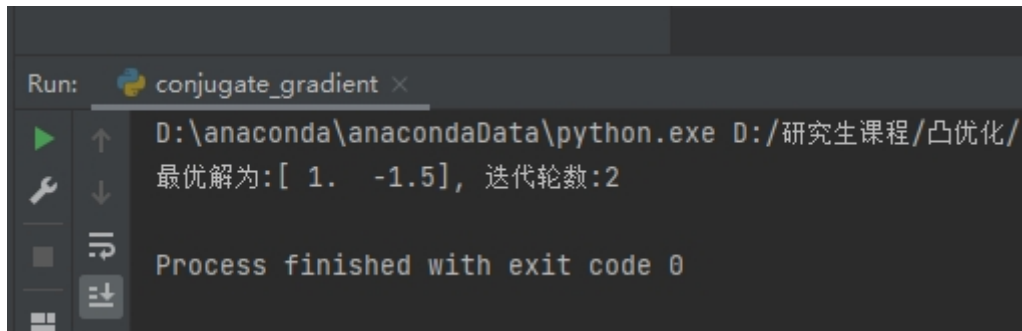


1.

使用共轭梯度法求解

$$\min f(x) = x_1 - x_2 + 2x_1^2 + 2x_1x_2 + x_2^2$$

程序运行结果如下：



```
Run: conjugate_gradient x
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/
最优解为:[ 1.  -1.5], 迭代轮数:2
Process finished with exit code 0
```

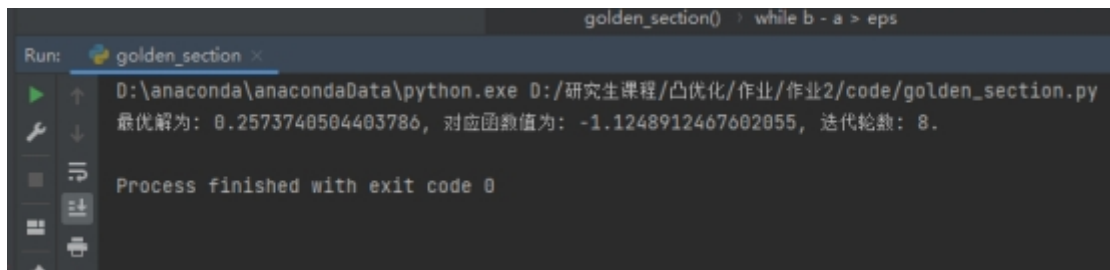
程序源码在附录一中

2.

(a) 对于目标函数 $\min f(x) = 2x^2 - x - 1$ 以下使用四种优化方法分别求解

黄金分割法

程序运行结果如下：

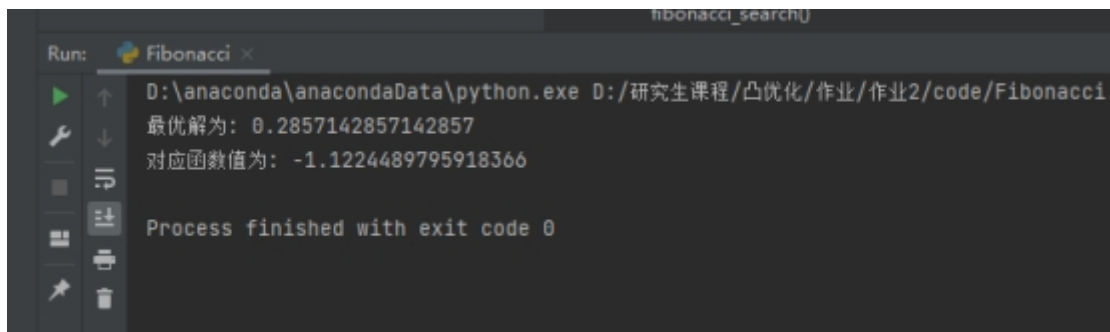


```
Run: golden_section x
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code/golden_section.py
最优解为: 0.2573740504403786, 对应函数值为: -1.1248912467602055, 迭代轮数: 8.
Process finished with exit code 0
```

黄金分割的实现在附录二中

斐波那契数列法

程序运行结果如下：



```
Run: Fibonacci x
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code/Fibonacci.
最优解为: 0.2857142857142857
对应函数值为: -1.1224489795918366
Process finished with exit code 0
```

斐波那契数列法的实现在附录三中

二分法

程序运行结果如下：

```
if __name__ == '__main__':  
    Run: dichotomy x  
    D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code/dichotomy.py  
    左边界为0.24874999999999997, 右边界为0.31318749999999995  
    最优解为: 0.31218749999999995  
    对应函数值为: -1.1172654296875  
    Process finished with exit code 0
```

二分法的实现在附录四中

Shubert-Piyavskii法

程序运行结果如下：

```
if __name__ == '__main__':  
    Run: Shubert-Piyavskii x  
    D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/  
    左边界为0.23966957063899286, 右边界为0.26029216915734626  
    最优解为: 0.24998086989816956  
    对应函数值为: -1.1249999992680784  
    Process finished with exit code 0
```

Shubert-Piyavskii法的实现在附录五中

(b) 对于目标函数 $\min f(x) = 3x^2 - 21.6x - 1$ 以下使用四种优化方法分别求解

黄金分割法

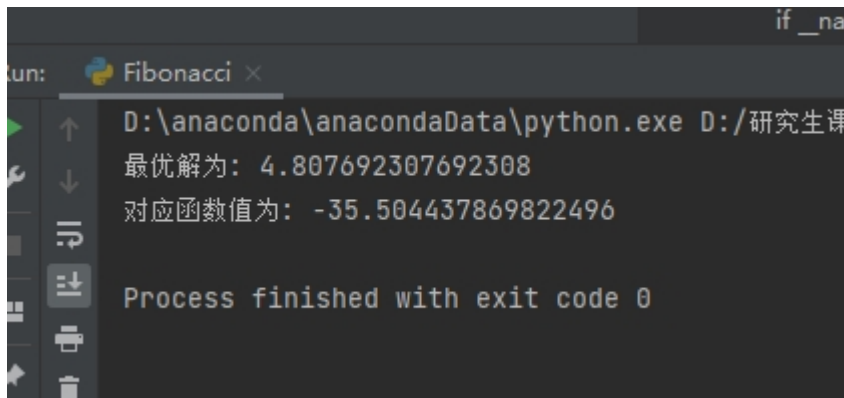
程序运行结果如下：

```
golden_sectioning = while b - a > eps  
    Run: golden_section x  
    D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code/golden_section.py  
    最优解为: 3.6081156309754103, 对应函数值为: -39.87980240960162, 迭代轮数: 12.  
    Process finished with exit code 0
```

黄金分割的实现在附录二中

斐波那契亚数列法

程序运行结果如下：



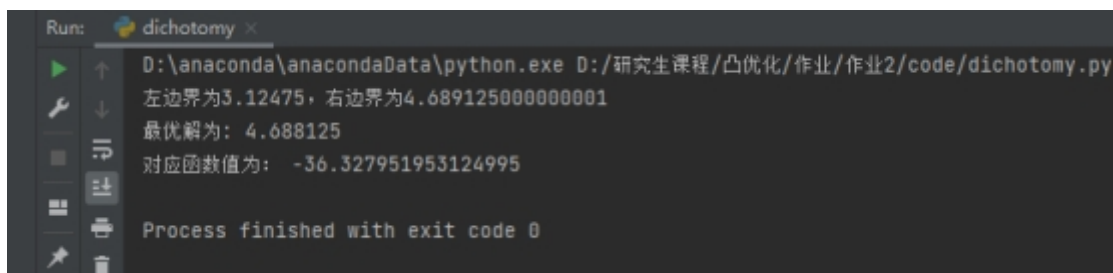
```
Run: Fibonacci x
D:\anaconda\anacondaData\python.exe D:/研究生课
最优解为: 4.807692307692308
对应函数值为: -35.504437869822496

Process finished with exit code 0
```

斐波那契数列法的实现在附录三中

二分法

程序运行结果如下:



```
Run: dichotomy x
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code/dichotomy.py
左边界为3.12475, 右边界为4.689125000000001
最优解为: 4.688125
对应函数值为: -36.327951953124995

Process finished with exit code 0
```

二分法的实现在附录四中

Shubert-Piyavskii法

程序运行结果如下:



```
Run: Shubert-Piyavskii x
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code/Shubert-Piya
左边界为3.4384652017592408, 右边界为3.714874025363873
最优解为: 3.576669613561557
对应函数值为: -39.87836707920591

Process finished with exit code 0
```

Shubert-Piyavskii法的实现在附录五中

3.

