Київський національний університет імені Тараса Шевченка
Факультет комп'ютерних наук та кібернетики
Кафедра дослідження операцій

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
за спеціальністю 113 «Прикладна математика»
на тему:

Методи оптимізації та їх практичне застосування

Студента 4 курсу
Бочка Владислава Володимировича

Науковий керівник:
доктор фізико-математичних наук,
доцент,
професор кафедри дослідження операцій
Самойленко Ігор Валерійович

Робота заслухана на засіданні кафедри дослідження операцій та рекомендована до захисту в ЕК, протокол №9 від 23 травня 2023 р.

Завідувач кафедри ДО проф. Іксанов О.М.

Київ – 2023

РЕФЕРАТ

Обсяг роботи 35 сторінок, 3 таблиці 7 використаних джерел.

Об’єктом дослідження є задача оптимізації:

$$\begin{cases} f(x) \to min \\ x \in X \end{cases}.$$

Метою роботи мінімізувати (чи максимізувати) цільову функцію $f(x)$ для $x \in X$. Для зручності, ми будемо розглядати методи оптимізації лише у розрізі мінімізації функції.

У даній роботі будуть розглянуті чотири основні методи чисельної оптимізації: градієнтний метод, метод Ньютона, метод проекції градієнта та метод спряжених напрямків. Окремо ми зупинимося на підвидах та особливостях кожного з вищеназваних методів, а також розповімо про їх слабкі та сильні сторони.

Також розглянемо загальні положення чисельних методів оптимізації.

ЗМІСТ

ВСТУП

Розглядається наступна задача оптимізації:

$$\begin{cases} f(x) \to min \\ x \in X \end{cases}.$$

Її метою є мінімізувати (чи максимізувати) цільову функцію $f(x)$ для $x \in X$. Для зручності, ми будемо розглядати методи оптимізації лише у розрізі мінімізації функції. Однак, слід пам'ятати, що задача максимізації легко перетворюється на задачу мінімізації після множення цільової функції на -1. У даній роботі будуть розглянуті чотири основні методи чисельної оптимізації: градієнтний метод, метод Ньютона, метод проекції градієнта та метод спряжених напрямків. Окремо ми зупинимося на підвидах та особливостях кожного з вищеназваних методів, а також розповімо про їх слабкі та сильні сторони.

Також варто розглянути й загальні положення чисельних методів оптимізації. Отож, чисельними методами оптимізації називають методи наближеного або точного розв'язання задач оптимізації, котрі зводяться до виконання певного скінченного числа елементарних операцій над числами.

Застосування чисельних методів оптимізації можливе лише після вирішення декількох кроків підготовки:

- створення математичної моделі об'єкта оптимізації;
- складання цільової функції оптимізації;
- опис областей визначення та методів розв'язання задач оптимізації.

Результатом застосування чисельних методів оптимізації є отримання таких значень x^*, для яких значення цільової функції буде мінімальним (максимальним) принаймні у якомусь околі точки x^*.

Чисельні методи оптимізації є дуже зручними для використання електронними обчислювальними машинами та є необхідними для тих задач, де знаходження оптимуму функції аналітичними методами ускладнене або повністю неможливе.

Однак, кожен з чисельних методів висуває певні вимоги до цільової функції та її обмежень. Тому перед тим, як використати той чи інший метод, завжди необхідно переконатись, чи задовольняє функція та її обмеження вимоги методу. У випадку, коли функція та її обмеження не задовольняють вимоги методу, метод може втратити збіжність.

Також слід пам’ятати, що при практичній реалізації методів чисельної оптимізації вони можуть втрачати точність через похибки машинної точності, а також похибки при чисельному знаходженні похідних. Через це чисельні методи оптимізації можуть потребувати значно більшу кількість ітерацій для знаходження оптимуму, або взагалі працювати непередбачувано через недостатню точність.

Всі методи чисельної оптимізації, що будуть розглянуті нижче, належать до групи методів спуску. Це означає, що їх крок має наступний вигляд:

$$x^{k+1} = x^k + \alpha_k h^k.$$

Приведені у цій роботі методи оптимізації можуть бути легко використані для будь-якого простору скінченної вимірності.

РОЗДІЛ 1. ТЕОРЕТИЧНА ЧАСТИНА

1. Градієнтний метод

Градієнтний метод базується на загальній схемі методів спуску: $x^{k+1} = x^k + \alpha_k h^k$. Ключовою особливістю градієнтного методу є вибір h^k: $h^k = -f'(x^k)$.

Таким чином, градієнтний метод матиме наступний алгоритм:

1. Обираємо довільну початкову точку x^0 та встановлюємо k=0.

2. Обираємо значення α_k методом дроблення або методом найшвидшого спуску.

3. Знаходимо $x^{k+1} = x^k - \alpha_k f'(x^k)$

4. Перевіряємо умову зупинки: $f'(x^{k+1}) \approx 0$. Якщо вона виконується, отримана точка є точкою мінімуму функції. Якщо вона не виконується, то збільшуємо k на 1 та переходимо до пункту 2.

Достатніми умовами збіжності градієнтного методу є наступні:

- функція $f(x)$ диференційована;
- функція $f(x)$ обмежена знизу на $\mathbb{R}^n$;
- градієнт функції $f(x)$ задовольняє умові Ліпшица.

Градієнтний метод також буде збігатись зі швидкістю геометричної прогресії, якщо виконуються наступні умови:

- функція $f(x)$ двічі диференційована;
- функція $f(x)$ сильно опукла на $\mathbb{R}^n$;
- матриця других похідних при $D > 0, x, h \in \mathbb{R}^n$ задовольняє умову $(f''(x)h, h) \leq D\left|\left|h\right|\right|^2$.

Повільна збіжність градієнтних методів не дозволяє вирішувати з їх допомогою складні задачі мінімізації. Градієнтні методи використовуються в комбінації з іншими, з більш високою швидкістю збіжності, на початковій стадії вирішення задачі, коли точка x^k перебуває далеко від мінімуму і крок уздовж антиградієнта дозволяє досягти істотного зменшення функції.

Ще одним недоліком методу градієнтного спуску є те, що за допомогою нього ми гарантовано знаходимо локальний мінімум, але не можемо бути впевненими у знаходженні глобального мінімуму.

Безсумнівною перевагою градієнтних методів є їх простота і можливість використовувати їх для мінімізації дуже різних за характером функцій.

Градієнтний метод є одним з фундаментальних алгоритмів оптимізації, який використовується в багатьох задачах машинного навчання. Він має велику важливість для ефективного навчання моделей з великою кількістю параметрів, таких як нейронні мережі.

Ось декілька ключових причин, чому градієнтний метод важливий для машинного навчання:

1. Оптимізація параметрів моделі: У задачах машинного навчання нам часто потрібно знайти набір параметрів моделі, які мінімізують функцію втрати або максимізують функцію вигоди. Градієнтний метод дозволяє нам оновлювати параметри моделі, керуючись напрямком та величиною найшвидшого зниження функції втрати.

