НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Факультет прикладної математики
Кафедра прикладної математики

Курсова робота

із дисципліни «Методи Оптимізації»

на тему

Збіжність методу Пірсона при мінімізації степеневої функції

Виконала:
студентка групи КМ-81
Верзун П.В.

Керівник:
канд. техн. наук, доцент
Ладогубець Т.С.

Київ — 2021

ЗМІСТ

ПОСТАНОВКА ЗАДАЧІ

Дослідити збіжність метода Пірсона при мінімізації степеневої функції

$$f_2(x)=(10(x_1-x_2)^2+(x_1-1)^2)^4,\ x^{(0)}=(-1{,}2\,;\,0{,}0);$$

в залежності від:

1. Величини кроку h при обчисленні похідних.
2. Схеми обчислення похідних.
3. Виду методу одновимірного пошуку (ДСК-Пауелла або Золотого перетину).
4. Точності методу одновимірного пошуку.
5. Значення параметру в алгоритмі Свена.

Використати метод штрафних функцій (метод внутрішньої точки) для умовної оптимізації в залежності від:

1. Розташування локального мінімума (всередині/поза допустимою областю).
2. Виду допустимої області (випукла/невипукла).

ТЕОРЕТИЧНА ЧАСТИНА

Методи змінної метрики. Квазіньютонівські методи

Для застосування методу Ньютона потрібно знати матрицю Гессе даної функції і здійснити обернення цієї матриці. У багатьох випадках може виявитися, що матриці Гессе невідомі, або їх обчислення пов'язано з великими труднощами, або вони можуть бути отримана тільки чисельними методами.

Методи Змінної метрики апроксимують матрицю Гессе або обернену до неї, використовуючи для цього значення тільки перших похідних. У більшості цих методів використовуються спряжені напрямки. Перехід з точки $x^{(k)}$ в точку $x^{(k+1)}$ визначається наступним чином:

$$x^{(k+1)} = x^{(k)} - \lambda^{*(k)} A(x^{(k)}) \nabla f(x^{(k)}),$$

де

$A(x^{(k)})$ - матриця порядку n×*n*, що називається метрикою, змінюється на кожній ітерації і являє собою апроксимацію оберненої матриці Гессе;

$\lambda^{*(k)}$ - скаляр, що визначається за допомогою методів одновимірного пошуку;

$\nabla f(x^{(k)})$ - значення градієнта в точці $x^{(k)}$.

Для апроксимації матриці, оберненої до матриці Гессе, використовується наступне співвідношення:

$$A^{(k+1)} = A^{(k)} + A_c^{(k)},$$

де

$A_c^{(k)}$ - коригуюча матриця.

Необхідно Побудувати Матрицю $A^{(k)}$ таким чином, щоб послідовність $A^{(0)}, A^{(1)}, A^{(2)}, \ldots, A^{(k+1)}$ давала наближення до $\mathrm{H}^{-1} = \nabla^2 f(x^*)$. Вибір $A_c^{(k)}$ визначає

конкретний метод змінної метрики. Для забезпечення збіжності матриця $A^{(k+1)}$ має бути додатньо визначеною.

Розкладемо в ряд Тейлора $\nabla f(x)$:

$$\nabla f(x) = \nabla f(x^{(k)}) + \mathrm{H}(x^{(k)})(x - x^{(k)}),$$

тоді для $x^{(k+1)}$ маємо:

$$\nabla f(x^{(k+1)}) = \nabla f(x^{(k)}) + \mathrm{H}(x^{(k)})(x^{(k+1)} - x^{(k)})$$

$$\nabla f(x^{(k+1)}) - \nabla f(x^{(k)}) = \mathrm{H}(x^{(k)})(x^{(k+1)} - x^{(k)}).$$

Помноживши обидві частини рівняння на $\mathrm{H}^{-1}(x^{(k)})$, отримаємо:

$$(x^{(k+1)} - x^{(k)}) = \mathrm{H}^{-1}(x^{(k)})\left[\nabla f(x^{(k+1)}) - \nabla f(x^{(k)})\right].$$

(Якщо *f(x)* – квадратична функція, то $\mathrm{H}(x^{(k)}) = \mathrm{H}$, тобто матриця Гессе постійна). Це рівняння можна розглядати як систему *n* лінійних рівнянь, що містять *n невідомих* параметрів, які необхідно оцінити для того, щоб апроксимувати $\mathrm{H}^{-1}(x)$ або $\mathrm{H}(x)$ при заданих значеннях $f(x), \nabla f(x), \Delta x$ на більш ранніх етапах пошуку. Для вирішення цих лінійних рівнянь можуть бути використані різні методи, кожен з яких призводить до різних методів змінної метрики.У досить великій групі методів $\mathrm{H}^{-1}(x^{(k+1)})$ апроксимується за допомогою інформації, отриманої на*к*-му кроці:

$$\mathrm{H}^{-1}(x^{(k+1)}) \approx \omega A^{(k+1)} = \omega(A^{(k)} + A_c^{(k)}),$$

де

ω - маштабуючий множник (константа, що зазвичай дорівнює 1).

