Київський національний університет
імені Тараса Шевченка
Факультет комп'ютерних наук та кібернетики
Кафедра обчислювальної математики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 113 Прикладна математика
на тему:

Алгоритми розв'язання варіаційної нерівності

Виконав студент 4-го курсу
Скибицький Нікіта Максимович _____

Науковий керівник:
доктор фіз.-мат. наук, професор
Семенов Володимир Вікторович _____

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Роботу розглянуто й допущено до захисту на засіданні кафедри обчислювальної математики
«___» _____ 202_ р.,
протокол № ___
Завідувач кафедри
С. І. Ляшко _____

Київ — 2020

РЕФЕРАТ

Обсяг роботи 56 сторінок, 1 ілюстрація, 9 таблиць, 14 джерел посилань, 1 додаток.

ВАРІАЦІЙНА НЕРІВНІСТЬ, ПРОЕКТИВНИЙ МЕТОД, МОНОТОННИЙ ОПЕРАТОР, ЕКСТРАГРАДІЄНТНИЙ МЕТОД.

Об’єктом роботи є процес розв’язування варіаційної нерівності за допомогою власноруч розробленого програмного засобу. Предметом роботи є власноруч розроблений програмний засіб розв’язування варіаційної нерівності.

Метою роботи є розробка програмного засібу розв’язування варіційної нерівності за допомогою алгоритмів Корпелевич, Tseng’a, Попова та Маліцького—Tam’a, тестування програмного засобу на модельних задачах, оцінка і аналіз результатів.

Методи розроблення: комп’ютерне моделювання, розробка програмного продукту. Інструменти розроблення: текстовий редактор Sublime Text 3, Jupyter Notebook, мова програмування Python.

Результати роботи: розроблено програмний засіб розв’язування варіційної нерівності за допомогою алгоритмів Корпелевич, Tseng’a, Попова та Маліцького—Tam’a, виконано тестування програмного засобу на модельних задачах, проведена оцінка і аналіз результатів

ЗМІСТ

1 Вступ

1.1 Варіаційна нерівність

Розглянемо абстрактний (тобто поки що не накладаємо на нього ніяких обмежень і не вимагаємо від нього ніяких властивостей) оператор A який діє на підмножині C гільбертового простору H.

Визначення (варіаційної нерівності)**.** Кажуть, що для точки $x \in C$ виконується *варіаційна нерівність* якщо

$$\langle A(x), y - x\rangle \geq 0, \quad \forall y \in C. \tag{1.1}$$

Твердження 1. У випадку $C = H$ виконання варіаційної нерівності для точки x рівносильне виконанню рівності $A(x) = 0$.

Доведення. Справді, у випадку $C = H$ точка y пробігає увесь простір H. Тому для довільної фіксованої точки x точка $y-x$ також пробігає увесь простір H. Візьмемо y такий, що $y - x = -A(x)$, тоді

$$\langle A(x), y - x\rangle = \langle A(x), -A(x)\rangle = -\left\|A(x)\right\|^2 \leq 0, \tag{1.2}$$

причому рівність можлива лише якщо $A(x) = 0$. Отже, варіаційна нерівність може виконуватися тоді і тільки тоді, коли $A(x) = 0$. □

1.2 Зв'язок із задачами оптимізації

Прояснимо зв'язок варіаційної нерівності із задачами оптимізації.

Твердження 2. Для задачі

$$f \to \min_{C} \tag{1.3}$$

у випадку опуклості як f так і C критерієм того, що точка x є розв'язком є виконання нерівності

$$\langle f'(x), y - x\rangle \geq 0, \quad \forall y \in C. \tag{1.4}$$

Доведення. Запишемо лінійну апроксимацію f:

$$f(y) = f(x) + \langle f'(x), y - x\rangle + o\left(\|y - x\|\right). \tag{1.5}$$

Припустимо тепер, що другий доданок менше нуля для якогось $y = x + z$, тоді

$$f(x + z) - f(x) = \langle f'(x), z\rangle + o\left(\|z\|\right). \tag{1.6}$$

Розглянемо (з опуклості C випливає, що $x + \varepsilon z \in C$, а отже можемо підставляти таке y) тепер $y = x + \varepsilon z$, отримаємо

$$f(x + \varepsilon z) - f(x) = \langle f'(x), \varepsilon z\rangle + o\left(\|\varepsilon z\|\right). \tag{1.7}$$

З визначення $o(\cdot)$ зрозуміло, що при $\varepsilon \to +0$ знак правої частини визначає перший доданок.