2. Швидкість навчання: Градієнтний метод допомагає знайти оптимальні параметри моделі швидше, ніж прості методи, такі як випадковий пошук або сітчатий пошук. Використовуючи інформацію про градієнт функції втрати, алгоритм може швидко збігатися до локального оптимуму або прийнятного рішення.

3. Стійкість до шуму: Градієнтний метод може бути стійким до шуму в даних. Він враховує загальну тенденцію градієнту, а не окремі випадкові аномалії. Це дозволяє моделі краще адаптуватися до шумних даних та зберігати загальну структуру задачі.

4. Розширення на глибоке навчання: Градієнтний метод є основою для оптимізації глибоких нейронних мереж. Великі нейронні мережі мають мільйони або навіть мільярди параметрів, і градієнтний метод допомагає навчити ці параметри на величезних обсягах даних.

5. Варіації градієнтного методу: Градієнтний метод має багато варіацій, таких як стохастичний градієнтний спуск (SGD), метод Адама (Adam), RMSProp та інші. Ці варіації дозволяють удосконалити швидкість збіжності, регулювання швидкості навчання, обробку рідкісних даних та інші аспекти оптимізації.

Градієнтний метод є потужним інструментом, який допомагає нам покращувати якість та швидкість навчання моделей машинного навчання. Він забезпечує оптимальне оновлення параметрів моделі на основі градієнту функції втрати, що дозволяє досягати кращої точності та знижувати помилки передбачень.

1.1 **Градієнтний метод при виборі α методом дроблення**

При виборі альфа методом дроблення велику роль відіграє значення коефіцієнта дроблення β. В нашій роботі значення цього коефіцієнту становитиме 0.5. Сенс методу дроблення полягає в тому, аби поступово зменшуючи значення α_k підібрати таке α_k, за якого функція в наступній точці градієнтного методу була меншою, аніж у поточній.

Алгоритм вибору альфа методом дроблення є наступним:

1. Встановлення $\alpha_k = 1$.

2. Визначення $h^k = -f'(x^k)$

3. Перевіряємо чи $f(x^k) \leq f(x^k + \alpha_k h^k)$. Якщо нерівність виконується, альфа вважається знайденою. Якщо ні, то $\alpha_k = \alpha_k \cdot \beta$ й повторюємо перевірку.

1.2. Градієнтний метод при виборі α методом найшвидшого спуску

Вибір α методом найшвидшого спуску є найкращим з точки зору вибору оптимального альфа й за ідеальних умов показує найменшу кількість ітерацій методу градієнтного спуску. Сутність методу найшвидшого спуску в тому, що ми мінімізуємо по α_k значення функції $f(x^k - \alpha_k f'(x^k))$, за умови того, що x^k та $f'(x^k)$ є відомими константами. Таким чином, вибір альфа методом найшвидшого спуску зводиться до задачі одномірної оптимізації будь-яким довільним способом.

2. Метод Ньютона

Метод Ньютона, як і метод градієнтного спуску, належить до сімейства методів спуску, крок яких має вигляд $x^{k+1} = x^k + \alpha_k h^k$. Ключовою особливістю методу Ньютона є вибір h^k: $h^k = -f''(x^k)^{-1} f'(x^k)$.

Таким чином, метод Ньютона матиме наступний алгоритм:

1. Обираємо довільну початкову точку x^0 та встановлюємо k=0.

2. Обираємо значення α_k методом дроблення або класичним методом.

3. Знаходимо $x^{k+1} = x^k - \alpha_k f''(x^k)^{-1} f'(x^k)$

4. Перевіряємо умову зупинки: $f'(x^{k+1}) \approx 0$. Якщо вона виконується, отримана точка є точкою мінімуму функції. Якщо вона не виконується, то збільшуємо k на 1 та переходимо до пункту 2.

Як ми бачимо, алгоритм методу Ньютона досить схожий на алгоритм методу градієнтного спуску, але суттєво відрізняється необхідністю знаходити матрицю других похідних та обернену до неї. Тобто, відбувається квадратична апроксимація, що дозволяє сподіватись на меншу кількість ітерацій, аніж у методі градієнтного спуску.

Достатніми умовами для збіжності методу Ньютона є наступні:

- функція $f(x)$ двічі диференційована;
- функція $f(x)$ сильно опукла з константою $\Theta>0$ на $\mathbb{R}^n$;
- функція $f(x)$ для $x, \tilde{x} \in \mathbb{R}^n, M > 0$ задовольняє умову $\left||f''(x) - f''(\tilde{x})|\right| \leq M\left||x - \tilde{x}|\right|$;
- початкова точка x^0 така, що $\left||f'(x^0)|\right| \leq \frac{8\Theta^2}{M}$.

Лише за таких умов збіжність методу Ньютона є гарантованою. Окрім того, збіжність методу матиме квадратичну швидкість. Слід звернути увагу, що на відміну від методу градієнтного спуску, метод Ньютона накладає обмеження не лише на цільову функцію, а й на вибір початкового наближення.

Іншим недоліком методу Ньютона є великий обсяг обчислень на кожному кроці (потрібно щоразу обчислювати та обертати матрицю других похідних цільової функції, що потребує значних обчислень).

2.1. Вибір α для класичного методу Ньютона

Класичний вибір альфа є найпростішим: альфа встановлюється певною додатною константою, зазвичай, одиницею. Це збільшує необхідну кількість ітерацій для методу Ньютона, однак дозволяє не витрачати ресурси електронної обчислювальної машини на пошук альфа.

2.2. Вибір α для узагальненого методу Ньютона

Вибір α для узагальненого методу Ньютона практично аналогічний подібній процедурі для методу дроблення методу градієнтного спуску. При виборі альфа методом дроблення велику роль відіграє значення коефіцієнта дроблення β. В нашій роботі значення цього коефіцієнту становитиме 0.5. Сенс методу дроблення полягає в тому, аби поступово зменшуючи значення α_k підібрати таке α_k, за якого функція в наступній точці методу Ньютона була меншою, аніж у поточній.

Алгоритм вибору альфа методом дроблення є наступним:

1. Встановлення $\alpha_k = 1$.

2. Визначення $h^k = -f''(x)f'(x^k)$

3. Перевіряємо чи $f(x^k) \leq f(x^k + \alpha_k h^k)$. Якщо нерівність виконується, альфа вважається знайденою. Якщо ні, то $\alpha_k = \alpha_k \cdot \beta$ й повторюємо перевірку.

3. Метод проекції градієнта

Особливістю цього методу є те, що ми шукаємо мінімум не на $x \in \mathbb{R}^n$, а на $x \in X$, де X замкнена опукла множина в $\mathbb{R}^n$. За винятком цього, метод проекції градієнта збігається з градієнтним методом та має ітеративну формулу $\vec{x}_{k+1} = projection(\vec{x}_k + \alpha_k \vec{h}_k)$, де $\vec{h}_k = -f'(\vec{x}_k)$. Тут $projection$ є функцією проекції точки $\vec{x}_k + \alpha_k \vec{h}_k$ на множину X.