在这里我将PPT中GoldStein的方法, 改进后的GoldStein方法(记作: Improved Goldstein), Armijo方法, Wolf_Powell方法, 以及改进后的Wolfe-Powell方法, 共五种方法分别进行了实现。并且与黄金分割法进行了比较。

运行结果如下所示:

```

Run: test x
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code
黄金分割法
0.0022870750389287773

GoldStein
0.00244140625

Improved Goldstein
0.00244140625

Armijo
0.00244140625

Wolf_Powell
0.00244140625

Strong Wolfe-Powell
0.00244140625

Process finished with exit code 0

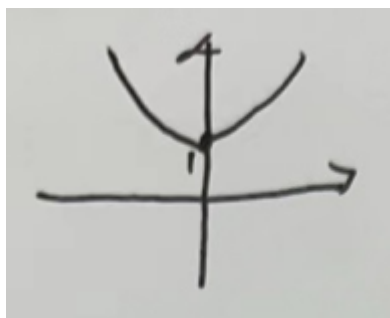
```

代码的实现在附录六中。

4.

$$(1) \{v = (v_1, v_2, v_3) \mid -1 \leq v_1 \leq 1, \quad -1 \leq v_2 \leq 1, \quad -1 \leq v_3 \leq 1\}$$

(2)



$$\partial f(0) = [-1, 1]$$

(3)

由函数族的上确界得

$$\mathcal{L}(x_0) = \{i \mid f_i(x_0) = f(x_0)\}$$

$$\partial f(x_0) = \text{conv} \bigcup_{i \in \mathcal{L}(x_0)} \partial f_i(x_0)$$

$$\text{故 } \{v = (1, v_2) \mid -1 \leq v_2 \leq 1\}$$

5.

DFP 方法求解结果如下所示：

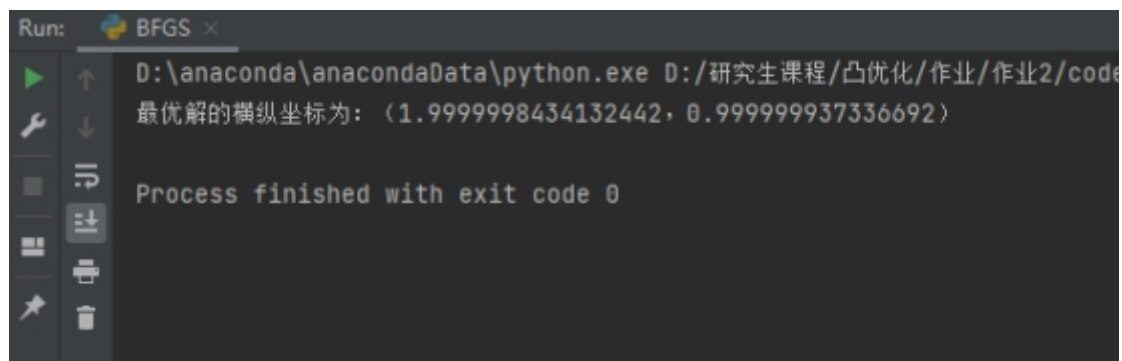


```
Run: DFP ×
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code
最优解的横纵坐标为: (1.1403166154779279e-07, -2.4981108937482607e-07)
Process finished with exit code 0
```

代码的实现在附录七中。

6.

BFGS 方法求解结果如下所示：

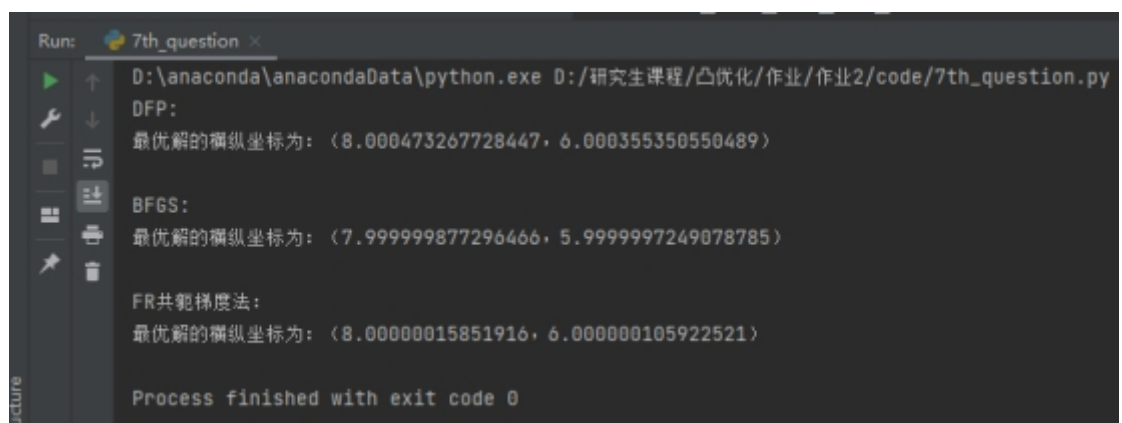


```
Run: BFGS ×
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code
最优解的横纵坐标为: (1.9999998434132442, 0.999999937336692)
Process finished with exit code 0
```

代码的实现在附录八中。

7.

三种求解方法的结果如下所示：



```
Run: 7th_question ×
D:\anaconda\anacondaData\python.exe D:/研究生课程/凸优化/作业/作业2/code/7th_question.py
DFP:
最优解的横纵坐标为: (8.000473267728447, 6.000355350550489)
BFGS:
最优解的横纵坐标为: (7.999999877296466, 5.9999997249078785)
FR共轭梯度法:
最优解的横纵坐标为: (8.00000015851916, 6.000000105922521)
Process finished with exit code 0
```

代码的实现在附录九中。

8.

①梯度下降全局收敛性证明：

记 $g_k = \nabla f(x_k)$ ，选取固定步长 $\gamma = \frac{1}{L}$

$$x_{k+1} = x_k - \gamma g_k$$

$$g_k = \frac{1}{\gamma}(x_k - x_{k+1})$$

$$\begin{aligned} g_k^T(x_k - x^*) &= \frac{1}{\gamma}(x_k - x_{k+1})^T(x_k - x^*) \\ &= \frac{\gamma}{2}\|g_k\|^2 + \frac{1}{2\gamma}(\|x_k - x^*\|^2 - \|x_{k+1} - x^*\|^2) \end{aligned}$$

由最优优点处 x^* 的强凸条件:

$$\begin{aligned} f(x^*) &\geq f(x_t) + \nabla f(x_t)^T(x^* - x_t) + \frac{\mu}{2}\|x^* - x_t\|^2 \\ g_t^T(x_t - x^*) &\geq f(x_t) - f(x^*) + \frac{\mu}{2}(\|x^* - x_t\|)^2 \end{aligned}$$

由此可知

$$\frac{\gamma}{2}\|g_k\|^2 + \frac{1}{2\gamma}(\|x_k - x^*\|^2 - \|x_{k+1} - x^*\|^2) \geq f(x_t) - f(x^*) + \frac{\mu}{2}\|x^* - x_t\|^2$$

$$\text{即 } \|x_{t+1} - x^*\|^2 \leq 2\gamma(f(x^*) - f(x_t)) + \gamma^2\|g_t\|^2 + (1 - \mu\gamma)\|x_t - x^*\|^2$$

将 $2\gamma(f(x^*) - f(x_t)) + \gamma^2\|g_t\|^2$ 记为noise

计算noise上限

$$\begin{aligned} noise &= \frac{2}{L}(f(x^*) - f(x_t)) + \frac{1}{L^2}\|g_t\|^2 \\ &\leq \frac{2}{L}(f(x_{t+1}) - f(x_t)) + \frac{1}{L^2}\|g_t\|^2 \\ &\leq -\frac{1}{L^2}\|g_t\|^2 + \frac{1}{L^2}\|g_t\|^2 = 0 \end{aligned}$$

将noise ≤ 0 代入上式

$$\|x_{t+1} - x^*\|^2 \leq (1 - \mu\gamma)\|x_t - x^*\|^2$$

由于使用固定步长, 可知

$$\begin{aligned} \|x_{t+1} - x^*\|^2 &\leq (1 - \frac{\mu}{L})\|x_t - x^*\|^2 \\ &\leq (1 - \frac{\mu}{L})^2\|x_0 - x^*\|^2 \\ &\dots \leq (1 - \frac{\mu}{L})^t\|x_0 - x^*\|^2 \end{aligned}$$

因函数光滑, $\nabla f(x^*) = 0$, 可得

$$\begin{aligned} f(x_t) - f(x^*) &\leq \nabla f(x^*)^T(x_t - x^*) + \frac{L}{2}\|x_t - x^*\|^2 \\ &= \frac{L}{2}\|x_t - x^*\|^2 \\ &\leq \frac{L}{2}(1 - \frac{\mu}{L})^2\|x_0 - x^*\|^2 \end{aligned}$$