Тобто, права частина буде від'ємною для якогось достатньо малого ε. Але тоді від'ємною буде і ліва частина, $f(x + \varepsilon z) - f(x) < 0$. Але це означає, що $f(x + \varepsilon z) < f(x)$. Отже, x не є точкою мінімуму f на C. Отримане протиріччя завершує доведення. □

Зауваження. У випадку відсутності опуклості або f або C або і того і того, цей критерій перетворюється на необхідну умову.

1.3 Зв'язок із сідловими точками

Розглянемо тепер оптимізацію з обмеженнями, тобто задачу

$$f(x) \xrightarrow[\substack{g_i(x)\le 0\\ i=1\ldots n}]{} \min. \tag{1.8}$$

Для цієї задачі можна побудувати функцію Лагранжа,

$$L(x,y) = f + \sum_{i=1}^{n} y_i g_i(x), \tag{1.9}$$

де y_i — множники Лагранжа.

Постає задача пошуку сідлової точки (справді, якщо у f мінімум в $\bar{x}$, то у L в $(\bar{x},\bar{y})$ буде мінімум по x і максимум по y, і навпаки) функції L.

Визначення (сідлової точки)**.** Точка $(\bar{x},\bar{y})$ називається *сідловою точкою* функції L якщо

$$L(\bar{x},y) \le L(\bar{x},\bar{y}) \le L(x,\bar{y}) \quad \forall x\ \forall y \tag{1.10}$$

тобто по x маємо мінімум в $\bar{x}$, а по y — максимум в $\bar{y}$.

Можемо записати ці умови наступним чином:

$$\begin{cases} \langle \nabla_1 L(\bar{x},\bar{y}), x-\bar{x}\rangle \ge 0 \quad \forall x \in C_1 \subseteq H_1, \\ \langle -\nabla_2 L(\bar{x},\bar{y}), y-\bar{y}\rangle \ge 0 \quad \forall y \in C_2 \subseteq H_2, \end{cases} \tag{1.11}$$

Зауваження. Ці нерівності можна об'єднати в одну:

$$\langle \nabla_1 L(\bar{x},y), x-\bar{x}\rangle + \langle -\nabla_2 L(\bar{x},\bar{y}), y-\bar{y}\rangle \ge 0. \tag{1.12}$$

1.4 Ерроу та Гурвіц

Приклад. Розглянемо тепер цілком конкретну функцію

$$L(x, y) = x \cdot y \tag{1.13}$$

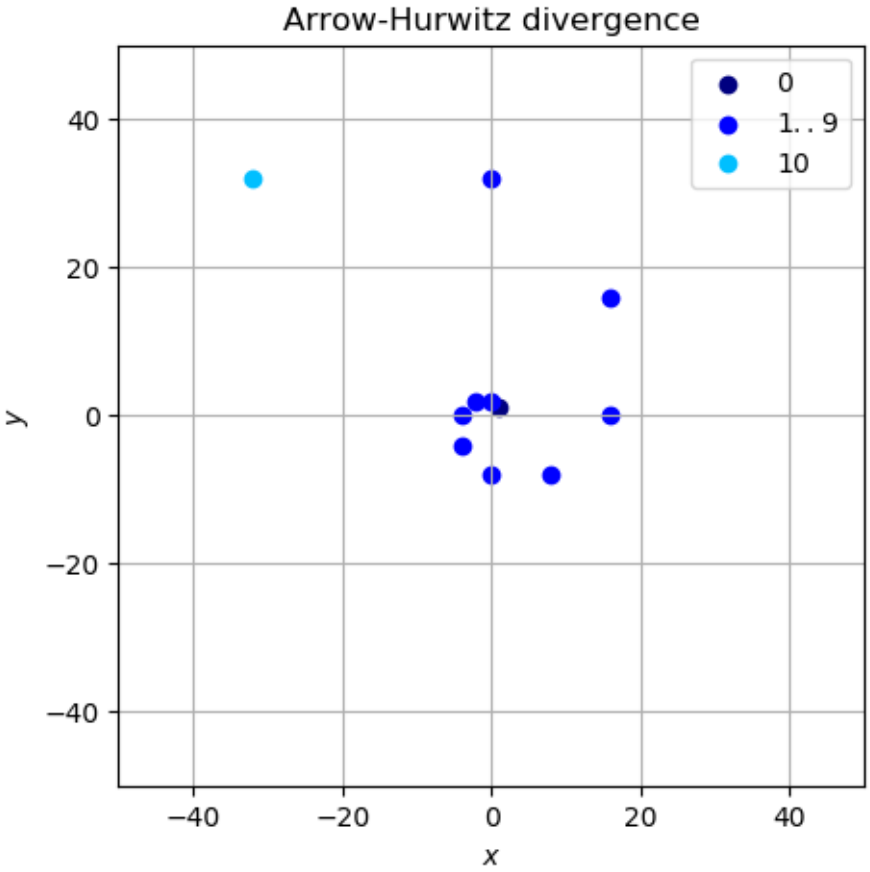
і спробуємо знайти її сідлову точку.