На *(k+1)* – му кроці відомі наступні значення: $x^{(k)}, \nabla f(x^{(k)}), \nabla f(x^{(k+1)}), A^{(k)}$, потрібно нарахувати $A^{(k+1)}$ таким чином, що задовольняє співвідношення:

$$A^{(k+1)}\left[\nabla f(x^{(k+1)}) - \nabla f(x^{(k)})\right] = \frac{1}{\omega}(x^{(k+1)} - x^{(k)})$$

або

$$A^{(k+1)}\Delta g^{(k)} = \frac{1}{\omega}\Delta x^{(k)},$$

де

$$\Delta g^{(k)} = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)})$$

$$\Delta x^{(k)} = x^{(k+1)} - x^{(k)}.$$

Нехай $A_c^{(k)} = A^{(k+1)} - A^{(k)}$, підставляючи у попереднє рівняння, отримаємо:

$$A_c^{(k)} \Delta g^{(k)} = \frac{1}{\omega} \Delta x^{(k)} - A^{(k)} \Delta g^{(k)}.$$

Це рівняння необхідно вирішити відносно $A_c^{(k)}$. Воно має наступний розв'язок:

$$A_c^{(k)} = \frac{1}{\omega} \frac{\Delta x^{(k)} y^{\mathrm{T}}}{y^{\mathrm{T}} \Delta g^{(k)}} - \frac{A^{(k)} \Delta g^{(k)} z^{\mathrm{T}}}{z^{\mathrm{T}} \Delta g^{(k)}},$$

де

y, z– довільні вектори розмірності n×*1*.

Якщо при ω*=1 обирається* лінійна комбінація з двох напрямків $\Delta x^{(k)}$ та $A^{(k)} \Delta g^{(k)}$, а саме:

$y = z = \Delta x^{(k)} - A^{(k)} \Delta g^{(k)}$ - виходить алгоритм Бройдена.

Якщо ж $y = \Delta x^{(k)}, \quad z = A^{(k)} \Delta g^{(k)}$, матриця обраховується по алгоритму Дэвидона –Флетчера - Пауэлла.

Якщо $y = z = \nabla x^{-k}$ то наближення матриці напрямів буде виглядати так:

$$\eta^{k+1} = \eta^k + \frac{\left[\Delta \bar{x}^k - \eta^k \cdot \Delta \bar{g}^k\right] \cdot \left(\Delta \bar{x}^k\right)^{\mathrm{T}}}{\left(\Delta \bar{x}^k\right)^{\mathrm{T}} \cdot \Delta \bar{g}^k},$$ і отримаємо 2 алгоритм Пірсона, а якщо

$$\eta^{k+1} = \eta^k + \frac{\left[\Delta \bar{x}^k - \eta^k \cdot \Delta \bar{g}^k\right] \cdot \left[\eta^k \Delta \bar{g}^k\right]^{\mathrm{T}}}{\left(\Delta \bar{g}^k\right)^{\mathrm{T}} \eta^k \Delta \bar{g}^k},$$

$\bar{y} = \bar{z} = \eta^k \cdot \Delta \bar{g}^k$, $\eta^0 = R^0$. , отримаємо 3 алгоритм Пірсона. Можуть бути й інші можливості вибору векторів *y, z*.

Якщо кроки $\Delta x^{(k)}$ визначаються шляхом мінімізації *f(x)* в напрямку $S^{(k)}$, то всі методи, за допомогою яких вираховують симетричну матрицю $A^{(k+1)}$, що задовольняє умові

$$A^{(k+1)} \Delta g^{(k)} = \frac{1}{\omega} \Delta x^{(k)},$$

дають напрямки, які є взаємно спряженими (для квадратичної функції).

Квазіньютонівські методи можуть бути отримані за допомогою загального підходу до побудови методів спряжених напрямів, запропонованого Хуаном. Це означає, напрямки пошуку, що генеруються за допомогою квазіньютонівських методів, є спряденими, що дозволяє говорити про надлінійну швидкість збіжності квазіньютонівських алгоритмів, за умови точного визначення значень $\lambda^{*(k)}$ і навіть при неточному одновимірному пошуку, але при виконанні умови:

$$\Delta x^{(i)} = A^{(k)} \Delta g^{(i)} \quad i = 1,2\square \,, k \quad k \geq n$$

Ефективність квазіньютонівських процедур слабко залежить від точності одновимірного пошуку.

Методи Пірсона

Пірсон запропонував кілька методів з апроксимацією зворотного гессіану без явного обчислення других похідних, тобто шляхом спостережень за змінами напрямку антиградієнта. при цьому отримуються спряжені напрями. Ці алгоритми відрізняються тільки деталями. Далі наведено ті з них, які отримали найбільш широке поширення в прикладних областях.