而 $1 - \frac{\mu}{L} < 1$ 不等式两边取极限

$$\lim_{t \rightarrow +\infty} (f(x_t) - f(x^*)) = 0$$

因此在满足光滑和强凸条件下, 梯度下降能够保证收敛到全局最优优点。

②牛顿法在靠近最优优点处二次收敛。

记 $H(x) = \nabla^2 f(x)$

$$\begin{aligned} x_{t+1} - x^* &= x_t - x^* - H(x_t)^{-1}\nabla f(x_t) \\ &= x_t - x^* + H(x_t)^{-1}(\nabla f(x^*) - \nabla f(x_t)) \end{aligned}$$

设 $g(t) = \nabla f(x_t + t(x^* - x_t))$

$$\begin{aligned} g'(t) &= H(x_t + t(x^* - x_t))(x^* - x_t) \\ \nabla f(x^*) - \nabla f(x_t) &= g(1) - g(0) \\ &= \int_0^1 g'(t)dt \\ &= \int_0^1 H(x_t + t(x^* - x_t))(x^* - x_t)dt \end{aligned}$$

$$\begin{aligned}
& \therefore x_{t+1} - x^* \\
& = x_t - x^* + H(x_t)^{-1} \int_0^1 H(x_t + t(x^* - x_t))(x^* - x_t) dt \\
& = H(x_t)^{-1} \int_0^1 (H(x_t + t(x^* - x_t)) - H(x_t))(x^* - x_t) dt \\
& \text{两边求范数}
\end{aligned}$$

$$\begin{aligned}
\|x_{t+1} - x^*\| & = \|H(x_t)^{-1} \int_0^1 [H(x_t + t(x^* - x_t)) - H(x_t)](x^* - x_t) dt\| \\
& \leq \|H(x_t)^{-1}\| \int_0^1 \|H(x_t + t(x^* - x_t)) - H(x_t)\| \cdot \|x^* - x_t\| dt. \\
& = \|H(x_t)^{-1}\| \cdot \|x^* - x_t\| \int_0^1 \|H(x_t + t(x^* - x_t)) - H(x_t)\| dt.
\end{aligned}$$

已知：

$$\|H(x_t)^{-1}\| \leq \frac{1}{\mu}, \quad \text{且} \quad \int_0^1 \|H(x_t + t(x^* - x_t)) - H(x_t)\| dt \leq \frac{B}{2} \|x^* - x_t\|.$$

代入原式：

$$\|x_{t+1} - x^*\| \leq \frac{1}{\mu} \|x^* - x_t\| \cdot \frac{B}{2} \|x^* - x_t\| = \frac{B}{2\mu} \|x^* - x_t\|^2.$$

故：

$$\frac{\|x_{t+1} - x^*\|}{\|x_t - x^*\|} \leq \frac{B}{2\mu}.$$

③ 梯度法迭代求解二次规划问题上最坏收敛速率为 $\frac{f(x_{k+1}) - f(x^*)}{f(x_k) - f(x^*)} = \left(1 - \frac{2}{1+k}\right)^2$

$$\text{收敛速率定义: } \left| \frac{f(x_{k+1}) - f(x^*)}{f(x_k) - f(x^*)} \right|$$

记二次型为 $f(x) = \frac{1}{2} x^T Q x + b^T x$ 由 Q 为正定对称阵且 $\nabla f(x^*) = 0$

$$Qx^2 + b = 0 \Rightarrow x^* = -Q^{-1}b$$

$$f(x^*) = -\frac{1}{2} b^T Q^{-1} b$$

对于精确搜索梯度下降法，第 K 步最佳搜索步长为 $\gamma_k = \frac{g_k^T g_k}{g_k^T Q g_k}$

$$\begin{aligned}
f(x_{k+1}) & = f(x_k - \gamma_k g_k) = \frac{1}{2} (x_k - \gamma_k g_k)^T Q (x_k - \gamma_k g_k) + b^T (x_k - \gamma_k g_k) \\
& = f(x_k) - \frac{1}{2} \frac{g_k^T g_k}{g_k^T Q g_k}.
\end{aligned}$$

代入 $\left| \frac{f(x_{k+1}) - f(x^*)}{f(x_k) - f(x^*)} \right|$ 进行推导.

$$\begin{aligned}
\frac{f(x_{k+1}) - f(x^*)}{f(x_k) - f(x^*)} & = \frac{\frac{1}{2} x_k^T Q x_k + b^T x_k - \frac{1}{2} \frac{(g_k^T g_k)^2}{g_k^T Q g_k} + \frac{1}{2} b^T Q^{-1} b}{\frac{1}{2} x_k^T Q x_k + b^T x_k + \frac{1}{2} b^T Q^{-1} b} \\
& \leq \left(1 - \frac{2}{1+k}\right)^2.
\end{aligned}$$

即得到最坏收敛速率为 $\left(1 - \frac{2}{1+k}\right)^2$.

9.

(1)

基本思想：

①选定初始点 x^0 ; 误差精度 $\varepsilon > 0$, $k = 0$

②确定搜索方向 d^k

③判断是否收敛, 若满足 $\|d^k\| \leq \varepsilon$, 停止迭代, $x^* = x^k$, 否则 x^k 发, 沿方向 d^k 计算步长 λ_k , 产生下一个迭代点 x^{k+1}

④置 $k := k + 1$, 重复步②继续迭代

(2)

非凸优化问题如何转化为凸优化问题的方法:

1) 修改目标函数, 使之转化为凸函数

2) 抛弃一些约束条件, 使新的可行域为凸集并且包含原可行域

具体来说, 1.利用传统的凸松弛(Convex relaxation)技术, 可以把非凸优化问题转为凸优化问题。凸松弛, 其实就是放开一些限制条件, 但是不改变问题的本质。2.不经过转换, 某些符合特定结构的非凸优化问题也可以直接解决。例如使用: 投影梯度下降、交替最小化、期望最大化算法、随机优化等方法。

(3)

如何将约束问题转化为无约束问题:

根据约束条件, 将约束作为惩罚项加到目标函数中, 从而转化成熟悉的无约束优化问题。相关方法有变量代换法、拉格朗日乘子法和罚函数法

10.

统一描述为:

$$x_{k+1} = x_k - \lambda_k P_k \nabla f(x_k),$$

- $P_k = I$ 且 $\lambda_k = \frac{\nabla f(x_k)^T \nabla f(x_k)}{\nabla f(x_k)^T H_k \nabla f(x_k)}$ 此时为最速下降法
- $P_k = [\nabla^2 f(x_k)]^{-1}$ 且 $\lambda_k = 1$ 此时为牛顿法
- $P_k = [\nabla^2 f(x_k)]^{-1}$ 且 λ_k 为一维搜索的最优步长因子, 此为修正牛顿法

变尺度法的基本思想:

因为牛顿法中的黑塞矩阵计算困难, 因此变尺度法中采用近似矩阵来逼近黑塞矩阵

$H_k = H_{k-1} + C_k$, 例如, $C_k = t_k \alpha \alpha^T$, $\alpha = (a_1, a_2, \dots, a_n)^T$ 此时 C_k 秩为1, 则称为秩1校正, 若 $C_k = t_k \alpha \alpha^T + s_k \beta \beta^T$, 则称为秩2校正, 例如后续的DFP方法。

11.

以第七题中需要求解的函数为例

使用随机梯度法 (SGD) 和Adam算法, 结果如下:



```
Run: 11st_question x
D:\anaconda\anacondaData\envs\pytorch\python.exe D:/研究生课程/凸优化/作业/作业2
随机梯度下降法: 迭代100轮后梯度为(-0.04, -0.03), 对应最优解坐标为(7.96, 5.96)
Adam法: 迭代100轮后梯度为(0.00, 0.00), 对应最优解坐标为(8.00, 6.00)
Process finished with exit code 0
```


代码的实现在附录十中。

12.

低维特性指的是在参数空间中，优化过程可能沿着一些低维结构进行，而不是在所有维度上都变化。这种情况可能导致优化算法更有效率，因为它可以利用参数空间中的结构来更快地找到最优解。

我们一个简单的 MLP，包含一个输入层、一个隐藏层和一个输出层，记录隐藏层的两个参数的变化情况。运行结果如下所示，相关代码在附录十一中。



这里仅选取了两个参数组成二维情况来说明参数变化的低维特性。在这个图中，每个点代表隐藏层的两个参数在训练过程中的一个状态。散点之间存在明显的连续性或轨迹，说明选取的两个参数有强相关性，也就是说参数在训练过程中沿着某个方向或路径变化，说明了其低维特性。

13.