Розв'язок. Розглянемо алгоритм

$$\begin{aligned} x_{k+1} &:= x_k - \rho_k \nabla_1 L(x_k, y_k) = x_k - \rho_k y_k, \\ y_{k+1} &:= y_k + \rho_k \nabla_2 L(x_k, y_k) = y_k + \rho_k x_k, \end{aligned} \tag{1.14}$$

який називається *методом Ерроу-Гурвіца*.

Покладемо $(x_0, y_0) = (1, 1)$, $\rho_k \equiv 1$ і побачимо наступні ітерації:



Вони, очевидно, розбігаються, хоча здавалося що ми рухаємося у напрямку правильних градєнтів по кожній з компонент.

Зауваження. Для цієї задачі змінний розмір кроку нас не врятує, його зменшення тільки згладить спіраль по якій точки розбігаються.

Окрім емпіричних спостережень, ми можемо явно довести розбіжність, розглянемо для цього $|x_{k+1}|^2 + |y_{k+1}|^2$:

$$|x_{k+1}|^2 + |y_{k+1}|^2 = |x_k - \rho_k y_k|^2 + |y_k + \rho_k x_k|^2 = \\ = |x_k|^2 + \rho_k^2 \left(|x_k|^2 + |y_k|^2\right) + |y_k|^2 > |x_k|^2 + |y_k|^2, \quad (1.15)$$

у той час як сідловою точкою, очевидно, є $(0, 0)$. □

Зауваження. Не зважаючи на розбіжність методу Ерроу-Гурвіца на такій простій задачі, Ерроу свого часу був удостоєний Нобелівськиї премії з економіки, за задачі які цим методом можна розв'язати.

Виникає закономірне зпитання а що ж це за задачі такі.

1.5 Сильно опуклі функції

Для відповіді на це питання нам доведеться ввести

Визначення (сильно опуклої функції)**.** Функція $f = f(x)$ називається *μ-сильно опуклою* для деякого $\mu > 0$ якщо

$$f(\alpha \cdot x + (1-\alpha) \cdot y) \leq \alpha \cdot f(x) + (1-\alpha) \cdot f(y) - \mu \cdot \alpha \cdot (1-\alpha) \cdot \|x-y\|^2. \quad (1.16)$$

Про всяк введемо також трохи більш узагальнене

Визначення (сильно опуклої функції)**.** Функція $f = f(x)$ називається *g-сильно опуклою* для деякої g якщо

$$f(\alpha \cdot x + (1-\alpha) \cdot y) \leq \alpha \cdot f(x) + (1-\alpha) \cdot f(y) - \alpha \cdot (1-\alpha) \cdot g(\|x-y\|). \quad (1.17)$$

Можемо записати також альтернетивне визначення μ-сильно опуклості:

$$f(y) \geq f(x) + \langle \nabla f(x), y-x \rangle + \mu \cdot \|x-y\|^2. \quad (1.18)$$

Так от виявляється, що для задач із сильно опуклою функцією f метод Ерроу-Гурвіца збігається. Доведення наводиться нижче у більш загальному випадку, а поки що останнє

Зауваження. Існують певні ергодичні теореми, які стверджують збіжність середнього значення

$$(\tilde{x}_n, \tilde{y}_n) = \left(\frac{x_0 + \ldots + x_n}{n+1}, \frac{y_0 + \ldots + y_n}{n+1} \right) \quad (1.19)$$

до якоїсь сідлової точки $(\bar{x}, \bar{y})$, але подібні усереднені методи не є практичними, адже вони збігаються дуже повільно, а у задачі вище вже за тисячу ітерацій числа стають настільки великі що машинні похибки “переважають” усю теорію.