Таким чином метод проекції градієнту матиме наступний алгоритм:

1. Обираємо довільну початкову точку x^0 та встановлюємо k=0.

2. Обираємо значення α_k методом дроблення або методом найшвидшого спуску.

3. Знаходимо $\vec{x}_{k+1} = projection(\vec{x}_k + \alpha_k \vec{h}_k)$

4. Перевіряємо умову зупинки: $f'(x^{k+1}) \approx 0$ або значення функції в поточній та попередній точках майже не відрізняються. Якщо вона виконується, отримана точка є точкою мінімуму функції. Якщо вона не виконується, то збільшуємо k на 1 та переходимо до пункту 2.

Достатніми умовами збіжності методу проекції градієнта є наступні:

- $X \in \mathbb{R}^n$ опукла і замкнена множина
- функція $f(x)$ диференційована на X;
- функція $f(x)$ сильно опукла з константою $\Theta>0$ на $\mathbb{R}^n$;
- градієнт функції $f(x)$ задовольняє умову Ліпшіца.

За виконання таких умов гарантується збіжність методу проекції градієнта зі швидкістю геометричної прогресії.

Сильні та слабкі сторони цього методу відповідають аналогічним сторонам градієнтного методу. В параграфах 3.1-3.5 будуть коротко розглянуті деякі розповсюджені види множини X та розглянуті методи проекції на неї.

3.1. Проекція на кулю

Розглядаємо випадок, коли множина X—куля. Тоді ця множина буде задана наступним чином:

$$X = \{x \in \mathbb{R}^n : ||x - x^0|| \leq r\}.$$

Тут x^0 центр кулі, а r радіус кулі. Тоді проекція на кулю буде відбуватись наступним чином:

$$projection(x) = x^0 + \frac{x - x^0}{||x - x^0||} r.$$

3.2. Проекція на координатний паралелепіпед

Розглядаємо випадок, коли множина X—координатний паралелепіпед. Тоді ця множина буде задана наступним чином:

$$X = \{x \in \mathbb{R}^n : b_j \leq x_j \leq c_j, j = 1, \dots, n\}.$$

Тут b координати початку координатного паралелепіпеду, а c координати кінця координатного паралелепіпеду. Тоді проекція на координатний паралелепіпед буде відбуватись наступним чином:

$$projection(x)_j = \begin{cases} b_j, x_j < b_j \\ x_j, b \leq x_j \leq c_j \\ c_j, x_j > c_j \end{cases}.$$

3.3. Проекція на невід'ємний ортант

Розглядаємо випадок, коли множина X—невід'ємний ортант. Тоді ця множина буде задана наступним чином:

$$X = \{x \in \mathbb{R}^n : x_j \geq 0, j = 1, \dots, n\}.$$

Тоді проекція на координатний паралелепіпед буде відбуватись наступним чином:

$$projection(x) = (\max(0, x_1), \max(0, x_2), \dots, \max(0, a_n)).$$

3.4. Проекція на гіперплощину

Розглядаємо випадок, коли множина X—гіперплощина. Тоді ця множина буде задана наступним чином:

$$X = \{x \in \mathbb{R}^n : (p, x) = \beta, j = 1, \dots, n\}.$$

Тут p це ненульовий вектор розмірності n, а β дійсне число. Тоді проекція на координатний паралелепіпед буде відбуватись наступним чином:

$$projection(x) = x + \big(\beta - (p, x)\big) \cdot \frac{p}{\big||p|\big|^2}.$$

3.5. Проекція на півпростір

Розглядаємо випадок, коли множина X—півпростір. Тоді ця множина буде задана наступним чином:

$$X = \{x \in \mathbb{R}^n : (p, x) \geq \beta, j = 1, \dots, n\}.$$

Тут p це ненульовий вектор розмірності n, а β дійсне число. Тоді проекція на координатний паралелепіпед буде відбуватись наступним чином:

$$projection(x) = x + \max\Big(0, \big(\beta - (p, x)\big)\Big) \cdot \frac{p}{\big||p|\big|^2}.$$

4. Метод спряжених напрямків

Метод спряжених напрямків (також відомий як метод спряжених градієнтів) є досить цікавою модифікацією градієнтного методу. Ідеї методу спряжених напрямків базуються на мінімізації квадратичної функції за скінченну кількість кроків, однак є адаптація й для неквадратичної функції. За своєю сутністю, метод спряжених напрямків мінімізує функцію лише по одній координаті з x_j за одну ітерацію. Слід пам'ятати, що метод спряжених напрямків також має загальну формулу $x^{k+1} = x^k + \alpha_k h^k$.

Таким чином, метод спряжених напрямків матиме наступний алгоритм:

1. Обираємо довільну початкову точку x^0 та встановлюємо *k=0*.

2. Обчислюємо $h^0 = -f'(x^0)$

3. Обираємо значення α_k методом дроблення спуску.

5. Знаходимо $x^{k+1} = x^k + \alpha_k h^k$

6. Перевіряємо умову зупинки: $f'(x^{k+1}) \approx 0$. Якщо вона виконується, отримана точка є точкою мінімуму функції. Якщо вона не виконується, то збільшуємо k на 1.

7. Знаходимо значення β_{k-1} заданим методом.

8. Знаходимо $h^k = -f'(x^k) + \beta_{k-1} h^{k-1}$ та переходимо до кроку 3.

Достатніми умовами збіжності методу спряжених градієнтів є наступні:

- функція $f(x)$ диференційована;
- функція $f(x)$ обмежена знизу на $\mathbb{R}^n$;

градієнт функції $f(x)$ задовольняє умові Ліпшица.

4.1. Метод спряжених напрямків для квадратичної функції

Метод спряжених напрямків для квадратичної функції завжди знаходить її оптимум за кількість кроків, що не перевищують розмірність простору. Однак, на практиці, через похибки машинної точності це може потребувати відчутно більше кроків.

Особливістю методу спряжених напрямків для квадратичної функції є спосіб знаходження β:

$$\beta_k = \frac{\langle f'(x^{k+1}), f''(x^{k+1}) h^k \rangle}{\langle h^k, f''(x^{k+1}) h^k \rangle}.$$

4.2. Метод спряжених напрямків для неквадратичної функції

Метод спряжених напрямків для неквадратичної функції не має скінченної кількості кроків.

Особливістю методу спряжених напрямків для квадратичної функції є спосіб знаходження β:

$$\beta_k = \begin{cases} \dfrac{\langle f'(x^{k+1}), f'(x^{k+1}) - f'(x^k)\rangle}{\left||f'(x^k)|\right|^2}, \left[\dfrac{k}{n}\right] \neq 0 \\ 0, \left[\dfrac{k}{n}\right] = 0 \end{cases}.$$

Більше інформації надано у додатку А.

