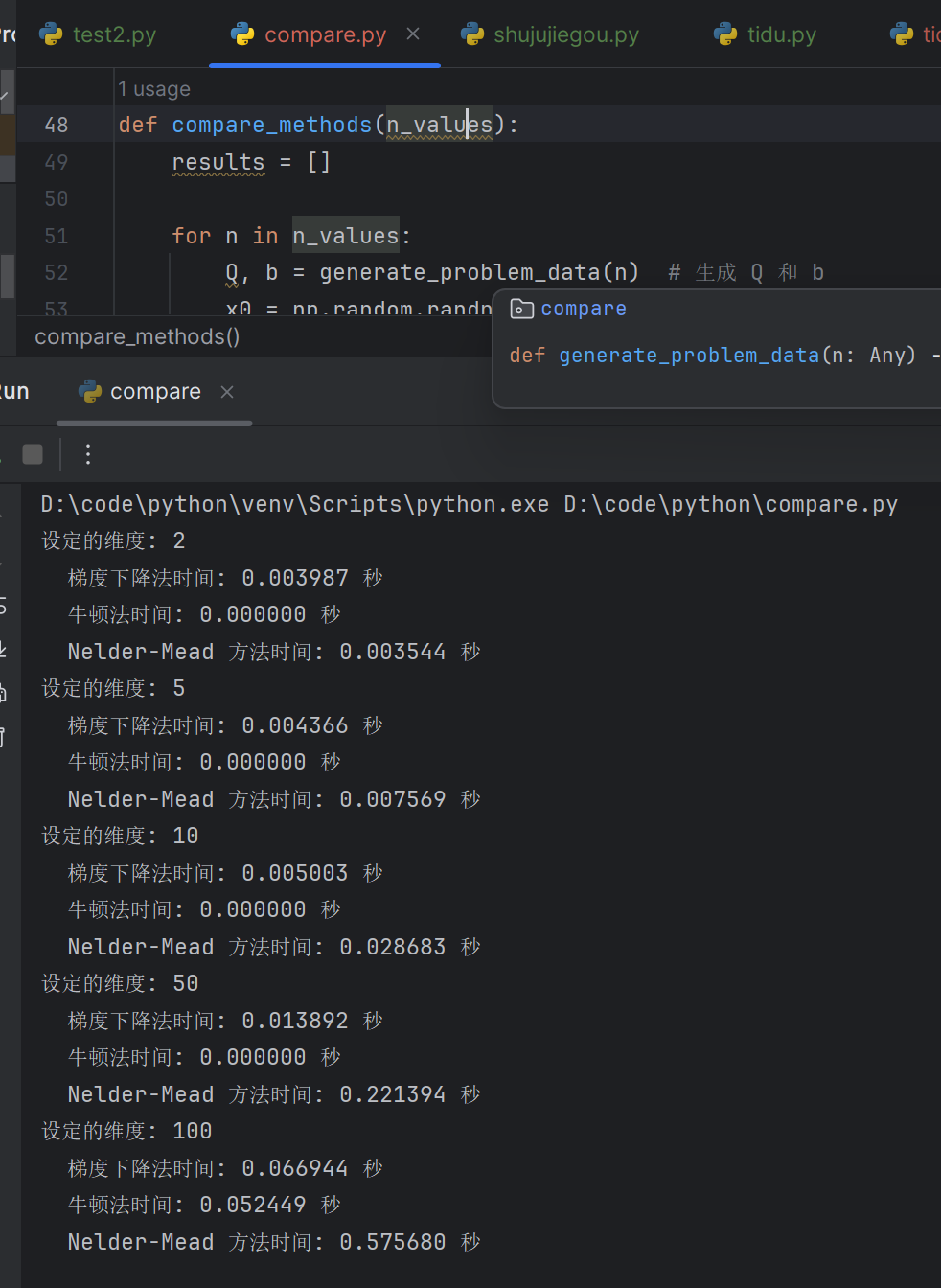
为了比较三种方法的速度，我们可以这样做：

1. **测试维度选择**：选定多个不同的维度 n（例如，n=2,5,10,50,100）
2. **运行时间测量**：使用 Python 中的 time 模块来测量每种算法的运行时间，记录从算法开始到结束所用的时间。
3. ***！*相同的优化问题**：确保对每个维度 **n** 生成相同的矩阵 **Q** 和向量 **b，**以保证三种算法优化的目标函数相同。
4. **结果记录与比较**：对于每个维度 n，记录三种算法的运行时间，并将其进行比较。

**2代码实现**

|  |
| --- |
| Python def compare\_methods(n\_values):  results = []   for n in n\_values:  Q, b = generate\_problem\_data(n) # 生成 Q 和 b  x0 = np.random.randn(n) # 随机初始化 x0   # 记录梯度下降法的时间  start\_time = time.time()  gradient\_descent(Q, b, x0)  gradient\_time = time.time() - start\_time   # 记录牛顿法的时间  start\_time = time.time()  newton\_method(Q, b)  newton\_time = time.time() - start\_time   # 记录 Nelder-Mead 方法的时间  start\_time = time.time()  nelder\_mead\_method(Q, b, x0)  nelder\_mead\_time = time.time() - start\_time |

**结果展示**



**结果分析和总结拓展**

**根据课上ppt预期的结果是：**

1. **牛顿法**在低维问题上非常快，因为它是解析解法，不需要迭代。在高维情况下，计算 Q−1 的时间成本会增加，导致运行时间上升。
2. **梯度下降法**在小维度下表现尚可，但随着维度增加，由于需要更多的迭代来收敛，时间消耗也会增加。
3. **Nelder-Mead 方法**作为无导数优化方法，收敛速度较慢，特别是在高维空间中，由于没有梯度信息，可能会表现出较大的时间开销。

**总结拓展**

通过比较三种方法在不同维度 n 下的运行时间，我们可以清楚地看到每种方法的特征：

* **牛顿法**：适合低维问题，并且在二次问题上具有非常快的收敛速度。
* **梯度下降法**：适用于大部分维度，但随着维度增大，迭代次数可能会显著增加。
* **Nelder-Mead 方法**：适合没有梯度信息的优化问题，但对于高维问题，速度较慢。