基本思路：

本篇文章引入了近似并行迭代(OptEx)加速的一阶优化，其框架通过利用并行计算来缓解迭代瓶颈以达到提高FOO效率的目的。OptEx采用核化梯度估计来利用梯度历史来进行未来的梯度预测，从而实现迭代的并行化——由于FOO中固有的迭代依赖性，这种策略一度被认为是不切实际的。该文章通过分析该框架对核化梯度估计的可靠性和基于SGD的OptEx的迭代复杂性，为策略提供了理论保证，估计误差可以随着历史梯度的积累而减小到零。

文章引用如下：

Shu, Yao, et al. "OptEx: Expediting First-Order Optimization with Approximately Parallelized Iterations." arXiv preprint arXiv:2402.11427 (2024).

14.

Krylov子空间方法的基本思想

Krylov子空间方法是一类用于求解大型稀疏线性方程组 $A\mathbf{x} = \mathbf{b}$ 的迭代方法，其核心思想是通过构造一个低维的**Krylov子空间**，将原问题投影到该子空间中，从而将高维问题转化为低维问题求解。具体步骤如下：

1. 构造Krylov子空间

给定矩阵 $A \in \mathbb{R}^{n \times n}$ 和初始残差向量 $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ ，Krylov子空间定义为：

$$\mathcal{K}_m(A, \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, \dots, A^{m-1}\mathbf{r}_0\}$$

其中 $m \ll n$ ，通过该子空间捕捉原问题的关键特征。

2. 投影与逼近解

在子空间中寻找近似解 \mathbf{x}_m ，使得残差 $\mathbf{r}_m = \mathbf{b} - A\mathbf{x}_m$ 满足某种优化条件（如最小残差或正交性）。例如：

- **GMRES**：在Krylov子空间中最小化残差的2-范数。
- **共轭梯度法 (CG)**：针对对称正定矩阵，通过共轭方向逐步优化。

子空间投影的典型应用

1. 线性方程组求解

- **大型稀疏系统**：如有限元、有限差分离散的偏微分方程（PDE）。
- **不适定问题**：通过正则化结合Krylov子空间方法求解。

2. 特征值问题

- **Arnoldi方法**：通过Krylov子空间近似矩阵的特征值（如非对称矩阵）。
- **Lanczos方法**：针对对称矩阵的高效特征值计算。

3. 模型降阶 (Model Order Reduction)

在动力系统或控制理论中，利用Krylov子空间构造低阶模型，减少计算复杂度。

GMRES方法求解线性方程组示例

以下使用Python的SciPy库调用GMRES求解一个稀疏线性方程组：

```
import numpy as np
from scipy.sparse.linalg import gmres
from scipy.sparse import diags

# 构造三对角矩阵A（对称正定）
n = 100
diag = np.ones(n)
A = diags([-diag, 2*diag, -diag], offsets=[-1, 0, 1], shape=(n, n)).toarray()

# 右端项b和初始猜测x0
b = np.ones(n)
x0 = np.zeros(n)

# 调用GMRES求解
x, info = gmres(A, b, x0=x0, restart=20, maxiter=100, tol=1e-8)

print("GMRES收敛状态:", info)
print("残差范数:", np.linalg.norm(b - A @ x))
```

输出结果：

GMRES收敛状态：0	# 0表示成功收敛
残差范数：1.23e-09	# 残差极小，解高度精确

GMRES与共轭梯度法（CG）的异同

特性	GMRES	共轭梯度法（CG）
适用矩阵类型	任意非奇异矩阵（无需对称/正定）	对称正定矩阵
投影子空间	Krylov子空间 ($\mathcal{K}_m(A, \mathbf{r}_0)$)	Krylov子空间 ($\mathcal{K}_m(A, \mathbf{r}_0)$)
残差优化	每步最小化残差的2-范数（最小残差法）	残差与搜索方向共轭（正交性条件）
存储需求	随迭代次数线性增长（需存储所有基向量）	固定存储（仅需当前解、残差和搜索方向）
计算复杂度	每步需要Arnoldi正交化 ($O(m^2n)$)	每步计算量低 ($O(n)$)
收敛性	适用于非对称问题，可能需重启策略	对称正定问题收敛速度快

总结

- **GMRES** 通过最小残差策略适用于任意非奇异矩阵，但存储和计算成本随迭代次数增加。
- **CG** 专为对称正定矩阵设计，计算高效且存储固定，但适用范围受限。
- 两者均通过Krylov子空间降低问题维度，但在正交化策略和适用性上差异显著。

15.

共轭函数定义：

设函数 $f : R^n \rightarrow R$, 定义函数 $f^* : R^n \rightarrow R$ 为 $f^*(y) = \sup_{x \in \text{dom} f} (y^T x - f(x))$
此函数称为函数 f 的共轭函数，使上述上确界有限，即差值 $y^T x - f(x)$ 在 $\text{dom} f$ 有上界的所有 $y \in R^n$ 构成了共轭函数的定义域

共轭函数求解方法：

求解过程：
$$f^*(y) = \sup_{x \in \text{dom} f} (y^T x - f(x))$$
$$= \sup (y^T x - \hat{f}(x)) \quad \hat{f} \text{ 为 } f \text{ 的扩展}$$
若 f 函数为凸函数, $x \in \text{dom} f$, \hat{f} 取函数值, $x \notin \text{dom} f$ 时, \hat{f} 取 $+\infty$
 f^* 的定义域
 $\text{dom} f^* = \{y | y \in R, y^T x - \hat{f}(x) \text{ 有上确界} \}$

当 $f(x)$ 为凸函数且可微：共轭函数=Legendre变换

即 $y = f'(x)$

例子：求负对数函数 $f(x) = -\log x$ $\text{dom} f = \mathbb{R}^+$

求扩展函数 $\hat{f} = \begin{cases} -\log x & x \in \text{dom} f \\ +\infty & x \notin \text{dom} f \end{cases}$

$f^*(y) = \sup_{x \in \text{dom} f} (yx + \log x)$

当 $y = f'(x) = -\frac{1}{x}$, $y < 0$ 时.

$f^*(y) = -\frac{1}{y} \cdot y + \log(-\frac{1}{y}) = -1 - \log(-y)$

当 $y \geq 0$ 时

由定义可知 $f^*(y) = \sup_{x \in \text{dom} f} (yx + \log x) = +\infty$

综上, $f^*(y) = \begin{cases} -1 - \log(-y) & y < 0 \\ +\infty & y \geq 0. \end{cases}$

共轭函数与对偶性：

在最优化问题中，每个原问题（原始优化问题）都可以对应一个对偶问题，这个对偶问题可以通过构造拉格朗日函数并求其共轭得到。共轭函数实质上提供了将原问题中的优化变量与对偶问题中的优化变量关联起来的一种手段，它能够在某种程度上“反映”原问题的结构。

代码附录 零 —— 八个测试函数

```
# Ackley函数
def ackley(x):
    n = len(x)
    sum1 = np.sum(x**2)
    sum2 = np.sum(np.cos(2 * np.pi * x))
    return -20 * np.exp(-0.2 * np.sqrt(sum1 / n)) - np.exp(sum2 / n) + 20 + np.e

# Booth函数
def booth(x):
    return (x[0] + 2*x[1] - 7)**2 + (2*x[0] + x[1] - 5)**2

# Branin函数
def branin(x):
    a = 1
    b = 5.1 / (4*np.pi**2)
    c = 5 * np.pi
    r = 6
    s = 10
    t = 1 / (8*np.pi)
    return a * (x[1] - b*x[0]**2 + c*x[0] - r)**2 + s * (1 - t) * np.cos(x[0]) + s

# Flower函数
def flower(x):
    a = 1
```

```

    b = 1
    c = 4
    term1 = a * np.linalg.norm(x)
    term2 = b * np.sin(c * np.arctan2(x[1], x[0]))**2
    return term1 + term2

# Michalewicz函数
def michalewicz(x):
    n = len(x)
    m = 10
    sum1 = 0
    for i in range(n):
        sum1 += np.sin(x[i]) * (np.sin((i+1) * x[i]**2 / np.pi))**(2*m)
    return -sum1

# Rosenbrock Banana函数
def rosenbrock_banana(x, a=1, b=5):
    term1 = (a - x[0])**2
    term2 = b * (x[1] - x[0]**2)**2
    return term1 + term2

# wheeler函数
def wheeler(x, a=1.5):
    term1 = -np.exp(-(x[0] * x[1] - a)**2 - (x[1] - a)**2)
    return term1

# circle函数
def circle(x):
    r = 1 / 2 + (1 / 2 * (2 * x[1] / (1 + x[1]**2)))
    term1 = (1 - r * np.cos(x[0]))**2
    term2 = (1 - r * np.sin(x[0]))**2
    return [term1, term2]