РОЗДІЛ 2. ПРАКТИЧНА ЧАСТИНА

1 Результати дослідження

1.1Результати для функції $f(x) = x_1^2 - x_2 + 2x_2^2 + 3$

Результати дослідження різних методів оптимізації для функції $f(x) = x_1^2 - x_2 + 2x_2^2 + 3$ вказано у таблиці 1.

Таблиця 1. Результати для $f(x) = x_1^2 - x_2 + 2x_2^2 + 3$

method	alpha	x	iteration	f(x)	success	projection	beta
gradient descent	steepest descent	[0.00036522 0.2498775]	7	2.875000163	True		
gradient descent	crushing	[-8.8817842e-16 2.5000000e-01]	2	2.875	True		
newton method	classic	[2.23154828e-10 2.50000000e-01]	1	2.875	True		
newton method	crushing	[2.23154828e-10 2.50000000e-01]	1	2.875	True		
projection method	steepest descent	[0.00379037 0.24873654]	5	2.87501756	True	sphere	
projection method	steepest descent	[0. 0.25000001]	2	2.875	True	parallelepiped	
projection method	steepest descent	[0. 0.25000001]	2	2.875	True	non-negative octant	
projection method	steepest descent	[6.5 3.5]	2	66.25	True	hyperplane	
projection method	steepest descent	[6.5 3.5]	2	66.25	True	semispace	
projection method	crushing	[-8.8817842e-16 2.5000000e-01]	2	2.875	True	sphere	

projection method	crushing	[5.46229728e-14 2.50000000e-01]	2	2.875	True	parallelepiped	
projection method	crushing	[5.46229728e-14 2.50000000e-01]	2	2.875	True	non-negative octant	
projection method	crushing	[6.50976562 3.49023438]	8	66.2502861	True	hyperplane	
projection method	crushing	[6.50976562 3.49023438]	8	66.2502861	True	semispace	
conjugate directions	crushing	[7.29886784e-08 2.50000000e-01]	65	2.875	True		squared
conjugate directions	crushing	[1.16161704e-10 2.50000000e-01]	17	2.875	True		non-squared

Як ми можемо помітити, найкращий результат за кількістю ітерацій мав метод Ньютона. Проте, інші методи також змогли успішно відшукати мінімум функції.

1.2. Результати для функції $f(x) = x_1^2 + 18x_2^2 + 0.01x_1x_2 + x_1 - x_2$

Результати дослідження різних методів оптимізації для функції $f(x) = x_1^2 + 18x_2^2 + 0.01x_1x_2 + x_1 - x_2$ вказано у таблиці 2.

Таблиця 2. Результати для $f(x) = x_1^2 + 18x_2^2 + 0.01x_1x_2 + x_1 - x_2$

method	alpha	x	iteration	f(x)	success	projection	beta
gradient descent	steepest descent	[-0.49997361 0.02791587]	9	-0.26403	True		
gradient descent	crushing	[-0.49969406 0.02790714]	43	-0.26403	True		
newton method	classic	[-0.50013959 0.02791671]	1	-0.26403	True		
newton method	crushing	[-0.50013959 0.02791671]	1	-0.26403	True		
projection method	steepest descent	[-0.5001603 0.02791671]	4	-0.26403	True	sphere	
projection method	steepest descent	[0. 0.02777778]	3	-0.01389	True	parallelepiped	
projection method	steepest descent	[0. 0.02777778]	3	-0.01389	True	non-negative octant	
projection method	steepest descent	[9.42338071 0.57661929]	2	103.686	True	hyperplane	
projection method	steepest descent	[9.42338059 0.57661941]	2	103.686	True	semispace	
projection method	crushing	[-0.44273235 0.0340914]	18	-0.26004	True	sphere	
projection method	crushing	[0. 0.01574595]	1001	-0.01128	False	parallelepiped	
projection method	crushing	[0. 0.01574595]	1001	-0.01128	False	non-negative octant	
projection method	crushing	[9.42446722 0.57553278]	8	103.686	True	hyperplane	

projection method	crushing	[9.42446722 0.57553278]	8	103.686	True	semispace	
conjugate directions	crushing	[-0.50018429 0.02789767]	10	-0.26403	True		squared
conjugate directions	crushing	[-0.49984363 0.02793186]	40	-0.26403	True		non-squared

Як ми можемо помітити, найкращий результат за кількістю ітерацій мав метод Ньютона. Однак також можемо помітити, що пошук мінімуму методом проекції при знаходженні альфа методом дроблення, виявився неуспішним (тобто, перевищена максимально припустима кількість ітерацій 1000). З огляду на те, що метод найшвидшого спуску для пошуку альфа завжди показував кращу продуктивність аніж знаходження альфа методом дроблення, можна зробити висновок, про те, що метод найшвидшого спуску дійсно показує найкращі можливі результати.

1.3. Результати для функції $f(x) = (x_1 - 2)^2 + (x_2 + 1)^2 + 15$

Результати дослідження різних методів оптимізації для функції $f(x) = (x_1 - 2)^2 + (x_2 + 1)^2 + 15$ вказано у таблиці 3.

Таблиця 3. Результати для $f(x) = (x_1 - 2)^2 + (x_2 + 1)^2 + 15$

method	alpha	x	iteration	f(x)	success	projection	beta
gradient descent	steepest descent	[1.99999999 -0.99999998]	1	15	True		
gradient descent	crushing	[2. -1.]	2	15	True		
newton method	classic	[2. -1.]	1	15	True		
newton method	crushing	[2. -1.]	1	15	True		
projection method	steepest descent	[1.99999999 -0.99999998]	1	15	True	sphere	
projection method	steepest descent	[2.00000001 0.]	2	16	True	parallelepiped	
projection method	steepest descent	[2.00000001 0.]	2	16	True	non-negative octant	
projection method	steepest descent	[6.49999995 3.50000005]	2	55.5	True	hyperplane	
projection method	steepest descent	[6.49999995 3.50000005]	2	55.5	True	semispace	
projection method	crushing	[3. -3.]	1	20	True	sphere	
projection method	crushing	[2. 0.]	3	16	True	parallelepiped	

projection method	crushing	[2. 0.]	3	16	True	non-negative octant	
projection method	crushing	[6.5 3.5]	3	55.5	True	hyperplane	
projection method	crushing	[6.5 3.5]	3	55.5	True	semispace	
conjugate directions	crushing	[2. -1.]	1	15	True		squared
conjugate directions	crushing	[2. -1.]	1	15	True		non-squared

Ця функція квадратична й досить проста та має очевидний мінімум. Завдяки цьому, всі методи успішно впорались зі знаходженням мінімуму за невелику кількість кроків.

1.4. Аналіз отриманих результатів

Виходячи з результатів, отриманих у попередніх пунктах цього параграфу, проведемо детальний аналіз даних та синтезуємо певні висновки, що базуються на них.