1.6 Монотонні оператори

Нагадаємо, що ми намгаємося знайти точку $x \in C$ яка задовольняє варіаційній нерівності

$$\langle Ax, y - x\rangle \geq 0 \quad \forall y \in C, \tag{1.20}$$

де оператор A, взагалі кажучи, не нерозтягуючий.

Подивимося на цю задачу як на задачу знаходження нерухомої точки оператора

$$T : x \mapsto P_C\left(x - \rho Ax\right), \tag{1.21}$$

де $\rho > 0$. Одразу зауважимо, що ці міркування приводять нас до наступного алгоритму

$$x_{k+1} \coloneqq P_C\left(x_k - \rho_k Ax_k\right), \tag{1.22}$$

збіжність якого ми зараз і проаналізуємо.

Взагалі хотілося б (відомо багато теорем щодо збіжності описаного алгоритму за таких умов) щоб оператор T був нерозтягуючим. Маємо:

$$\begin{aligned}\|Tx - Ty\|^2 \leq \|x - y - \rho(Ax - Ay)\|^2 \leq \\ \leq \|x - y\|^2 - 2\rho\langle Ax - Ay, x - y\rangle + \rho^2\|Ax - Ay\|^2.\end{aligned} \tag{1.23}$$

Для подальших оцінок нам знадобиться поняття монотонного оператора.

Визначення (монотонного оператора)**.** Оператор A: $H \to H$ називається *монотонним* якщо

$$\langle Ax - Ay, x - y\rangle \geq 0 \quad \forall x\ \forall y. \tag{1.24}$$

Поняття монотонності для операторів відіграє схожу роль з поняттям монотонності функцій.

Приклад. Оператор A називається *опуклим* якщо його градієнт ∇A монотонний.

Аналогічно до μ-сильно опуклих функцій існують μ-сильно опуклі оператори, для визначення яких вводиться

Визначення (сильно монотонного оператора)**.** Оператор A: $H \to H$ називається *μ-сильно монотонним* зі сталою $\mu > 0$ якщо

$$\langle Ax - Ay, x - y\rangle \geq \mu \cdot \|x - y\|^2 \quad \forall x\ \forall y. \tag{1.25}$$

Якщо оператор A — μ-сильно монотонний то можемо продовжити ланцюжок оцінок:

$$\|x-y\|^2 - 2\rho\langle Ax-Ay, x-y\rangle + \rho^2\|Ax-Ay\|^2 \leq \\ \leq \|x-y\|^2 - 2\rho\mu\|x-y\|^2 + \rho^2\|Ax-Ay\|^2. \quad (1.26)$$

Якщо ж при цьому оператор A ще й L-ліпшицеви (ліпшицевий з константою L, тобто $\|Ax-Ay\| \leq L\cdot\|x-y\|$), то можемо продовжити ще:

$$\|x-y\|^2 - 2\rho\mu\|x-y\|^2 + \rho^2\|Ax-Ay\|^2 \leq \\ \leq \|x-y\|^2 - 2\rho\mu\|x-y\|^2 + \rho^2L^2\|x-y\|^2 = \\ = (1-2\rho\mu+\rho^2L^2)\|x-y\|^2 = \kappa(\rho)\cdot\|x-y\|^2. \quad (1.27)$$

тобто достатньо обрати ρ так, щоб $\kappa(\rho)\in(0,1)$.

Розв'язуючи отриману квадратну нерівність знаходимо:

$$\rho \in \left(0, \frac{2\mu}{L^2}\right), \quad (1.28)$$

тобто знайшли цілий інтервал значень ρ для яких наш алгоритм буде збіжним.