Алгоритм Пірсона № 2

У цьому алгоритмі зворотний гессіан апроксимується матрицею H^k, що обчислюється на кожному кроці по формулі

$$H^{k+1} = H^k + \frac{\left((X^{k+1} - X^k) - H^k \cdot (\nabla f(X^{k+1}) - \nabla f(X^k))\right) \cdot (X^{k+1} - X^k)^T}{(X^{k+1} - X^k)^T \cdot \left(\nabla f(X^{k+1}) - \nabla f(X^k)\right)}.$$

В якості початкової матриці H^0 обирають довільну додатньо визначену симетричну матрицю. Даний алгоритм Пірсона часто призводить до ситуацій, коли матриця H^k починає осцилювати - коливаючись між додатньо визначеною і не

додатньо визначеною, при цьому визначник матриці близький до нуля. Для уникнення цієї ситуації необхідно через кожні n кроків перевизначати матрицю, прирівнюючи її до H^0.

Алгоритм Пірсона № 3

У цьому алгоритмі матриця H^{k+1} визначається за формулою

$$H^{k+1} = H^k + [X^{k+1} - X^k - H^k \cdot (\nabla f(X^{k+1}) - \nabla f(X^k))] \frac{\left(H^k \cdot (\nabla f(X^{k+1}) - \nabla f(X^k))\right)^T}{\left(\nabla f(X^{k+1}) - \nabla f(X^k)\right)^T \cdot H^k \cdot \left(\nabla f(X^{k+1}) - \nabla f(X^k)\right)}.$$

Траєкторія спуску, породжена цим алгоритмом, аналогічна поведінці алгоритму Дэвидона-Флетчера-Пауелла, але кроки h коротше. Пірсон також запропонував різновид цього алгоритму з циклічним перезаданіем матриці.

РОЗРАХУНКОВА ЧАСТИНА

У даній частині наведені результати оцінки збіжності алгоритму Пірсона (обрано алгоритм пірсона 3) в залежності від параметрів. Критерієм оцінки виступає кількість обчислень невідомої функції f.

Аналітичне обчислення

Мінімум функції був обчислений аналітично за допомогою сервісу wolfram.alpha. Дана функція має найменше значення f(x) = 0 у точці (1; 1).

1. Оцінка залежності збіжності методу від величини кроку h обчислення похідних

У даному пункті розрахування проводились при наступних фіксованих параметрах:

Схема обчислення похідних – ліва

Метод одновимірного пошуку – ДСК-Пауелла

Точність методу одновимірного пошуку – 0.01

Параметр у алгоритмі Свенна – 0.1

Точність методу – 0.001

Таблиця 1. Залежність збіжності методу від кроку обчислення похідних

Значення кроку h обчислення похідних	Кількість обчислень функції	Знайдена точка мінімуму	Значення функції
0.1	257	[1.0157161342428926, 1.0146866611674394]	4.403012065434074e-15
0.01	696	[1.1434371776832453, 1.143750272698839]	1.7921593344692182e-07
0.001	779	[0.9791442126235174, 0.9766757888316703]	6.047265176896031e-14
0.0001	923	[0.9868885150926394, 0.9864485188274148]	2.6620331376166223e-17
0.00001	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
0.000001	523	[0.971693113391159, 0.9723283465130793]	4.205941632641038e-13

Найбільш оптимальним значенням кроку обчислення похідних є 0.00001.

2. Оцінка залежності збіжності методу від схеми обчислення похідних

У даному пункті розрахування проводились при наступних фіксованих параметрах:

Крок обчислення похідних – 0.00001

Метод одновимірного пошуку – ДСК-Пауелла

Точність методу одновимірного пошуку – 0.01

Параметр у алгоритмі Свенна – 0.1

Точність методу – 0.001

Таблиця 2. Залежність збіжності методу від схеми обчислення похідних

Схема обчислення похідних	Кількість обчислень функції	Знайдена точка мінімуму	Значення функції
Ліва	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
Права	486	[0.9513070868834709, 0.9509877789773657]	3.165723059438061e-11
Центральна	1335	[0.9550626906381624, 0.9507424164157785]	2.368260583892649e-11

Отже, найбільш оптимальною схемою обчислення похідних є ліва.

3. Оцінка залежності збіжності методу від виду одновимірного пошуку

У даному пункті розрахування проводились при наступних фіксованих параметрах:

Крок обчислення похідних – 0.00001

Схема обчислення похідних – ліва

Точність методу одновимірного пошуку – 0.01

Параметр у алгоритмі Свенна – 0.1

Точність методу – 0.001

Таблиця 3. Залежність збіжності методу від методу одновимірного пошуку

Метод одновимірного пошуку	Кількість обчислень функції	Знайдена точка мінімуму	Значення функції
ДСК-Пауелла	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
Золотий перетин	526	[0.979172972460655, 0.978276512421345]	3.8098569905062554e-14

Отже, найбільш оптимальним методом одновимірного пошуку є метод ДСК-Пауелла .