Спершу, розберемо результати роботи метод градієнтного спуску та його модифікації. Згідно з теоретичними викладками, метод градієнтного спуску з пошуком альфа методом дроблення має мати більшу кількість ітерацій, аніж метод градієнтного спуску з пошуком альфа методом найшвидшого спуску. Дані з таблиці 2 та таблиці 3 підтверджують це, однак результати, показані у таблиці 1 не відповідають цьому. Нами було розглянуто дві можливі причини отримання таких результатів. Перша гіпотеза полягала в тому, що неочікувані результати були отримані унаслідок помилок машинної точності. Друга гіпотеза полягала в тому, що існує кілька пар точок (x_1,x_2), у яких функція $f(x) = x_1^2 - x_2 + 2x_2^2 + 3$ досягає свого мінімуму. Після додаткового аналізу результатів, нами було виявлено, що причини невідповідності отриманих даних очікуваним, полягає саме у другій гіпотезі. Таким чином, ми можемо дійти висновку, що застосування методу градієнтного спуску з однієї тієї самої початкової точки може привести до різних локальних мінімумів функції, якщо обирати різні способи знаходження

альфа. Саме цей висновок є основним результатом проведення тестування методу градієнтного спуску.

Другими результатами, що ми розглянемо, будуть результати роботи методу Ньютона. Як ми можемо побачити з таблиці 1, таблиці 2 та таблиці 3— цей метод дозволив досягти мінімуму за найменшу кількість ітерацій, що відповідає теоретичним викладкам. Проте, слід не забувати, що це досягається за рахунок високих вимог до цільової функції та великого обсягу додаткових обчислень. Різниці між вибором альфа класичним методом чи методом дроблення у розглянутих прикладах виявлено не було. Проте, слід зазначити, що більш безпечним є вибір альфа методом дроблення.

Третіми результатами, котрі будуть розглядатись у цьому пункті, будуть результати роботи методу проекції. Унікальність цього методу не дозволяє його порівнювати з іншими методами оптимізації, що розглянуті у цій роботі. Проте, практичну користь цього методу не можна переоцінити. Він забезпечує мінімізацію на обмеженому просторі, що відповідає більшості задач з фізики у реальному світі. Цікавим спостереженням є те, що для різних функцій метод проекції на різні обмежені простори дає різні порядки кількості ітерацій, таким чином, ми не можемо сказати, що на якомусь обмеженому просторі мінімум функції завжди знаходиться швидше, ніж на інших обмежених просторах. Важливим є також той факт, що метод вибору альфа методом найшвидшого спуску не лише зменшує кількість ітерацій методу проекції, а й інколи забезпечують його збіжність там, де збіжність при виборі альфа методом дроблення не забезпечує збіжності.

Четвертими результатами, котрі будуть розглядатись у цьому пункті, будуть результати методу спряжених напрямків. В роботі розглядаються два типи цього методу оптимізації: для квадратичної та неквадратичної функції. Як ми бачимо з таблиці 3, для квадратичної функції цей метод дозволяє знайти мінімум за один крок. Це відповідає даним про те, що метод спряжених

напрямків дозволяє знайти мінімум квадратичної функції не більше ніж за кількість кроків, рівну розмірності простору. Також експерименти, результати яких можна побачити у таблиці 1 та таблиці 2, показали, що для неквадратичної функції, в залежності від конкретної функції, може бути кращим використовувати як метод спряжених напрямків для квадратичної функції, так і метод спряжених напрямків для неквадратичної функції.

Таким чином, можна підбити такі результати аналізу отриманих результатів:

- для методу градієнтного спуску найкращою виявилась модифікація з вибором альфа методом найшвидшого спуску;
- метод Ньютона забезпечує найменшу кількість ітерацій для довільної функції (за теоретичними даними, програє у цьому параметрі методу спряжених напрямків для квадратичної функції при застосуванні до квадратичної функції, однак за емпіричними результатами результативність показана однаковою);
- кількість ітерацій для методу проекції залежить індивідуально від функції та обмеженого простору;
- для методу проекції набагато краще використовувати модифікацію з пошуком альфа методом найшвидшого спуску;
- для методу спряжених напрямків для квадратичної функції найкраще використовувати модифікацію методу для квадратичних функцій, проте для неквадратичних функцій можна використовувати як модифікацію методу для квадратичних функцій, так і модифікацію методу для неквадратичних функцій—ефективність модифікації залежить від конкретної функції.

1.5. Похибки машинної точності

Машинна точність є однією з основних проблем, з якими стикаються розробники програмного забезпечення і науковці, що працюють з

обчислювальними системами. Ця проблема полягає в тому, що комп'ютери можуть зберігати та оброблювати тільки певний діапазон чисел з обмеженою точністю, що може призводити до похибок в обчисленнях.

Однією з основних причин похибок машинної точності є використання десяткових дробів у двійковій системі обчислення. Наприклад, десяткове число 0.1 не може бути точно представлене у двійковій системі, тому що вона не має скінченного десяткового розкладу. Це може призвести до неточностей у обчисленнях, особливо коли використовуються довгі послідовності обчислень з числами з плаваючою комою.

Іншою причиною похибок машинної точності є округлення чисел. Комп'ютери можуть зберігати тільки обмежений набір цілих та дробових чисел, тому вони повинні округлювати значення, які не можуть бути точно представлені в цьому форматі. Це може призводити до невеликих похибок у кінцевому результаті обчислень.

Також, похибки машинної точності можуть виникати через порядок обчислень та використання невірного алгоритму. Деякі алгоритми можуть бути менш точними або працювати повільніше на обмежених ресурсах, що може призвести до похибок.

Щоб уникнути цих похибок, розробники можуть використовувати більш точні формати чисел з плаваючою комою, такі як double або decimal, або використовувати інші методи для збільшення точності обчислень. Також важливо ретельно контролювати порядок обчислень та використовувати ефективні алгоритми, що можуть зменшити кількість обчислень та зменшити ймовірність виникнення похибок. Для виконання точних обчислень можуть бути використані бібліотеки з високою точністю, такі як GNU Multiple Precision Arithmetic Library, яка надає можливість виконувати операції з великими числами з високою точністю.

Щоб зменшити похибки машинної точності, розробники також можуть використовувати методи нумеричного аналізу, такі як методи інтерполяції та

апроксимації, що дозволяють наближено розв'язувати складні математичні задачі з високою точністю.

Узагальнюючи, похибки машинної точності є важливою проблемою, з якою стикаються розробники програмного забезпечення та науковці, що працюють з обчислювальними системами. Ці похибки можуть призводити до неточних результатів обчислень та впливати на роботу програм. Однак, існують різні методи та бібліотеки, які можуть допомогти зменшити похибки машинної точності та забезпечити більш точні результати обчислень.

Більше інформації надано у додатку А.