Здавалося б все добре, але подивимося, у якій точці досягається мінімум $\kappa(\rho)$:

$$\tilde{\rho} = \frac{\mu}{L^2}, \quad (1.29)$$

і чому він дорівнює:

$$\kappa(\tilde{\rho}) = 1-2\rho\mu+\rho^2L^2 = 1-2\frac{\mu}{L^2}\mu + \left(\frac{\mu}{L^2}\right)^2L^2 = 1-\frac{\mu^2}{L^2}. \quad (1.30)$$

Зауваження. На жаль, правда життя така, що μ зазвича доволі мале, а L навпаки — доволі велике, тому $\rho<1$ але зовсім трохи. А це у свою чергу означає повільну збіжність.

1.7 Регуляризація

У той же час майже довільну опуклу функцію f можна замінити (цей процес називається регуляризацією) на ε-сильно опуклу функцію $f_\varepsilon = f + \varepsilon \|x\|^2$, тому може здатися, що всі наші проблеми розв'язані.

Так, у загальному випадку для монотонного оператора A можна розглянути оператор $A_\varepsilon = A + \varepsilon \mathbf{1}$, де $\mathbf{1}$, $x \mapsto x$ — *одиничний* (*тотожний*) *оператор*. Тоді можемо записати

$$\langle A_\varepsilon x - A_\varepsilon y, x - y\rangle = \underbrace{\langle Ax - Ay, x - y\rangle}_{\geq 0} + \varepsilon \cdot \|x - y\|^2 \geq \varepsilon \cdot \|x - y\|^2, \quad (1.31)$$

тобто оператор A_ε є ε-сильно опуклим.

Це наштовхує на думки про побудову алгоритму з ітераціями вигляду

$$x_{k+1} \coloneqq P_C\left(x_k - \rho_k A_{\varepsilon_k} x_k\right), \quad (1.32)$$

але тоді постає ще ряд запитань, наприклад які умови мають задовольняти $\{\rho_k\}_{k=1}^\infty$ і $\{\varepsilon_k\}_{k=1}^\infty$ для збіжності цього алгоритму. Поки що ці запитання лишаємо без відповіді.

1.8 Зв'язок із мережевими іграми і рівновагою Неша

Цей розділ взято з доповіді [5].

У багатьох соціальних та економіних задачах, рішення окремих індивідів (агентів) залежать лише від дій їхніх друзів, колег, однолітків чи суперників. Як приклади можна навести:

- Поширення інновацій, стилю життя.
- Формування громадських думок і соціальне навчання.
- Суперництво між конкурентними фірмами.
- Забезпечення суспільних благ.

Такі взаємодії можна промоделювати мережевою грою, що означає виконання наступних припущень:

- Агенти взаємодіють по ребрам мережі, представленої графом.
- Виграш кожного гравця залежить від його власних дій і від *агрегованого* значення дій агентів у його околі.

Формальніше, модель мережевої гри наступна: n агентів взаємодіють по мережі $G \in \mathbb{R}^{n\times n}$:

$$\begin{cases} G_{i,j} > 0 & \text{вплив } j \text{ на } i, \\ G_{i,i} = 0 & \text{без петель.} \end{cases} \tag{1.33}$$

У кожного агента i є:

- стретегія $x^i \in \mathcal{X}^i$, де $\mathcal{X}^i \subset \mathbb{R}^n$ — допустима множина стратегій ;
- цільова функція $J^i(x^i, z^i(x)) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, де $z^i(x) \coloneqq \sum_{j=1}^n G_{i,j}x^j$ — агрегатор.

Кожен агент намагється навчитися обчислювати свою оптимальну відповідь:

$$x^i_{\text{br}}(z^i) \coloneqq \operatorname*{argmin}_{x^i\in\mathcal{X}^i} J^i(x^i, z^i). \tag{1.34}$$

Нагадаємо

Визначення (рівноваги за Нешем)**.** Множина стратегій $\{\bar{x}_i\}_{i=1}^n$ називається *рівновагою за Нешем* якщо для кожного гравця i, $\bar{x}^i \in \mathcal{X}^i$:

$$J^i\left(\bar{x}^i, \sum_{j=1}^n G_{i,j}\bar{x}^j\right) \le J^i\left(x^i, \sum_{j=1}^m G_{i,j}\bar{x}^i\right), \quad \forall x^i \in \mathcal{X}^i. \tag{1.35}$$

Можна показати, що при виконанні наступних припущень:

- $\mathcal{X}^i \subset \mathbb{R}^n$ — замкнені, опуклі та обмежені;
- $J^i(x^i, z^i(x))$ опукла по x^i, для кожного вектора доповнюючих стратегій $x^{-i} \in \mathcal{X}^{-i}$;
- $J^i(x^i, z^i) \in C^2$ по x^i і z^i.