4. Оцінка залежності збіжності методу від точності одновимірного пошуку

У даному пункті розрахування проводились при наступних фіксованих параметрах:

Крок обчислення похідних – 0.00001

Схема обчислення похідних – ліва

Метод одновимірного пошуку – ДСК-Пауелла

Параметр у алгоритмі Свенна – 0.1

Точність методу – 0.001

Таблиця 4. Залежність збіжності методу від точності методу одновимірного пошуку

Точність методу одновимірного пошуку	Кількість обчислень функції	Знайдена точка мінімуму	Значення функції
0.1	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
0.01	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
0.001	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
0.0001	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
0.00001	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
0.000001	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20

Нехай найбільш оптимальною точністю методу одновимірного пошуку є 0.001.

5. Оцінка залежності збіжності методу від значення параметру в алгоритмі Свенна

У даному пункті розрахування проводились при наступних фіксованих параметрах:

Крок обчислення похідних – 0.00001

Схема обчислення похідних – ліва

Метод одновимірного пошуку - ДСК-Пауелла

Точність методу одновимірного пошуку – 0.001

Точність методу – 0.001

Таблиця 5. Залежність збіжності методу від значення параметру в алгоритмі Свенна

Значення параметру в алгоритмі Свенна	Кількість обчислень функції	Знайдена точка мінімуму	Значення функції
0.1	238	[1.000683392607946, 1.001942068724517]	7.07590315090186e-20
0.5	488	[0.8816844886653227, 0.8786025156753593]	3.945309555148154e-08
0.01	269	[-0.4590389374544552, -0.5523889697319542]	24.111796120302625
0.001	267	[0.9900588118689696, 0.9889233868118137]	1.5577941071184947e-16
0.0001	653	[0.9804170715895828, 0.9799179250426171]	2.2195795731558443e-14

Отже, найбільш оптимальним значенням параметру в алгоритмі Свенна є 0.1.

Використовуючи усі оптимальні значення параметрів методу, знайдених вище, алгоритм був удосконалений для вирішення задач умовної оптимізації.

Нехай маємо наступне обмеження нерівності:

$- x_1^2 - x_2^2 + 0.8 \geq 0$

Штрафна функція має вигляд: $(10(x1-x2)^2 +(x1-1)^2)^4 +R(0.8-x1^2-x2^2)^2$

1. Оцінка залежності збіжності методу від точності методу для умовної оптимізації

Таблиця 7. Залежність збіжності методу від штрафного параметру для умовної оптимізації (область випукла)

R	X1	X2	f(x1, x2)

Точка поза областю і має ввійти і пройти через область			
-	-1.2	-1	754.18875
0.1	0.6590376	0.5880662729	0.000810219
1	0.6590376	0.5880662729	0.000810219
2	0.6590376	0.5880662729	0.000810219
5	0.6590376	0.5880662729	0.000810219
Точка поза областю і має дійти до границі			
-	2	0.3	799254.9625
0.1	0.633230389	0.63173386402	0.0003276699
1	0.633230389	0.63173386402	0.0003276699
10	0.633230389	0.63173386402	0.0003276699
Точка всередині області			
-	-0.2	0	11.52004
0.1	0.6263663	0.638793618	0.000396914
1	0.6263663	0.638793618	0.000396914
10	0.6263663	0.638793618	0.000396914

Додамо еще одне обмеження та зробимо допустиму область невипуклою:

Додаткове обмеження : $x_1^2 + x_2^2 - 0.4 \geq 0$

Штрафна фунція має вигляд: $(10(x1-x2)^2 + (x1-1)^2)^4 + R(0.8-x1^2-x2^2)^2 + R(x_1^2 + x_2^2 - 0.4)^2$

Таблиця 8. Залежність збіжності методу від штрафного параметру для умовної оптимізації (область невипукла)

R	X1	X2	f(x1, x2)
Точка поза областю і має ввійти і пройти через область			
-	-1.2	-1	754.60491
0.1	0.6328658355	0.64884006086	0.011849998
1	0.6328658355	0.64884006086	0.011849998
10	0.6328658355	0.64884006086	0.011849998
Точка поза областю і має дійти до границі			
-	2	0.3	799254.124501
0.01	0.64129838	0.6444134595	0.002101238849
1	0.64129838	0.6444134595	0.002101238849
10	0.64129838	0.6444134595	0.002101238849
Точка всередині області			
-	-0.6	-0.5	11.52004
0.1	0.6768306525	0.54851278788	0.006548776277
1	0.6768306525	0.54851278788	0.006548776277
10	0.6768306525	0.54851278788	0.006548776277

ВИСНОВКИ

У даній роботі був реалізований алгоритм методу Пірсона номер 3 пошуку мінімуму функції за допомогою мови програмування Python. Була досліджена збіжність даного методу для степеневої функції із заданої початкової точки, критерієм даної збіжності слугувала кількість обчислень функції.

Була досліджена збіжність методу Пірсона для степеневої функції в залежності від: кроку обчислення похідних, схеми обчислення похідних, методу одновимірного пошуку, точності методу одновимірного пошуку, параметрів у алгоритмі Свенна, критерію закінчення.

Нижче представлені результати дослідження.