ВИСНОВКИ

Провівши дослідження трьох різних функцій за допомогою різноманітних варіацій методів чисельної оптимізації ми можемо зробити певні висновки. . У даній роботі нами розглянуто чотири основні методи чисельної оптимізації: градієнтний метод, метод Ньютона, метод проекції градієнта та метод спряжених напрямків. Також ми детально зупинились на підвидах та особливостях кожного з вищеназваних методів і розповіли про їх слабкі та сильні сторони.

Найкраще себе показав метод Ньютона, що збігається з теоретичним викладками. Найкращим методом пошуку альфа виявився метод найшвидшого спуску, що також відповідає теоретичним засадам.

Основними особистими спостереженнями, виявились наступні:

- найкращою модифікацією методу градієнтного спуску є модифікація з вибором альфа методом найшвидшого спуску;
- кількість ітерацій для методу проекції залежить індивідуально від функції та обмеженого простору;
- для методу проекції набагато краще використовувати модифікацію з пошуком альфа методом найшвидшого спуску;

- для методу спряжених напрямків для квадратичної функції найкраще використовувати модифікацію методу для квадратичних функцій

- для методу спряжених напрямків для неквадратичних функцій можна використовувати як модифікацію методу для квадратичних функцій, так і модифікацію методу для неквадратичних функцій— ефективність модифікації залежить від конкретної функції.

Таким чином, ми успішно перевірили дієвість чотирьох популярних методів чисельної оптимізації у різноманітних модифікаціях та зробили відповідні висновки. Ці висновки можуть допомогти краще використовувати вищеописані методи чисельної оптимізації.

Окремо слід зауважити, що за відсутності похибок машинної точності, робота методів мала б бути ще швидшою. Проте, на жаль, мова програмування Python 3 не підтримує у використаних нами математичних пакетах числа з плаваючою крапкою, котрі більші за розміром, аніж 64 біти. Однак, це можна змінити, використовуючи більш спеціалізовані бібліотеки—проте у більшості випадків це не є раціональним, адже значно уповільнить виконання однієї ітерації.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Arora, R. (2019). Optimization for Machine Learning. Cambridge University Press.
2. Bottou, L. (2018). Optimization Methods for Large-Scale Machine Learning. Now Publishers.
3. Cheney, M., & Kincaid, D. (2012). Numerical mathematics and computing. Cengage Learning.

4. Kaczmarz, S. (1937). Angenäherte Auflösung von Systemen linearer Gleichungen. Bulletin International de l'Académie Polonaise des Sciences et des Lettres. Classe des Sciences Mathématiques et Naturelles, (1), 355-357.
5. Polak, E., & Ribiere, G. (1969). Note sur la convergence de méthodes de directions conjuguées. Revue française d'informatique et de recherche opérationnelle. Série rouge, 3(16), 35-43.
6. Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. School of Computer Science, Carnegie Mellon University, Tech. Rep, 237
7. Quarteroni, A., Sacco, R., & Saleri, F. (2010). Numerical mathematics (Vol. 37). Springe

ДОДАТКИ

Додаток А

Градієнтний спуск:

```
"""Пошук мінімуму функції за допомогою методу градієнтного спуску"""
from derivative import Derivative
import numpy as np


class GradientDescent(Derivative):
    def __init__(self, function, start_point, alpha_method='steepest descent',
accuracy=1e-3):
        super().__init__(function, start_point, accuracy)
        self.alpha_method = alpha_method
        self.alpha = self.__get_alpha()

    def optimize(self):
        """Метод градієнтного спуску. Повертає точку мінімуму."""
        iteration = 0
        point = self.start_point
        while np.linalg.norm(self.get_first_derivative(point)) > self.accuracy:
            alpha = self.alpha(point)
            point -= alpha * self.get_first_derivative(point)
            iteration += 1
            if iteration > 1e3:
                return {'method': 'gradient descent', 'alpha': self.alpha_method, 'x': point,
'iteration': iteration,
                        'f(x)': self.lambda_function(point), 'success': False}
```

```
        return {'method': 'gradient descent', 'alpha': self.alpha_method, 'x': point,
'iteration': iteration,
                'f(x)': self.lambda_function(point), 'success': True}

    def __get_alpha(self):
        if self.alpha_method == 'steepest descent':
            return self.__steepest_descent
        elif self.alpha_method == 'crushing':
            return self.__crushing
        else:
            raise KeyError(f'Method of calculate alpha {self.alpha_method} is
unknown!')

    def __steepest_descent(self, point):
        """Пошук альфа методом найшвидшого спуску."""
        gradient = self.get_first_derivative(point)
        matrix_a = self.get_second_derivative(point)
        return np.dot(gradient, gradient) / np.dot(matrix_a @ gradient, gradient)

    def __crushing(self, point, beta=0.5):
        """Пошук альфа методом дроблення."""
        h = - self.get_first_derivative(point)
        alpha = 1
        while self.lambda_function(point + alpha * h) >=
self.lambda_function(point):
            alpha *= beta
        return alpha
```

Головна частина :

```
import pandas as pd
import numpy as np


from gradient_descent import GradientDescent
from newton_method import NewtonMethod
from projection_method import ProjectionMethod
from conjugate_directions import ConjugateDirections


pd.set_option('display.max_columns', 1000)


def test_gradient_descent(function, start_point=None):
```

```
    if start_point is None:
        start_point = np.array((1, 1), dtype=np.float64)
    test_results = []
    for alpha_method in ('steepest descent', 'crushing'):
        point = start_point.copy()
        gd = GradientDescent(function, point, alpha_method)
        test_results.append(gd.optimize())
    # print('Тестування методу градієнтного спуску:')
    # print(pd.DataFrame(test_results).T)
    return test_results


def test_newton_method(function, start_point=None):
    if start_point is None:
        start_point = np.array((1, 1), dtype=np.float64)
    test_results = []
    for alpha_method in ('classic', 'crushing'):
        point = start_point.copy()
        gd = NewtonMethod(function, point, alpha_method)
        test_results.append(gd.optimize())
    # print('Тестування методу Ньютона:')
    # print(pd.DataFrame(test_results).T)
    return test_results


def test_projection_method(function, start_point=None):
    if start_point is None:
        start_point = np.array((1, 1), dtype=np.float64)
    test_results = []
    for alpha_method in ('steepest descent', 'crushing'):
        for projection in (('sphere', (np.array((0, 0)), 10)),
                           ('parallelepiped', (np.array((0, 0)), np.array((10, 10)))),
                           ('non-negative octant', None), ('hyperplane', (np.array((1, 1)), 10)),
                           ('semispace', (np.array((1, 1)), 10))):
            point = start_point.copy()
            gd = ProjectionMethod(function, point, projection,
                                  alpha_method)
            test_results.append(gd.optimize())
    # print('Тестування методу проекції:')
    # print(pd.DataFrame(test_results).T)
    return test_results


def test_conjugate_directions(function, start_point=None):
```

```
    if start_point is None:
        start_point = np.array((1, 1), dtype=np.float64)
    test_results = []
    for alpha_method in ('crushing',):
        for beta_method in ('squared', 'non-squared'):
            point = start_point.copy()
            gd = ConjugateDirections(function, point, alpha_method,
                                     beta_method, 1e-3)
            test_results.append(gd.optimize())
    # print('Тестування методу спряжених напрямків:')
    # print(pd.DataFrame(test_results).T)
    return test_results


def test(function, start_point=None):
    test_results = test_gradient_descent(function, start_point)
    test_results += test_newton_method(function, start_point)
    test_results += test_projection_method(function, start_point)
    test_results += test_conjugate_directions(function, start_point)
    return test_results


if __name__ == '__main__':
    functions_list = (lambda x: x[0]**2 - x[1] + 2*x[1]**2 + 3,
                      lambda x: x[0]**2+18*x[1]**2+0.01*x[0]*x[1]+x[0]-x[1],
                      lambda x: (x[0]-2)**2 + (x[1]+1)**2+15)
    sheet_names = ('First', 'Second', 'Third')
    with pd.ExcelWriter('Results.xlsx') as writer:
        for l_function, sheet in zip(functions_list, sheet_names):
            result = pd.DataFrame(test(l_function))
            result.to_excel(writer, sheet, index=False)
```