справджується наступне

Твердження 3. $\bar{x}$ є рівновагою за Нешем $\iff$ $\bar{x}$ є розв'язком варіаційної нерівності із допустимою множиною $\mathcal{X}$ і функцією F визначеними наступним чином:

$$\mathcal{X} \coloneqq \mathcal{X}^1 \times \cdots \times \mathcal{X}^n; \tag{1.36}$$

$$F(x) \coloneqq [F^i(x)]_{i=1}^n \coloneqq \begin{bmatrix} \nabla_{x^1} J^1(x^1, z^1(x)) \\ \vdots \\ \nabla_{x^n} J^n(x^n, z^n(x)) \end{bmatrix} \tag{1.37}$$

Твердження 4 (Facchinei та Pang, 2003)**.** Якщо окрім цього, якобіан гри F строго монотонний, то рівновага за Нешем існує та єдина.

1.9 Зв’язок із транспортними мережами

Цей розділ взято з книги [10].

Розглянемо один із підходів до моделювання і дослідження транспортних потоків, що базується на теорії конкурентної безкоаліційної рівноваги, яка дозволяє описати доволі адекватних механізм функціонування автомобільних вуличнодорожніх мереж (ВДМ).

Моделі, яки ми розглянемо, застосовуються для отримання прогнозних оцінок завантаженності елементів транспортної мережі. Подібні задачі цікаві, зокрема, тим, що є одним із інструкментів об’єктиної оцінки ефективності проектів з модифікації ВДМ з точки зору розвантаження найбільш завантажених і проблемних ділянок доріг.

Моделювання транспортних потоків як задача прийняття рішень

Для визначення обсягів завантаження ВДМ у першу чергу необхідно з’ясувати, чим керуються водії, обираючи той чи інший маршрут. Поведінкові принципи користувачів транспортної мережі були остаточно сформульовані Вардропом у [J. Wardrop, Some Theoretical Aspects of Road Traffic Research, 1952], де він наводить дві можливі ситуації:

1. користувачі мережі незалежно один від одного обирають маршрути з мінімальними затратами; (user optimization)
2. користувачі мережі обирають маршрути з мінімальними сумарними затратами. (system optimization)

Визначення. Ці принципи називаються, відповідно, *першим і другим принципами Вардропа*.

Перший принцип Вардропа відповідає конкурентній безколіаційній рівновазі (жоден окремий користувач не може знизити свої витрати відхилившись від свого поточного маршруту). Цей принцип передбачає ідеальну інформованість окремих користувачів про витрати на усіх шляхах (сьогодні це досягається за допомогою розумних навігаторів), а також відсутність суттєвого впливу на витрати на шляхах зі сторони одного конкретного користувача (не виконується для вантажних автомобілів а також аварійних ситуацій, але здебільшого виконується). Ре-

зюмуючи, перший принцип Вардропа сьогодні доволі точно описує поведінкові принципи користувачів ВДМ, а не є якоюсь надміру ідеалізованою абстракцією.

Формалізація задачі

Виходячи з наведених міркувань, побудуємо економіко-математичну моделі розподілу транспортних потоків у ВДМ. Транспортну мережу опишему у вигляді орієнтованого графу $G(V, E)$, де V — множина вершин (перехресть), E — множина орієнтованих дуг мережі (доріг між сусідніми перехрестями).

При дослідженні потокоформуючих факторів у множині вершин розглянемо дві підмножини. Перше, $S \subset V$, містить пункти, що породжують потоки; елементи множини S (sources) назвемо джерелами/витоками. Друге, $D \subset V$, містить пункти, що поглинають потоки; елементи множини D (destinations) назвемо стоками. Для задачі моделюванні трудових міграцій (для ранкових годин пік), джерелами будуть спальні райони міста, а стоками — ділові райони міста (для вечірніх годин ситуація рівно протилежна).