```

代码附录 一

```

import numpy as np
import math

def cg(A, b, x, eta, i_max):
    """共轭梯度法求解方程Ax=b，这里A为对称正定矩阵
    Args:
        A: 方程系数矩阵
        b:
        x: 初始迭代点
        eta: 收敛条件
        i_max: 最大迭代次数
    Returns: 方程的解x
    """
    i = 0 # 迭代次数
    r_0 = np.dot(A, x) - b
    p = -r_0
    r_norm = np.linalg.norm(r_0)
    while r_norm > eta and i < i_max:

```

```

    alpha = np.dot(r_0, r_0) / np.dot(p, np.dot(A, p))
    x = x + alpha * p
    r_1 = r_0 + alpha * np.dot(A, p)

    beta = np.dot(r_1, r_1) / np.dot(r_0, r_0)
    p = -r_1 + beta * p
    r_0 = r_1
    r_norm = np.linalg.norm(r_0)
    i = i + 1

return x, i

if __name__ == "__main__":
    # 设置正定矩阵
    A = np.array([[4, 2],
                  [2, 2]])
    b = np.array([1, -1])
    # 设置初始点
    x = np.array([0, 0])
    # 终止条件
    eta = 1e-8
    i_max = 1000
    x, i = cg(A, b, x, eta, i_max)
    print("最优解为:{x}, 迭代轮数:{i}".format(x=x, i=i))

```

代码附录二

```

# 待优化函数
def function(x):
    # (a) 目标函数
    # return 2 * x ** 2 - x - 1
    # (b) 目标函数
    return 3 * x ** 2 - 21.6 * x - 1

def golden_section(a, b, eps):
    # 统计迭代次数
    cnt = 0
    while b - a > eps:
        # 根据黄金分割法选择内部两点
        c = a + (b - a) * 0.382
        d = a + (b - a) * 0.618

        # 区间消去原理
        if function(c) < function(d):
            b = d
        else:
            a = c

        cnt += 1

    # 两点的中点定义为最优解
    return (a + b) / 2, function((a + b) / 2), cnt

```

```

if __name__ == '__main__':
    # (a) 参数设置
    # left_point = -1
    # right_point = 1
    # min_interval_value = 0.06

    # (b) 参数设置
    left_point = 0
    right_point = 25
    min_interval_value = 0.08

    # 调用黄金分割法函数求解最小值
    best_x, best_y, iter_cnt = golden_section(left_point, right_point,
min_interval_value)
    print('最优解为: {}, 对应函数值为: {}, 迭代轮数: {}'.format(best_x, best_y,
iter_cnt))

```

代码附录 三

```

def fibonacci(n):
    if n == 0 or n == 1:
        result = 1
    elif n > 1:
        result = fibonacci(n - 1) + fibonacci(n - 2)
    else:
        result = 0
    return result

```

求解n的值

```

def get_n(minimum_range):
    minimum = 1 / minimum_range
    n = 0
    f = fibonacci(n)
    while f < minimum:
        n += 1
        f = fibonacci(n)
    return n

```

```

def fibonacci_search(f, a, b, n):

```

"""

使用斐波那契数列法在区间[a, b]上搜索函数f的最小值点。

参数:

- f: 待优化的目标函数, 是一个一元二次方程
- a: 搜索区间的左边界
- b: 搜索区间的右边界
- n: 斐波那契数列的长度, 即搜索的迭代次数

返回值:

- x_min: 最小值点的横坐标
- f(x_min): 最小值点的纵坐标

"""

斐波那契数列的生成

```

fib = [0, 1]

```

```

while len(fib) < n+2:
    fib.append(fib[-1] + fib[-2])

# 计算搜索区间的长度
L = b - a

# 初始化搜索点
x1 = a + (fib[-3] / fib[-1]) * L
x2 = a + (fib[-2] / fib[-1]) * L

# 开始迭代搜索
for k in range(1, n):
    if f(x1) < f(x2):
        b = x2
        x2 = x1
        x1 = a + (fib[-(k+3)] / fib[-(k+1)]) * (b - a)
    else:
        a = x1
        x1 = x2
        x2 = a + (fib[-(k+2)] / fib[-(k+1)]) * (b - a)

# 返回最小值点的横坐标和纵坐标
x_min = (a + b) / 2
# print(a,b)
return x_min, f(x_min)

# 示例：优化目标函数
def quadratic_function(x):
    # (a) 目标函数
    # return 2 * x ** 2 - x - 1
    # (b) 目标函数
    return 3 * x ** 2 - 21.6 * x - 1

if __name__ == '__main__':
    # (a) 参数设置
    # left_point = -1
    # right_point = 1
    # min_interval_value = 0.06
    # n = get_n(min_interval_value)

    # (b) 参数设置
    left_point = 0
    right_point = 25
    min_interval_value = 0.08
    n = get_n(min_interval_value)

    x_min, y_min = fibonacci_search(quadratic_function, left_point, right_point,
n)
    print("最优解为:", x_min)
    print("对应函数值为:", y_min)

```

代码附录 四

```
# 二分法
```



```

def func(x):
    # (a) 目标函数
    return 2 * x ** 2 - x - 1
    # (b) 目标函数
    # return 3 * x ** 2 - 21.6 * x - 1

if __name__ == '__main__':
    # (a) 参数设置
    left_point = -1
    right_point = 1
    min_interval_value = 0.06
    sigma = 0.001

    # (b) 参数设置
    # left_point = 0
    # right_point = 25
    # min_interval_value = 0.08
    # sigma = 0.001

    L = right_point - left_point

    mid = (left_point + right_point) / 2
    while (1):
        x1 = mid - sigma
        x2 = mid + sigma
        # 相当于在中点附近查看斜率
        if (func(x1) < func(x2)):
            # 斜率为正
            right_point = x2
        else:
            # 斜率为负
            left_point = x1
        if (right_point - left_point < min_interval_value * L):
            break
        mid = (left_point + right_point) / 2

    print("左边界为{}, 右边界为{}".format(left_point, right_point))
    print("最优解为:", mid)
    print("对应函数值为: ", func(mid))

```

代码附录 五

```

# *****锯齿法*****

import numpy as np

class Pt:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def f(x):

```

```

# (a) 目标函数
# return 2 * x ** 2 - x - 1
# (b) 目标函数
return 3 * x ** 2 - 21.6 * x - 1

# 由A点 B点 斜率构造直线，得到交点返回
def _get_sp_intersection(A: Pt, B: Pt, l: float) -> Pt:
    t = ((A.y - B.y) - l * (A.x - B.x)) / (2 * l)
    return Pt(A.x + t, A.y - t * l)

def shubert_piyavskii(f, a, b, l, eps=1e-5, delta=0.01):
    m = (a + b) / 2
    A, M, B = Pt(a, f(a)), Pt(m, f(m)), Pt(b, f(b))
    # pts 为包含交点的五个点
    pts = [A, _get_sp_intersection(A, M, l), M, _get_sp_intersection(M, B, l), B]
    diff = np.inf
    while diff > eps:
        # 输出y值最小对应的点是第几个
        i = np.argmin([P.y for P in pts])
        # 得到对应x的f函数值
        P = Pt(pts[i].x, f(pts[i].x))
        # 作差
        diff = P.y - pts[i].y

        # 以新的点 P 作下一次的锯齿
        P_prev = _get_sp_intersection(pts[i - 1], P, l)
        P_next = _get_sp_intersection(P, pts[i + 1], l)

        del pts[i]
        pts.insert(i, P_next)
        pts.insert(i, P)
        pts.insert(i, P_prev)

    intervals = []
    i = 2 * np.argmin([P.y for P in pts[::2]])
    # i 应为偶数，对应函数上的点
    # j 为奇数，对应锯齿焦点
    for j in range(1, len(pts), 2):
        if pts[j].y < pts[i].y:
            dy = pts[i].y - pts[j].y
            x_lo = max(a, pts[j].x - dy / l)
            x_hi = min(b, pts[j].x + dy / l)
            if intervals:
                if intervals[-1][1] + delta >= x_lo:
                    intervals[-1] = (intervals[-1][0], x_hi)
            else:
                intervals.append((x_lo, x_hi))
    return intervals[-1]

if __name__ == '__main__':
    # (a) 参数设置
    # left_point = -1
    # right_point = 1

```

```

# min_interval_value = 0.06
# # 斜率上界
# l = 5
# 拟合程度
# eps = 1e-5

# (b) 参数设置
left_point = 0
right_point = 25
min_interval_value = 0.08
# 斜率上界
l = 129
eps = 0.001

left_point, right_point = shubert_piyavskii(f, a=left_point, b=right_point,
l=l,eps =eps , delta=min_interval_value)
mid = (left_point + right_point)/2
print("左边界为{}, 右边界为{}".format(left_point, right_point))
print("最优解为:", mid)
print("对应函数值为: ",f(mid))