Метод Ньютона:

```
from derivative import Derivative
import numpy as np


class NewtonMethod(Derivative):
    def __init__(self, function, start_point, alpha_method='crushing',  accuracy=1e-
3):
        super().__init__(function, start_point, accuracy)
        self.alpha_method = alpha_method
        self.alpha = self.__get_alpha()
```

```
    def optimize(self):
        """Метод Ньютона. Повертає точку мінімуму."""
        iteration = 0
        point = self.start_point
        while np.linalg.norm(self.get_first_derivative(point)) > self.accuracy:
            alpha = self.alpha(point)
            point += alpha * self.get_h(point)
            iteration += 1
            if iteration > 1e3:
                return {'method': 'newton method', 'alpha': self.alpha_method, 'x': point,
'iteration': iteration,
                        'f(x)': self.lambda_function(point), 'success': False}
        return {'method': 'newton method', 'alpha': self.alpha_method, 'x': point,
'iteration': iteration,
                'f(x)': self.lambda_function(point), 'success': True}

    def get_h(self, point):
        """Отримання h для методу Ньютона."""
        gradient = self.get_first_derivative(point)
        second_derivative_matrix = self.get_second_derivative(point)
        try:
            h = - np.linalg.inv(second_derivative_matrix) @ gradient
        except np.linalg.LinAlgError:
            h = - np.linalg.pinv(second_derivative_matrix) @ gradient
        return h

    def __get_alpha(self):
        if self.alpha_method == 'classic':
            return self.__classic
        elif self.alpha_method == 'crushing':
            return self.__crushing
        else:
            raise KeyError(f'Method of calculate alpha {self.alpha_method} is
unknown!')

    def __crushing(self, point, beta=0.5, epsilon=0.25):
        """Пошук альфа методом дроблення."""
        alpha = 1
        h = self.get_h(point)
        f_x = self.lambda_function(point + alpha * h)
        while f_x - self.lambda_function(point) > epsilon * alpha *
np.dot(self.get_first_derivative(point), h):
            alpha *= beta
```

```
        return alpha

    def __classic(self, point, alpha=1):
        return alpha
```

Метод проекції:

```
"""Пошук мінімуму функції за допомогою методу проекції"""
from derivative import Derivative
import numpy as np


class ProjectionMethod(Derivative):
    def __init__(self, function, start_point, projection, alpha_method='crushing',
accuracy=1e-3):
        super().__init__(function, start_point, accuracy)
        self.alpha_method = alpha_method
        self.alpha = self.__get_alpha()
        self.projection_type, self.projection_args = projection
        self.projection = self.__get_projection_method()

    def stop(self, old_point, new_point):
        """Возвращает True, если условие остановки выполнено, False иначе."""
        if np.linalg.norm(self.get_first_derivative(new_point)) <= self.accuracy:
            return True
        elif np.linalg.norm(new_point - old_point) <= self.accuracy:
            return True
        elif np.linalg.norm(self.lambda_function(new_point) -
self.lambda_function(old_point)) <= self.accuracy:
            return True
        else:
            return False

    def optimize(self):
        """Метод проекції. Повертає точку мінімуму."""
        point = self.start_point
        iteration = 1
        old_point = np.array(point)
        alpha = self.alpha(point)
        point = self.projection(point - alpha * self.get_first_derivative(point))
        while self.stop(old_point, point) is not True:
            alpha = self.alpha(point)
            old_point = np.array(point)
            point = self.projection(point - alpha * self.get_first_derivative(point))
```

```
            iteration += 1
            if iteration > 1e3:
                return {'method': 'projection method', 'alpha': self.alpha_method,
'projection': self.projection_type,
                        'x': point, 'iteration': iteration, 'f(x)': self.lambda_function(point),
'success': False}
        return {'method': 'projection method', 'alpha': self.alpha_method, 'projection':
self.projection_type,
                'x': point, 'iteration': iteration, 'f(x)': self.lambda_function(point),
'success': True}

    def __get_projection_method(self):
        if self.projection_type == 'sphere':
            return self.__get_sphere_projection
        elif self.projection_type == 'parallelepiped':
            return self.__get_parallelepiped_projection
        elif self.projection_type == 'non-negative octant':
            return self.__get_non_negative_octant
        elif self.projection_type == 'hyperplane':
            return self.__get_hyperplane_projection
        elif self.projection_type == 'semispace':
            return self.__get_semispace
        else:
            raise KeyError(f'Projection type {self.projection_type} is unknown!')

    def __get_sphere_projection(self, point):  # special for v.9
        """Возвращает проекцию точки point на шар с центром center и радиусом
radius."""
        center, radius = self.projection_args
        if np.linalg.norm(point - center) > radius:
            point = center + (point - center) / np.linalg.norm(point - center) * radius
        assert np.linalg.norm(point - center) <= radius + self.accuracy
        return point

    def __get_parallelepiped_projection(self, point):
        b, c = self.projection_args
        point[point < b] = b[point < b]
        point[point > c] = c[point > c]
        return point

    def __get_non_negative_octant(self, point):
        point[point < 0] = 0
        return point
```

```
    def __get_hyperplane_projection(self, point):  # special for v.1
        """Возвращает проекцию точки point на плоскость (p,x)=beta."""
        p, beta = self.projection_args
        point = point + (beta - np.dot(p, point)) * p / np.linalg.norm(p) ** 2
        return point

    def __get_semispace(self, point):
        p, beta = self.projection_args
        point = point + max(0, (beta - np.dot(p, point))) * p / np.linalg.norm(p) ** 2
        return point

    def __get_alpha(self):
        if self.alpha_method == 'classic':
            return self.__classic
        elif self.alpha_method == 'steepest descent':
            return self.__steepest_descent
        elif self.alpha_method == 'crushing':
            return self.__crushing
        else:
            raise KeyError(f'Method of calculate alpha {self.alpha_method} is
unknown!')

    def __steepest_descent(self, point):
        """Пошук альфа методом найшвидшого спуску."""
        gradient = self.get_first_derivative(point)
        matrix_a = self.get_second_derivative(point)
        return np.dot(gradient, gradient) / np.dot(matrix_a @ gradient, gradient)

    def __crushing(self, point, beta=0.5):
        """Пошук альфа методом дроблення."""
        h = - self.get_first_derivative(point)
        alpha = 1
        while self.lambda_function(point + alpha * h) >=
self.lambda_function(point):
            alpha *= beta
        return alpha

    def __classic(self, point, alpha=1):
        return alpha
```