Через $P_{s,d}$ будемо позначати множину шляхів з витоку s у стік d. Окремі шляхи із витоку s у стік d будемо позначати $p_{s,d}$, або просто p якщо витік і стік несуттєві. Через f_p позначимо потік на шляху p, через $c_p(F)$ — маргинальні затрати на шляху p якщо загальна матриця потоків на шляхах дорівнює F (природніми прикладами колм затрати на одному шляху залежать від завантаженості інших є перехрестя головних і другорядних доріг, або ж паралельні дороги). Зрозуміло, що умова рівноваги має наступний вигляд:

$$\text{якщо } f_{p_{s,d}} > 0, \text{ то } c_p(F) = \min_{p' \in P_{s,d}} c_{p'}(F). \tag{1.38}$$

Окрім цього, для кожної пари (s, d) існує певний сталий (у випадку трудових міграцій) попит на транспорт, який будемо позначати $\rho_{s,d}$. Зрозуміло, що оптимальний рівноважний потік також має задовольняти наступним консервативним умовам:

$$\sum_{p \in P(s,d)} f_p = \rho_{s,d}. \tag{1.39}$$

Твердження 5. За таких умов допустима множина транспортних потоків $\mathcal{F}$ володіє гарними властивостями, а саме вона замкнута, опукла, і непорожня.

Зведення до варіаційної нерівності

Якщо залежність транспортних витрати від обсягів завантаження ВДМ є монотонною і неперевною, то пошук рівноважних потоків може бути зведений до розв'язання варіаційної нерівності.

Теорема 1. Потік $F^\star \in \mathcal{F}$ задовольняє умову рівноваги (1.38) тоді і лише тоді, коли він є розв'язком варіаційної нерівності

$$\langle C(F^\star), F - F^\star \rangle \geq 0, \quad \forall F \in \mathcal{F}, \tag{1.40}$$

де у матрицю C зібрані маргинальні транспортні витрати на усіх шляхах.

1.10 Подальші припущення

Надалі будемо розв'язувати наступну задачу:

$$\text{знайти } x \in C: \quad \langle Ax, y - x\rangle \geq 0, \quad \forall y \in C. \tag{1.41}$$

Також будемо вважати, що виконані наступні умови:

- множина $C \subseteq H$ — опукла і замкнена;
- оператор $A : H \to H$ — монотонний і ліпшицевий (із константою L);
- множина розв'язків (1.41) непорожня.

2 Алгоритми

2.1 Класичні алгоритми

Серед численних алгоритмів розв'язування (1.41) розглянемо три базових:

Алгоритм 1 (Корпелевич)**. Ініціалізація.** Вибираємо елементи x_1, $\lambda \in \left(0, \frac{1}{L}\right)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{2.1}$$

Якщо $x_n = y_n$ то зупиняємося і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A y_n), \tag{2.2}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Алгоритм 2 (P. Tseng)**. Ініціалізація.** Вибираємо елементи x_1, $\lambda \in \left(0, \frac{1}{L}\right)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{2.3}$$

Якщо $x_n = y_n$ то зупиняємося і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$x_{n+1} = y_n - \lambda(A y_n - A x_n), \tag{2.4}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Алгоритм 3 (Попов)**. Ініціалізація.** Вибираємо елементи x_1, y_0, $\lambda \in \left(0, \frac{1}{3L}\right)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A y_{n-1}). \tag{2.5}$$

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A y_n). \tag{2.6}$$

Якщо $x_{n+1} = x_n = y_n$ то зупиняємо алгоритм і x_n — розв'язок, інакше покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Зауваження. У наведеному вище вигляді алгоритми Tseng'а і Попова обчислюють оператор A тричі і двічі на кожну ітерацію відповідно. На цьому можна заощадити якщо кешувати обчислення оператора A. У випадку алгоритма Tseng'а спосіб кешування очевидний: один раз обчислюємо Ax_n і двічі використовуємо його (для y_n та x_{n+1}). У випадку алгоритма Попова кешування допомагає за рахунок того, що значення Ay_n використовується один раз на ітерації n для обчислення x_{n+1}, і ще раз на ітерації $n + 1$ для обчислення значення y_{n+1}.