Функція досить сильно залежить від кроку обчислення похідних та зі зменшенням його значення в 10 разів кількість обчислень функції зростає в середньому на 100. Оптимальним значенням (тобто, який не знижує точність підрахунків та водночас має відносно невелику кількість обчислень функції), є 10^{-5}.

Залежність від схеми обчислення похідних є практично однаковою у випадку правої або лівої схеми. Найбільш оптимальною була визначена ліва схема, оскільки у даному випадку для неї кількість обчислень функції була дещо менша.

Стосовно методу одновимірного пошуку, то метод ДСК проявив себе краще, ніж метод золотого перетину, що також є очевидним, оскільки методи, що використовують квадратичну апроксимацію, працюють швидше, ніж методи, що використовують правило виключення інтервалів.

Даний метод слабко залежить від точності методу одновимірного пошуку, за єдиним лише винятком – для малої точності (0.1) методу ДСК-Пауелла кількість обрахування функції було більше, ніж навіть для вищих точностей. Найбільш оптимальною точністю у даному випадку було визначено 10^{-3}.

Також не дуже сильно збіжність даного методу залежить від значення параметру в алгоритмі Свенна. Стандартне значення 0.1 і виявилось найбільш оптимальним у даному випадку.

Отже, при наступних параметрах:

- Крок обчислення похідних – 0.00001;
- Схема обчислення похідних – ліва;
- Метод одновимірного пошуку – ДСК-Пауелла;
- Точність методу одновимірного пошуку – 0.001;
- Параметр у алгоритмі Свенна – 0.1;
- Точність методу – 0.0001

був знайдений мінімум [1.0002488247144288, 1.0002420818478501] за 394 ітерацій. Викладки, що наведені вище, стосувались випадку безумовної оптимізації.

Для умовної оптимізації був використаний наступний метод штрафних функцій: зовнішні штрафи. Дослідження проведено показало, що не залежно від виду допустимої області, місця знаходження точки функція досягає границі допустимої області, вирішуються дві задачі безумовної оптимізації.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Т. С. Ладогубець. «Методи оптимізації» - електронний конспект. Київ.
2. Д. Хіммельблау. «Прикладне нелінійне програмування». М.; «Мир», 1975.
3. Г.В. Реклейтіс «Оптимизация в технике» частина 1, М.; «Мир», 1986.