Сполучені напрямки:

```
from derivative import Derivative
import numpy as np
```

```
import scipy as sp


class ConjugateDirections(Derivative):
    def __init__(self, function, start_point, alpha_method='steepest descent',
beta_method='non-square', accuracy=1e-3):
        super().__init__(function, start_point, accuracy)
        self.alpha_method = alpha_method
        self.alpha = self.__get_alpha()
        self.beta_method = beta_method
        self.beta = self.__get_beta()

    def stop(self, old_point, new_point):
        """Возвращает True, если условие остановки выполнено, False иначе."""
        if np.linalg.norm(self.get_first_derivative(new_point)) <= self.accuracy:
            return True
        elif np.linalg.norm(new_point - old_point) <= self.accuracy:
            return True
        elif np.linalg.norm(self.lambda_function(new_point) -
self.lambda_function(old_point)) <= self.accuracy:
            return True
        else:
            return False

    def optimize(self):
        """Метод спряжених напрямків (градієнтів)."""
        point = self.start_point
        h = - self.get_first_derivative(point)
        alpha = self.alpha(point, h)
        old_point = point
        point = point + alpha * h
        iteration = 1
        while not self.stop(old_point, point):
            beta = self.beta(point, old_point, h, iteration)
            h = - self.get_first_derivative(point) + beta * h
            alpha = self.alpha(point, h)
            point = point + alpha * h
            iteration += 1
            if iteration > 1e3:
                return {'method': 'conjugate directions', 'alpha': self.alpha_method, 'beta':
self.beta_method,
                        'x': point, 'iteration': iteration, 'f(x)': self.lambda_function(point),
'success': False}
```

```
        return {'method': 'conjugate directions', 'alpha': self.alpha_method, 'beta':
self.beta_method,
                'x': point, 'iteration': iteration, 'f(x)': self.lambda_function(point),
'success': True}

    def __get_alpha(self):
        if self.alpha_method == 'steepest descent':
            return self.__steepest_descent
        elif self.alpha_method == 'crushing':
            return self.__crushing
        else:
            raise KeyError(f'Method of calculate alpha {self.alpha_method} is
unknown!')

    def __steepest_descent(self, h, point):
        """Пошук альфа методом найшвидшого спуску."""
        alpha = sp.optimize.minimize(lambda a:
                                     self.lambda_function(point + a * h), np.array((1,)),
method='powell').x[0]
        current = self.lambda_function(point + alpha * h)
        return alpha

    def __crushing(self, point, h, beta=0.5):
        """Пошук альфа методом дроблення."""
        alpha = 1
        while self.lambda_function(point + alpha * h) >=
self.lambda_function(point):
            alpha *= beta
        return alpha

    def __get_beta(self):
        if self.beta_method == 'squared':
            return self.__get_beta_squared
        elif self.beta_method == 'non-squared':
            return self.__get_beta_non_squared
        else:
            raise KeyError(f'Method of calculate beta {self.beta_method} is
unknown!')

    def __get_beta_squared(self, point, old_point, h, iteration):
        """Повертає значення бета для квадратичної функції."""
        if iteration % point.size == 0:
            beta = 0
        else:
```

```
            d_f = self.get_first_derivative(point)
            d2_f = self.get_second_derivative(point)
            beta = np.dot(d_f, d2_f @ h) / np.dot(h, d2_f @ h)
        return beta

    def __get_beta_non_squared(self, point, old_point, h, iteration):
        """Повертає значення бета для неквадратичної функції. Алгоритм
Флетчера-Рівса"""
        if iteration % point.size == 0:
            beta = 0
        else:
            d_f = self.get_first_derivative(point)
            d_f_old = self.get_first_derivative(old_point)
            beta = np.linalg.norm(d_f) ** 2 / np.linalg.norm(d_f_old) ** 2
        return beta
```

Диравційний метод:

```
import numpy as np


class Derivative:
    """Пошук повних та часткових похідних першого та другого порядків"""
    def __init__(self, function, start_point, accuracy=1e-3):
        self.lambda_function = function
        self.start_point = start_point
        self.dim = start_point.size
        self.accuracy = accuracy

    def get_first_partial_derivative(self, point, brew_number):
        """Отримання частинної першої похідної за змінною номер
brew_number."""
        h = np.zeros(point.size)
        h[brew_number] = self.accuracy
        while max(map(abs, (self.lambda_function(point + h),
self.lambda_function(point - h)))) > h[brew_number] * 1e6:
            h *= 10
        return (self.lambda_function(point + h) - self.lambda_function(point - h)) / (2
* h[brew_number])

    def get_second_partial_derivative(self, point, first_brew_number,
second_brew_number):
        """Отримання частинної другої похідної за змінними номер
first_brew_number, second_brew_number."""
```

```
        if first_brew_number == second_brew_number:
            h = np.zeros(self.dim)
            h[first_brew_number] = self.accuracy
            return (self.lambda_function(point + h) - 2 * self.lambda_function(point) +
                    self.lambda_function(point - h)) / self.accuracy ** 2
        else:
            h_s = np.zeros(self.dim)
            h_ns = np.zeros(self.dim)
            h_s[first_brew_number] = self.accuracy
            h_ns[first_brew_number] = self.accuracy
            h_s[second_brew_number] = self.accuracy
            h_ns[second_brew_number] = - self.accuracy
            return (self.lambda_function(point + h_s) - self.lambda_function(point +
h_ns) -
                    self.lambda_function(point - h_ns) + self.lambda_function(point -
h_s)) / (4 * self.accuracy ** 2)

    def get_first_derivative(self, point):
        """Отримання градієнту."""
        gradient = np.zeros(self.dim)
        for i in np.arange(self.dim):
            gradient[i] = self.get_first_partial_derivative(point, i)
        return gradient

    def get_second_derivative(self, point):
        """Отримання матриці других похідних."""
        matrix_a = np.zeros((self.dim, self.dim))
        for i in np.arange(self.dim):
            for j in np.arange(self.dim):
                matrix_a[i][j] = self.get_second_partial_derivative(point, i, j)
        return matrix_a
```