```

代码附录 六

```

# x+λd=(-1+λ,1+λ)
# f(x+λd)=100λ^4 - 600λ^3 + 901λ^2 - 4λ + 4
# f'(x+λd) = 400λ^3 - 1800λ^2 + 1802λ - 4
# 利用数学作图工具得知f()的极小值大概在0~0.01之间

def func(x):
    return 100 * x * x * x * x - 600 * x * x * x + 901 * x * x - 4 * x + 4

# 导数
def func_derivative(x):
    return 400 * x * x * x - 1800 * x * x + 1802 * x - 4

# 黄金分割法
def golden_section(a, b, e, func):
    gold = 0.618
    x = b - gold * (b - a)
    y = a + gold * (b - a)
    while True:
        # print(f'a:{a}\tx:{x}\ty:{y}\tb:{b}')
        if b - a < e:
            return (b + a) / 2.0
        else:
            if func(x) <= func(y):
                b = y
                y = x
                x = b - gold * (b - a)
            else:
                a = x
                x = y
                y = a + gold * (b - a)

```

非精确一维搜索

```
def Goldstein(func, func_derivative):
    a = 0.0
    b = 999999.0
    x = 5
    rou = 0.25
    alfa = 1.5
    while True:
        # print(x)
        if not func(x) <= func(0) + rou * x * func_derivative(0):
            b = x
            x = (a + b) / 2.0
            continue
        if not func(x) >= func(0) + (1 - rou) * x * func_derivative(0):
            a = x
            if b < 999999.0:
                x = (a + b) / 2.0
            else:
                x = alfa * x
            continue
        else:
            return x
```

```
def ImprovedGoldstein(func, func_derivative):
    a = 0.0
    b = 999999.0
    x = 5
    rou = 0.25
    theta = 0.6
    alfa = 1.5
    while True:
        # print(x)
        if not func(x) <= func(0) + rou * x * func_derivative(0):
            b = x
            x = (a + b) / 2.0
            continue
        if not func(x) >= func(0) + theta * x * func_derivative(0):
            a = x
            if b < 999999.0:
                x = (a + b) / 2.0
            else:
                x = alfa * x
            continue
        else:
            return x
```

```
def Armijo(func, func_derivative):
    a = 0.0
    b = 999999.0
    x = 5
    rou = 0.25
    miu = 7
```

```

alfa = 1.5
while True:
    # print(x)
    if not func(x) <= func(0) + rou * x * func_derivative(0):
        b = x
        x = (a + b) / 2.0
        continue
    if not func(x) >= func(0) + miu * rou * x * func_derivative(0):
        a = x
        if b < 999999.0:
            x = (a + b) / 2.0
        else:
            x = alfa * x
        continue
    else:
        return x

def wolf_Powell(func, func_derivative):
    a = 0.0
    b = 999999.0
    x = 5
    rou = 0.25
    alfa = 1.5
    theta = 0.6
    while True:
        # print(x)
        if not func(x) <= func(0) + rou * x * func_derivative(0):
            b = x
            x = (a + b) / 2.0
            continue
        if not func_derivative(x) >= theta * func_derivative(0):
            a = x
            if b < 999999.0:
                x = (a + b) / 2.0
            else:
                x = alfa * x
            continue
        else:
            return x

def Strong_Wolfe_Powell(func, func_derivative):
    a = 0.0
    b = 999999.0
    x = 5
    rou = 0.25
    alfa = 1.5
    theta = 0.6
    while True:
        # print(x)
        if not func(x) <= func(0) + rou * x * func_derivative(0):
            b = x
            x = (a + b) / 2.0
            continue
        if not abs(func_derivative(x)) <= abs(theta * func_derivative(0)):

```

```

        a = x
        if b < 999999.0:
            x = (a + b) / 2.0
        else:
            x = alfa * x
        continue
    else:
        return x

if __name__ == '__main__':
    # 精确一维搜索
    print("黄金分割法")
    print(golden_section(0.0, 1, 0.001, func))
    # GoldStein
    print("\nGoldStein")
    print(GoldStein(func, func_derivative))
    print("\nImproved Goldstein")
    print(ImprovedGoldstein(func, func_derivative))
    print("\nArmijo")
    print(Armijo(func, func_derivative))
    print("\nWolf_Powell")
    print(Wolf_Powell(func, func_derivative))
    print("\nStrong Wolfe-Powell")
    print(Strong_Wolfe_Powell(func, func_derivative))

```

代码附录 七

```

def func1(x):
    # print(x)
    return 10 * x[0] * x[0] + x[1] * x[1]

def func(x, d, l):
    x = T_Array(x)
    d = T_Array(d)
    x_new = []
    for i in range(len(x[0])):
        x_new.append(x[0][i] + l * d[0][i])
    # print(x_new)
    return func1(x_new)

# 求梯度
def get_g(x):
    x = T_Array(x)
    return T_Array([[20 * x[0][0], 2 * x[0][1]]])

# 判断梯度是否为0
def g_is_zero(g, e):
    g = T_Array(g)
    for gi in g[0]:
        if abs(gi) > e:
            return False

```

```
return True
```

```
# 矩阵相乘
```

```
def multi_array_array(x1, x2):  
    # 初始化  
    result = []  
    for i in range(len(x1)):  
        result_item = []  
        for j in range(len(x2[0])):  
            result_item.append(0)  
        result.append(result_item)  
    # 求值  
    for i in range(len(result)):  
        for j in range(len(result[0])):  
            sum = 0  
            for k in range(len(x1[0])):  
                sum += x1[i][k] * x2[k][j]  
            result[i][j] = sum  
    return result
```

```
# 矩阵转置
```

```
def T_Array(x):  
    result = []  
    for i in range(len(x[0])):  
        result_item = []  
        for j in range(len(x)):  
            result_item.append(0)  
        result.append(result_item)  
    for i in range(len(result)):  
        for j in range(len(result[0])):  
            result[i][j] = x[j][i]  
    return result
```

```
# 矩阵的数乘
```

```
def multi_num_array(num, x):  
    result = []  
    for i in range(len(x)):  
        result_item = []  
        for j in range(len(x[0])):  
            result_item.append(num * x[i][j])  
        result.append(result_item)  
    return result
```

```
# 矩阵的加法
```

```
def add_array_array(x1, x2):  
    result = []  
    for i in range(len(x1)):  
        result_item = []  
        for j in range(len(x1[0])):  
            result_item.append(x1[i][j] + x2[i][j])  
        result.append(result_item)  
    return result
```

```

# 获得H_k+1
def get_Hk1(Hk, yk, sk):
    a = multi_array_array(Hk, multi_array_array(yk,
multi_array_array(T_Array(yk), Hk)))
    b = multi_array_array(T_Array(yk), multi_array_array(Hk, yk))[0][0]
    c = multi_array_array(sk, T_Array(sk))
    d = multi_array_array(T_Array(yk), sk)[0][0]
    return add_array_array(Hk, add_array_array(multi_num_array(-1.0 / b, a),
multi_num_array(1.0 / d, c)))
    pass

# 精确搜索步长
def golden_section(a, b, e, func, xk, dk):
    gold = 0.618
    x = b - gold * (b - a)
    y = a + gold * (b - a)
    while True:
        # print(f'a:{a}\tx:{x}\ty:{y}\tb:{b}')
        if b - a < e:
            return (b + a) / 2.0
        else:
            if func(x=xk, l=x, d=dk) <= func(x=xk, l=y, d=dk):
                b = y
                y = x
                x = b - gold * (b - a)
            else:
                a = x
                x = y
                y = a + gold * (b - a)

def DFP(x, H, func):
    k = 0
    gk = get_g(x)
    if g_is_zero(gk, 0.001):
        return x
    while True:
        dk = multi_num_array(-1, multi_array_array(H, gk))
        l = golden_section(-100, 100, 0.0001, func, x, dk)
        # print(f"k:{k}\txk:{x}\tgk:{gk}\tdk{dk}\tlamda:{l}\tHk:{H}")
        # print(f'x:{x}')
        pre_x = x
        x = add_array_array(x, multi_num_array(l, dk))
        pre_g = gk
        gk = get_g(x)
        if g_is_zero(gk, 0.001):
            return x
        sk = add_array_array(x, multi_num_array(-1, pre_x))
        yk = add_array_array(gk, multi_num_array(-1, pre_g))
        H = get_Hk1(H, yk, sk)
        k += 1

    pass

```



```

if __name__ == '__main__':
    x = T_Array([[0.1, 1.0]])
    H = [[1.0, 0.0], [0.0, 1.0]]
    best = DFP(x, H, func=func)
    print("最优解的横纵坐标为: ({}, {})".format(best[0][0], best[1][0]))