Лістинг програми наведений нижче:

```
import math
from numpy import *
from sympy import *
from math import *


def cur_func(x1, x2):
    global func_usage
    func_usage = func_usage + 1
    return Stepen_func(x1, x2)


def Rozenbrok_func(x1, x2):
    return 100 * (x1 * x1 - x2) ** 2 + (x1 - 1) ** 2


def Stepen_func(x1, x2):
    return (10 * (x1 - x2) ** 2 + (x1 - 1) ** 2) ** 4


def my_func(x1, x2):
    return (x1 - 1) ** 2 + (x2 - 1) ** 2


def left_derivative(point, h, var_to_count, cur_func_value):
    if var_to_count == "x1":
        return (cur_func_value - cur_func(point[0] - h, point[1])) / h
    if var_to_count == "x2":
        return (cur_func_value - cur_func(point[0], point[1] - h)) / h


def right_derivative(point, h, var_to_count, cur_func_value):
    if var_to_count == "x1":
        return (cur_func(point[0] + h, point[1]) - cur_func_value) / h
    if var_to_count == "x2":
```

```
        return (cur_func(point[0], point[1] + h) - cur_func_value) / h


def central_derivative(point, h, var_to_count):
    if var_to_count == "x1":
        return (cur_func(point[0] + h, point[1]) - cur_func(point[0] - h, point[1]))
/ (
            2 * h
        )
    if var_to_count == "x2":
        return (cur_func(point[0], point[1] + h) - cur_func(point[0], point[1] - h))
/ (
            2 * h
        )


def find_gradient(point, cur_func_value):

    return [
        left_derivative(point, h, "x1", cur_func_value),
        left_derivative(point, h, "x2", cur_func_value),
    ]


# return [right_derivative(point, h,"x1", cur_func_value), right_derivative(point,
h,"x2", cur_func_value)]
# return [central_derivative(point, h,"x1"), central_derivative(point, h,"x2")]


def find_norm(thing):
    return abs(math.sqrt(thing[0] * thing[0] + thing[1] * thing[1]))


def find_delta_lambda(norm_point, norm_step):
    global parametr_Svena
    try:
```

```
        step = parametr_Svena * (norm_point / norm_step)
    except ZeroDivisionError:
        step = parametr_Svena
    return step


def System_End_Criteria(cur_point, cur_func_value, prevoius_point,
prevoius_func_value):
    point_difference = [
        cur_point[0] - prevoius_point[0],
        cur_point[1] - prevoius_point[1],
    ]
    point_error = find_norm(point_difference) / find_norm(prevoius_point)

    func_error = abs(cur_func_value - prevoius_func_value)
    return [point_error, func_error]


def Sven_algorithm(delta_lambda, point, func_value, gradient):
    local_delta_lambda = 0
    cur_lambda = 0
    plus_lambda = delta_lambda
    minus_lambda = -delta_lambda
    plus_point = [
        point[0] - plus_lambda * gradient[0],
        point[1] - plus_lambda * gradient[1],
    ]
    minus_point = [
        point[0] - minus_lambda * gradient[0],
        point[1] - minus_lambda * gradient[1],
    ]

    print(
        f"/n/n АЛГОРИТМ СВЕНА /n/n Start point: {point},  {func_value}/nDelta Lambda:
{delta_lambda} /nAnti-Gradient: {[-gradient[0],-gradient[1]]} /n"
    )
```

```
    plus_func = cur_func(plus_point[0], plus_point[1])
    minus_func = cur_func(minus_point[0], minus_point[1])

    print(
        f"Plus delta: {plus_point},  {plus_func}/nMinus Delta {minus_point},
{minus_func}/n"
    )

    global next_func_value
    global cur_func_value
    cur_func_value = func_value

    print(f"Начальное значение функции: {cur_func_value}")
    if plus_func > func_value and minus_func > func_value:
        while plus_func > func_value and minus_func > func_value:
            plus_lambda = plus_lambda / 2
            minus_lambda = minus_lambda / 2
            plus_point = [
                point[0] - plus_lambda * gradient[0],
                point[1] - plus_lambda * gradient[1],
            ]
            minus_point = [
                point[0] - minus_lambda * gradient[0],
                point[1] - minus_lambda * gradient[1],
            ]

            plus_func = cur_func(plus_point[0], plus_point[1])
            minus_func = cur_func(minus_point[0], minus_point[1])
            print(
                f"NEW plus delta: {plus_point},  {plus_func}/nNEW Minus Delta
{minus_point},  {minus_func}/n"
            )
    if plus_func < func_value:
        local_delta_lambda = plus_lambda
        point = plus_point
```

```
        next_func_value = plus_func
        func_value = plus_func
    elif minus_func < func_value:
        local_delta_lambda = minus_lambda
        point = minus_point
        next_func_value = minus_func

    print(f"Теперешнее значение функции {next_func_value}/n")

    points_arr = []
    func_value_arr = []
    lambda_valu_arr = []
    points_arr.append(point)
    func_value_arr.append(next_func_value)
    lambda_valu_arr.append(local_delta_lambda)

    k = 1
    while next_func_value < cur_func_value:
        cur_lambda = cur_lambda + local_delta_lambda * 2 ** k
        new_point = [
            point[0] - cur_lambda * gradient[0],
            point[1] - cur_lambda * gradient[1],
        ]
        points_arr.append(new_point)

        lambda_valu_arr.append(cur_lambda)
        cur_func_value = next_func_value
        next_func_value = cur_func(new_point[0], new_point[1])
        func_value_arr.append(next_func_value)

        print(f"Новая точка: {new_point}")
        print(f"Теперешнее значение функции {next_func_value}/n")
        k = k + 1

    cur_lambda = cur_lambda - local_delta_lambda * k
    new_point = [
```

```
            point[0] - cur_lambda * gradient[0],
            point[1] - cur_lambda * gradient[1],
        ]
        points_arr.append(new_point)
        lambda_valu_arr.append(cur_lambda)
        print(f"Новая точка: {new_point}")
        next_func_value = cur_func(new_point[0], new_point[1])
        func_value_arr.append(next_func_value)
        print(f"Теперешнее значение функции {next_func_value}/n")

        return [
            [
                points_arr[len(points_arr) - 4],
                func_value_arr[len(func_value_arr) - 4],
                lambda_valu_arr[len(points_arr) - 4],
            ],
            [
                points_arr[len(points_arr) - 3],
                func_value_arr[len(func_value_arr) - 3],
                lambda_valu_arr[len(points_arr) - 3],
            ],
            [
                points_arr[len(points_arr) - 1],
                func_value_arr[len(points_arr) - 1],
                lambda_valu_arr[len(points_arr) - 1],
            ],
        ]


def DSK_Pauela(left_border_arr, middle_arr, right_border_arr, gradient):
    print("/nДСК-ПАУЭЛА/n")
    delta_x = [
        middle_arr[0][0] - left_border_arr[0][0],
        middle_arr[0][1] - left_border_arr[0][1],
    ]
    try:
```

```
        star_x = [
            middle_arr[0][0]
            + (
                (delta_x[0] * (left_border_arr[1] - right_border_arr[1]))
                / (2 * (left_border_arr[1] - 2 * middle_arr[1] +
right_border_arr[1]))
            ),
            middle_arr[0][1]
            + (
                (delta_x[1] * (left_border_arr[1] - right_border_arr[1]))
                / (2 * (left_border_arr[1] - 2 * middle_arr[1] +
right_border_arr[1]))
            ),
        ]
    except ZeroDivisionError:
        star_x = [middle_arr[0], middle_arr[1]]
        return star_x

    star_func_value = cur_func(star_x[0], star_x[1])
    print(f"Point : {star_x}/n Function: {star_func_value}")

    global tochnost_odn
    while abs(left_border_arr[1] - right_border_arr[1]) > tochnost_odn:
        if left_border_arr[1] < right_border_arr[1]:
            right_border_arr = middle_arr
        else:
            left_border_arr = middle_arr

        middle_arr = [star_x, star_func_value]
        a_1 = [
            (middle_arr[1] - left_border_arr[1])
            / (middle_arr[0][0] - left_border_arr[0][0]),
            (middle_arr[1] - left_border_arr[1])
            / (middle_arr[0][1] - left_border_arr[0][1]),
        ]
        a_2 = [
```

```
            (1 / (right_border_arr[0][0] - middle_arr[0][0]))
            * (
                (
                    (right_border_arr[1] - left_border_arr[1])
                    / (right_border_arr[0][0] - left_border_arr[0][0])
                )
                - a_1[0]
            ),
            (1 / (right_border_arr[0][1] - middle_arr[0][1]))
            * (
                (
                    (right_border_arr[1] - left_border_arr[1])
                    / (right_border_arr[0][1] - left_border_arr[0][1])
                )
                - a_1[1]
            ),
        ]
        star_x = [
            ((left_border_arr[0][0] + middle_arr[0][0]) / 2) - (a_1[0] / (2 *
a_2[0])),
            ((left_border_arr[0][1] + middle_arr[0][1]) / 2) - (a_1[1] / (2 *
a_2[1])),
        ]
        star_func_value = cur_func(star_x[0], star_x[1])
        print(a_1, a_2, star_x, star_func_value)
    return [star_x, star_func_value]


def gold_section(left_border_arr, least_func_value_arr, right_border_arr, gradient):

    print("/n3ОЛОТОЕ СЕЧЕНИЕ/n")
    alpha = right_border_arr[2] - left_border_arr[2]
    print(f"Gradient: {gradient}  /nAlpha: {alpha}/n")
    x1_lambda = left_border_arr[2] + 0.382 * alpha
    x2_lambda = left_border_arr[2] + 0.618 * alpha
```

```
    print(
        f"Lambdas : /nLeft border: {left_border_arr[2]}  /nX1_lambda: {x1_lambda} \
        /nMiddle: {least_func_value_arr[2]} /nX2_lambda: {x2_lambda} \
        /nRight_border: {right_border_arr[2]}/n"
    )

    x1_dot = [
        left_border_arr[0][0] - x1_lambda * gradient[0],
        left_border_arr[0][1] - x1_lambda * gradient[1],
    ]
    x2_dot = [
        left_border_arr[0][0] - x2_lambda * gradient[0],
        left_border_arr[0][1] - x2_lambda * gradient[1],
    ]

    x1_dot_func = cur_func(x1_dot[0], x1_dot[1])
    x2_dot_func = cur_func(x2_dot[0], x2_dot[1])

    print(
        f"Points: /nLeft: {left_border_arr[0], left_border_arr[1]} /nX1: {x1_dot,
x1_dot_func} \
         /nMiddle: {least_func_value_arr[0], least_func_value_arr[1]} /nX2 :
{x2_dot,x2_dot_func} \
         /nRight: {right_border_arr[0],right_border_arr[1]}"
    )

    while left_border_arr[1] > x1_dot_func and right_border_arr[1] > x2_dot_func:
        if left_border_arr[1] < right_border_arr[1]:
            right_border_arr = [x2_dot, x2_dot_func, x2_lambda]

        else:
            left_border_arr = [x1_dot, x1_dot_func, x1_lambda]

        alpha = right_border_arr[2] - left_border_arr[2]

        print(f"Gradient: {gradient}  /nAlpha: {alpha}/n")
```

```
        global tochnost_odn
        if alpha ⩽ tochnost_odn:
            break

        x1_lambda = left_border_arr[2] + 0.382 * alpha
        x2_lambda = left_border_arr[2] + 0.618 * alpha

        print(