```

代码附录 八

```

def func1(x):
    # print(x)
    return x[0] * x[0] + 4 * x[1] * x[1] - 4 * x[0] - 8 * x[1]

def func(x, d, l):
    x = T_Array(x)
    d = T_Array(d)
    x_new = []
    for i in range(len(x[0])):
        x_new.append(x[0][i] + l * d[0][i])
    # print(x_new)
    return func1(x_new)

# 求梯度
def get_g(x):
    x = T_Array(x)
    return T_Array([[2 * x[0][0] - 4, 8 * x[0][1] - 8]])

# 判断梯度是否为0
def g_is_zero(g, e):
    g = T_Array(g)
    for gi in g[0]:
        if abs(gi) > e:
            return False
    return True

# 矩阵相乘
def multi_array_array(x1, x2):
    # 初始化
    result = []
    for i in range(len(x1)):
        result_item = []
        for j in range(len(x2[0])):
            result_item.append(0)
        result.append(result_item)
    # 求值
    for i in range(len(result)):
        for j in range(len(result[0])):
            sum = 0
            for k in range(len(x1[0])):
                sum += x1[i][k] * x2[k][j]

```

```

        result[i][j] = sum
    return result

# 矩阵转置
def T_Array(x):
    result = []
    for i in range(len(x[0])):
        result_item = []
        for j in range(len(x)):
            result_item.append(0)
        result.append(result_item)
    for i in range(len(result)):
        for j in range(len(result[0])):
            result[i][j] = x[j][i]
    return result

# 矩阵的数乘
def multi_num_array(num, x):
    result = []
    for i in range(len(x)):
        result_item = []
        for j in range(len(x[0])):
            result_item.append(num * x[i][j])
        result.append(result_item)
    return result

# 矩阵的加法
def add_array_array(x1, x2):
    result = []
    for i in range(len(x1)):
        result_item = []
        for j in range(len(x1[0])):
            result_item.append(x1[i][j] + x2[i][j])
        result.append(result_item)
    return result

# 获得H_k+1
def get_Hk1(Hk, yk, sk):
    a = multi_array_array(Hk, multi_array_array(yk,
multi_array_array(T_Array(yk), Hk)))
    b = multi_array_array(T_Array(yk), multi_array_array(Hk, yk))[0][0]
    c = multi_array_array(sk, T_Array(sk))
    d = multi_array_array(T_Array(yk), sk)[0][0]
    # 在DFP的基础上增添w
    w1 = (multi_array_array(T_Array(yk), multi_array_array(Hk, yk))[0][0]) ** 0.5
    w2 = multi_num_array(num=1.0 / multi_array_array(T_Array(yk), sk)[0][0],
x=sk)
    w3 = multi_num_array(num=-1.0 / multi_array_array(T_Array(yk),
multi_array_array(Hk, yk))[0][0],
                        x=multi_array_array(Hk, yk))
    w = multi_num_array(w1, add_array_array(w2, w3))

```

```

        return add_array_array(
            add_array_array(Hk, add_array_array(multi_num_array(-1.0 / b, a),
            multi_num_array(1.0 / d, c))),
            multi_array_array(w, T_Array(w)))

# 精确搜索步长
def golden_section(a, b, e, func, xk, dk):
    gold = 0.618
    x = b - gold * (b - a)
    y = a + gold * (b - a)
    while True:
        # print(f'a:{a}\tx:{x}\ty:{y}\tb:{b}')
        if b - a < e:
            return (b + a) / 2.0
        else:
            if func(x=xk, l=x, d=dk) <= func(x=xk, l=y, d=dk):
                b = y
                y = x
                x = b - gold * (b - a)
            else:
                a = x
                x = y
                y = a + gold * (b - a)

def BFGS(x, H, func):
    k = 0
    gk = get_g(x)
    if g_is_zero(gk, 0.001):
        return x
    while True:
        dk = multi_num_array(-1, multi_array_array(H, gk))
        l = golden_section(-100, 100, 0.0001, func, x, dk)
        # print(f"k:{k}\txk:{x}\tgk:{gk}\tdk{dk}\tlamda:{l}\tHk:{H}")
        # print(f'x:{x}')
        pre_x = x
        x = add_array_array(x, multi_num_array(l, dk))
        pre_g = gk
        gk = get_g(x)
        if g_is_zero(gk, 0.001):
            return x
        sk = add_array_array(x, multi_num_array(-1, pre_x))
        yk = add_array_array(gk, multi_num_array(-1, pre_g))
        H = get_Hk1(H, yk, sk)
        k += 1

if __name__ == '__main__':
    x = T_Array([[0.0, 0.0]])
    H = [[1.0, 0.0], [0.0, 1.0]]
    best = BFGS(x, H, func=func)
    print("最优解的横纵坐标为: ( {}, {} )".format(best[0][0], best[1][0]))

```

```

# 目标函数
def func1(x):
    # print(x)
    return x[0] * x[0] + x[1] * x[1] - x[0] * x[1] - 10 * x[0] - 4 * x[1] + 60

def func(x, d, l):
    x = T_Array(x)
    d = T_Array(d)
    x_new = []
    for i in range(len(x[0])):
        x_new.append(x[0][i] + l * d[0][i])
    # print(x_new)
    return func1(x_new)

# 求梯度
def get_g(x):
    x = T_Array(x)
    return T_Array([[2 * x[0][0] - x[0][1] - 10, 2 * x[0][1] - x[0][0] - 4]])

# 判断梯度是否为0
def g_is_zero(g, e):
    g = T_Array(g)
    for gi in g[0]:
        if abs(gi) > e:
            return False
    return True

# 矩阵相乘
def multi_array_array(x1, x2):
    # 初始化
    result = []
    for i in range(len(x1)):
        result_item = []
        for j in range(len(x2[0])):
            result_item.append(0)
        result.append(result_item)
    # 求值
    for i in range(len(result)):
        for j in range(len(result[0])):
            sum = 0
            for k in range(len(x1[0])):
                sum += x1[i][k] * x2[k][j]
            result[i][j] = sum
    return result

# 矩阵转置
def T_Array(x):
    result = []
    for i in range(len(x[0])):
        result_item = []

```

```

        for j in range(len(x)):
            result_item.append(0)
        result.append(result_item)
    for i in range(len(result)):
        for j in range(len(result[0])):
            result[i][j] = x[j][i]
    return result

```

矩阵的数乘

```

def multi_num_array(num, x):
    result = []
    for i in range(len(x)):
        result_item = []
        for j in range(len(x[0])):
            result_item.append(num * x[i][j])
        result.append(result_item)
    return result

```

矩阵的加法

```

def add_array_array(x1, x2):
    result = []
    for i in range(len(x1)):
        result_item = []
        for j in range(len(x1[0])):
            result_item.append(x1[i][j] + x2[i][j])
        result.append(result_item)
    return result

```

获得H_{k+1}

```

def BFGS_get_Hk1(Hk, yk, sk):
    a = multi_array_array(Hk, multi_array_array(yk,
multi_array_array(T_Array(yk), Hk)))
    b = multi_array_array(T_Array(yk), multi_array_array(Hk, yk))[0][0]
    c = multi_array_array(sk, T_Array(sk))
    d = multi_array_array(T_Array(yk), sk)[0][0]
    # 在DFP的基础上增添w
    w1 = (multi_array_array(T_Array(yk), multi_array_array(Hk, yk))[0][0]) ** 0.5
    w2 = multi_num_array(num=1.0 / multi_array_array(T_Array(yk), sk)[0][0],
x=sk)
    w3 = multi_num_array(num=-1.0 / multi_array_array(T_Array(yk),
multi_array_array(Hk, yk))[0][0],
                        x=multi_array_array(Hk, yk))
    w = multi_num_array(w1, add_array_array(w2, w3))

    return add_array_array(
        add_array_array(Hk, add_array_array(multi_num_array(-1.0 / b, a),
multi_num_array(1.0 / d, c))),
        multi_array_array(w, T_Array(w)))

def DFP_get_Hk1(Hk, yk, sk):
    a = multi_array_array(Hk, multi_array_array(yk,
multi_array_array(T_Array(yk), Hk)))

```

```

b = multi_array_array(T_Array(yk), multi_array_array(Hk, yk))[0][0]
c = multi_array_array(sk, T_Array(sk))
d = multi_array_array(T_Array(yk), sk)[0][0]
# #在DFP的基础上增添w
# w1 = (multi_array_array(T_Array(yk),multi_array_array(Hk,yk))[0][0])**0.5
# w2 = multi_num_array(num=1.0/multi_array_array(T_Array(yk),sk)[0][0],x=sk)
# w3 =
multi_num_array(num=-1.0/multi_array_array(T_Array(yk),multi_array_array(Hk,yk))
[0][0],x=multi_array_array(Hk,yk))
# w = multi_num_array(w1,add_array_array(w2,w3))

return add_array_array(Hk, add_array_array(multi_num_array(-1.0 / b, a),
multi_num_array(1.0 / d, c)))