            f"Lambdas : /n Left border: {left_border_arr[2]}  /n X1_lambda:
{x1_lambda} \
            /n Middle: {(right_border_arr[2] + left_border_arr[2])/2} /nX2_lambda:
{x2_lambda} \
            /nRight_border: {right_border_arr[2]}/n"
        )

        x1_dot = [
            left_border_arr[0][0] - x1_lambda * gradient[0],
            left_border_arr[0][1] - x1_lambda * gradient[1],
        ]
        x2_dot = [
            left_border_arr[0][0] - x2_lambda * gradient[0],
            left_border_arr[0][1] - x2_lambda * gradient[1],
        ]

        x1_dot_func = cur_func(x1_dot[0], x1_dot[1])
        x2_dot_func = cur_func(x2_dot[0], x2_dot[1])

        print(
            f"Points: /n Left: {left_border_arr[0], left_border_arr[1]} /nX1:
{x1_dot, x1_dot_func} \
             /nMiddle: {least_func_value_arr[0], least_func_value_arr[1]} /nX2 :
{x2_dot,x2_dot_func} \
             /nRight: {right_border_arr[0],right_border_arr[1]}"
        )
    mindot = left_border_arr
    if mindot[1] > x1_dot_func:
```

```
        mindot = [x1_dot, x1_dot_func]
    if mindot[1] > x2_dot_func:
        mindot = [x2_dot, x2_dot_func]
    if mindot[1] > right_border_arr[1]:
        mindot = right_border_arr

    return mindot


def Pearson3(
    start_point, start_func_value, start_gradient, start_point_norm, start_norm_grad
):
    global criteriy_okonchania
    cur_point = start_point
    cur_func_value = start_func_value
    cur_gradient = start_gradient
    cur_point_norm = start_point_norm
    cur_norm_grad = start_norm_grad
    a = [1, 0, 0, 1]
    end = [100, 100]
    # cur_norm_grad > criteriy_okonchania
    while end[0] > criteriy_okonchania or end[1] > criteriy_okonchania:
        print(f"/n/nНОВАЯ ИТЕРАЦИЯ ДФП/n/n")
        dirrection = [
            a[0] * cur_gradient[0] + a[1] * cur_gradient[0],
            a[2] * cur_gradient[1] + a[3] * cur_gradient[1],
        ]
        print(f"Direction: {[-dirrection[0], -dirrection[1]]}/n")
        interval = Sven_algorithm(
            find_delta_lambda(cur_point_norm, find_norm(dirrection)),
            cur_point,
            cur_func_value,
            dirrection,
        )
        print(f"Sven result: {interval}/n")
```

```
        # (DSK_Pauela или gold_section)
        final_result = gold_section(interval[0], interval[1], interval[2],
cur_gradient)
        print(f"Final: {final_result}")
        new_point = final_result[0]
        new_func_value = final_result[1]

        end = System_End_Criteria(new_point, new_func_value, cur_point,
cur_func_value)
        print(f"Criteria: {end}/n")
        # cur_norm_grad < criteriy_okonchania
        if end[0] < criteriy_okonchania and end[1] < criteriy_okonchania:
            break
        new_grad = find_gradient(new_point, new_func_value)

        delta_x = [new_point[0] - cur_point[0], new_point[1] - cur_point[1]]
        delta_g = [new_grad[0] - cur_gradient[0], new_grad[1] - cur_gradient[1]]

        first_up = [
            delta_x[0] * delta_x[0],
            delta_x[0] * delta_x[1],
            delta_x[1] * delta_x[0],
            delta_x[1] * delta_x[1],
        ]
        first_down = delta_x[0] * delta_g[0] + delta_x[1] * delta_g[1]
        first_go = [
            first_up[0] / first_down,
            first_up[1] / first_down,
            first_up[2] / first_down,
            first_up[3] / first_down,
        ]

        second_up_one = [
            a[0] * delta_g[0] + a[1] * delta_g[1],
            a[2] * delta_g[0] + a[3] * delta_g[1],
        ]
```

```
second_up_two = [
    second_up_one[0] * delta_g[0],
    second_up_one[0] * delta_g[1],
    second_up_one[1] * delta_g[0],
    second_up_one[1] * delta_g[1],
]
second_up_three = [
    second_up_two[0] * a[0] + second_up_two[1] * a[2],
    second_up_two[0] * a[1] + second_up_two[1] * a[3],
    second_up_two[2] * a[0] + second_up_two[3] * a[2],
    second_up_two[2] * a[1] + second_up_two[3] * a[3],
]

second_down_one = [
    delta_g[0] * a[0] + delta_g[1] * a[2],
    delta_g[0] * a[1] + delta_g[1] * a[3],
]
second_down_two = (
    second_down_one[0] * delta_g[0] + second_down_one[1] * delta_g[1]
)

second_go = [
    second_up_three[0] / second_down_two,
    second_up_three[1] / second_down_two,
    second_up_three[2] / second_down_two,
    second_up_three[3] / second_down_two,
]

a = [
    a[0] + first_go[0] - second_go[0],
    a[1] + first_go[1] - second_go[1],
    a[2] + first_go[2] - second_go[2],
    a[3] + first_go[3] - second_go[3],
]
print(f"A: {a[0], a[1]}/n   {a[2], a[3]}/n")
cur_point = final_result[0]
```

```
        cur_point_norm = find_norm(cur_point)
        cur_func_value = final_result[1]
        cur_gradient = find_gradient(cur_point, cur_func_value)
        cur_norm_grad = find_norm(cur_gradient)


if __name__ == "__main__":

    func_usage = 0

    Proizvodnie = 0.00001
    parametr_Svena = 0.1
    criteriy_okonchania = 0.00001
    tochnost_odn = 0.001
    h = Proizvodnie

    cur_point = [-1.2, 0]
    cur_norm_point = find_norm(cur_point)
    cur_func_value = cur_func(cur_point[0], cur_point[1])
    cur_gradient = find_gradient(cur_point, cur_func_value)
    cur_norm_gradient = find_norm(cur_gradient)

    Pearson3(cur_point, cur_func_value, cur_gradient, cur_norm_point,
cur_norm_gradient)
    print(f"Колличество вызова функции: {func_usage}")
```