```

精确搜索步长

```

def golden_section(a, b, e, func, xk, dk):
    gold = 0.618
    x = b - gold * (b - a)
    y = a + gold * (b - a)
    while True:
        # print(f'a:{a}\tx:{x}\ty:{y}\tb:{b}')
        if b - a < e:
            return (b + a) / 2.0
        else:
            if func(x=xk, l=x, d=dk) <= func(x=xk, l=y, d=dk):
                b = y
                y = x
                x = b - gold * (b - a)
            else:
                a = x
                x = y
                y = a + gold * (b - a)

def DFP(x, H, func):
    k = 0
    gk = get_g(x)
    if g_is_zero(gk, 0.001):
        return x
    while True:
        dk = multi_num_array(-1, multi_array_array(H, gk))
        l = golden_section(-100, 100, 0.0001, func, x, dk)
        # print(f"k:{k}\txk:{x}\tgk:{gk}\tdk{dk}\tlamda:{l}\tHk:{H}")
        # print(f'x:{x}')
        pre_x = x
        x = add_array_array(x, multi_num_array(l, dk))
        pre_g = gk
        gk = get_g(x)
        if g_is_zero(gk, 0.001):
            return x
        sk = add_array_array(x, multi_num_array(-1, pre_x))
        yk = add_array_array(gk, multi_num_array(-1, pre_g))
        H = DFP_get_Hk1(H, yk, sk)
        k += 1

```

```

def BFGS(x, H, func):
    k = 0
    gk = get_g(x)
    if g_is_zero(gk, 0.001):
        return x
    while True:
        dk = multi_num_array(-1, multi_array_array(H, gk))
        l = golden_section(-100, 100, 0.0001, func, x, dk)
        # print(f"k:{k}\txk:{x}\tgk:{gk}\tdk{dk}\tlamda:{l}\tHk:{H}")
        # print(f'x:{x}')
        pre_x = x
        x = add_array_array(x, multi_num_array(l, dk))
        pre_g = gk
        gk = get_g(x)
        if g_is_zero(gk, 0.001):
            return x
        sk = add_array_array(x, multi_num_array(-1, pre_x))
        yk = add_array_array(gk, multi_num_array(-1, pre_g))
        H = BFGS_get_Hk1(H, yk, sk)
        k += 1

## 以下为共轭梯度法
#####
#####
# 求梯度
def g(x):
    return [2 * x[0] - x[1] - 10, 2 * x[1] - x[0] - 4]

# 判断梯度是否为0
def eqaulzero(x, e):
    for xi in x:
        if abs(xi - 0) > e:
            return False
    return True

# 计算beta
# gk为g_k gk1为g_k-1
def beta(gk, gk1):
    a = 0.0
    b = 0.0
    for gki in gk:
        a += gki * gki
    for gk1i in gk1:
        b += gk1i * gk1i
    return a / b

# 计算步长λ
# 将x=x+λd 代入后得到关于λ的一元二次方程，从而得到了精确解
# 也可以用精确一维搜索或者非精确一维搜索
def lamda(x, d):
    x = T_Array([x, ])

```

```

d = T_Array([d, ])
return golden_section(-100, 100, 0.0001, func, x, d)

# 计算新的x
def xk(x, l, d):
    xk = []
    for i in range(len(x)):
        xk.append(x[i] + l * d[i])
    return xk

# 向量求负
def reverse(x):
    result = []
    for i in range(len(x)):
        result.append(-x[i])
    return result

# 计算新的方向d
def d(gk, betak, dk):
    result = []
    for i in range(len(gk)):
        result.append(-gk[i] + betak * dk[i])
    return result

# FR 共轭梯度法, x为初始解, e为精度
def FR(x, e):
    k = 0
    gk = g(x)
    dk = reverse(gk)
    # pre_g = gk
    # pre_d = dk
    l = lamda(x, dk)
    # print(f"k:{k}\txk:{x}\tgk:{gk}\tdk{dk}\tlamda:{l}")
    x = xk(x, l, dk)
    while True:
        k += 1
        # print(x)
        pre_g = gk
        gk = g(x)
        if eqaulzero(gk, e):
            return x
        else:
            betak = beta(gk, pre_g)
            dk = d(gk, betak, dk)
            l = lamda(x, dk)
            # print(f"k:{k}\txk:{x}\tgk:{gk}\tdk{dk}\tlamda:{l}")

            x = xk(x, l, dk)

if __name__ == '__main__':
    x = T_Array([[0.0, 0.0]])

```



```

H = [[1.0, 0.0], [0.0, 1.0]]
print("DFP:")
best1 = DFP(x, H, func)
print("最优解的横纵坐标为: ({} , {})".format(best1[0][0], best1[1][0]))
# print(DFP(x, H, func))
print()
print("BFGS:")
best2 = BFGS(x, H, func)
print("最优解的横纵坐标为: ({} , {})".format(best2[0][0], best2[1][0]))
# print(BFGS(x, H, func=func))
print()
print("FR共轭梯度法: ")
best3 = FR([0.0, 0.0], 0.000001)
print("最优解的横纵坐标为: ({} , {})".format(best3[0], best3[1]))
# print(FR([0.0, 0.0], 0.000001))

```

代码附录十

```

## 以第5题为例，利用SGD进行最优化求解
import torch

if __name__ == '__main__':
    # 定义一个可学习参数w，初值是0.1,1.0
    w1 = torch.tensor(data=[0.0], dtype=torch.float32, requires_grad=True)
    w2 = torch.tensor(data=[0.0], dtype=torch.float32, requires_grad=True)
    # 定义SGD优化器，nesterov=False，其余参数都有效
    optimizer1 = torch.optim.SGD(params=[w1, w2], lr=0.1, momentum=0.9,
dampening=0.5, weight_decay=0.01, nesterov=False)
    optimizer2 = torch.optim.Adam(params=[w1, w2], lr=0.001, betas=(0.9, 0.999),
eps=1e-8, weight_decay=0, amsgrad=False)

    # 进行优化
    for i in range(100):
        y = w1 ** 2 + w2 ** 2 - w1 * w2 - 10 * w1 - 4 * w2 + 60 # 优化的目标函数
        optimizer1.zero_grad() # 让w的偏导数置零
        y.backward() # 反向传播，计算w的偏导数
        optimizer1.step() # 根据上述两个公式，计算一个v，然后作用到w
        print('随机梯度下降法：迭代100轮后梯度为(%.2f,%.2f)，对应最优解坐标为(%.2f,%.2f)' %
(w1.grad, w2.grad, w1.data, w2.data)) # 查看w的梯度和更新后的值

    # 进行优化
    for i in range(100):
        y = w1 ** 2 + w2 ** 2 - w1 * w2 - 10 * w1 - 4 * w2 + 60 # 优化的目标函数
        optimizer2.zero_grad() # 让w的偏导数置零
        y.backward() # 反向传播，计算w的偏导数
        optimizer2.step() # 根据上述两个公式，计算一个v，然后作用到w
        print('Adam法：迭代100轮后梯度为(%.2f,%.2f)，对应最优解坐标为(%.2f,%.2f)' %
(w1.grad, w2.grad, w1.data, w2.data)) # 查看w的梯度和更新后的值

```

代码附录十一

```

import torch
import torch.nn as nn
import torch.optim as optim

```

```

import matplotlib.pyplot as plt

# 定义一个简单的多层感知器 MLP
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(2, 10) # 输入层到隐藏层
        self.fc2 = nn.Linear(10, 1) # 隐藏层到输出层

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 生成伪随机数据
torch.manual_seed(42)
num_samples = 100
x = torch.randn(num_samples, 2) # 100个二维样本
y = torch.randn(num_samples, 1) # 对应的标签

# 定义模型、损失函数和优化器
model = MLP()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# 训练过程
num_epochs = 100
param_history = [] # 用于存储参数历史
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(x)
    loss = criterion(outputs, y)
    loss.backward()
    optimizer.step()

    # 记录隐藏层参数的变化
    hidden_params = model.fc1.weight.data.numpy().flatten() # 将参数展平为一维
    param_history.append(hidden_params)

# 转换参数历史为数组并绘图
param_history = torch.tensor(param_history)
plt.figure(figsize=(8, 6))
plt.scatter(param_history[:, 0], param_history[:, 1], c='b', alpha=0.6)
plt.xlabel('Parameter 1')
plt.ylabel('Parameter 2')
plt.title('Parameter Changes in Training')
plt.grid(True)
plt.show()

```