В теорії, у випадку коли P_C обчислювати дешево (наприклад, коли це можливо аналітично), а A обчислювати дорого, такий трюк допомагає пришвидшити алгоритм Tseng'а у 1.5, а алгоритм Попова — у 2 рази.

2.2 Адаптивні алгоритми

Не так давно з'явилися адаптивні алгоритми, тобто такі, що не вимагають знання константи Ліпшиця. Наведемо адаптивні версії розгялнутих раніше алгоритмів:

Алгоритм 4 (Адаптивний Корпелевич)**. Ініціалізація.** Вибираємо елементи x_1, $\tau \in (0,1)$, $\lambda \in (0,+\infty)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{2.7}$$

Якщо $x_n = y_n$ то зупиняємося і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A y_n). \tag{2.8}$$

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } \langle Ax_n - Ay_n, x_{n+1} - y_n\rangle \le 0, \\ \min\left\{\lambda_n, \dfrac{\tau}{2}\dfrac{\|x_n - y_n\|^2 + \|x_{n+1} - y_n\|^2}{\langle Ax_n - Ay_n, x_{n+1} - y_n\rangle}\right\}, \text{ інакше.} \end{cases} \tag{2.9}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Зауваження. У алгоритмі 4 можна робити і так:

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ax_n - Ay_n = 0, \\ \min\left\{\lambda_n, \tau\dfrac{\|x_n - y_n\|}{\|Ax_n - Ay_n\|}\right\}, \text{ інакше.} \end{cases} \tag{2.10}$$

Алгоритм 5 (Адаптивний Tseng)**. Ініціалізація.** Вибираємо елементи x_1, $\tau \in (0,1)$, $\lambda \in (0,+\infty)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{2.11}$$

Якщо $x_n = y_n$ то зупиняємося і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$x_{n+1} = y_n - \lambda(Ay_n - Ax_n), \tag{2.12}$$

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ax_n - Ay_n = 0, \\ \min\left\{\lambda_n, \tau\dfrac{\|x_n - y_n\|}{\|Ax_n - Ay_n\|}\right\}, \text{ інакше.} \end{cases} \tag{2.13}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Алгоритм 6 (Адаптивний Попов)**. Ініціалізація.** Вибираємо елементи $x_1, y_0, \tau \in (0, \frac{1}{3})$, $\lambda \in (0, +\infty)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda Ay_{n-1}). \tag{2.14}$$

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda Ay_n). \tag{2.15}$$

Якщо $x_{n+1} = x_n = y_n$ то зупиняємо алгоритм і x_n — розв'язок, інакше переходимо на

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } \langle Ay_{n-1} - Ay_n, x_{n+1} - y_n\rangle \le 0, \\ \min\left\{\lambda_n, \dfrac{\tau}{2}\dfrac{\|y_{n-1} - y_n\|^2 + \|x_{n+1} - y_n\|^2}{\langle Ay_{n-1} - Ay_n, x_{n+1} - y_n\rangle}\right\}, \text{ інакше.} \end{cases} \tag{2.16}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

Зауваження. У алгоритмі 6 можна робити і так:

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ay_{n-1} - Ay_n = 0, \\ \min\left\{\lambda_n, \tau\dfrac{\|y_{n-1} - y_n\|}{\|Ay_{n-1} - Ay_n\|}\right\}, \text{ інакше.} \end{cases} \tag{2.17}$$

2.3 Алгоритм Маліцького—Tam'a

Зовсім нещодавно (у 2015-ому році) Юра Маліцький запропонував наступну схему:

Алгоритм 7 (Маліцький—Tam)**. Ініціалізація.** Вибираємо елементи $x_1, x_0 \in H$, $\lambda \in (0, \frac{1}{2L})$. Покладаємо $n = 1$.

Крок. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A x_n - \lambda(Ax_n - Ax_{n-1})). \tag{2.18}$$

Якщо $x_{n+1} = x_n = x_{n-1}$ то зупиняємо алгоритм і x_n — розв'язок, інакше покладаємо $n := n + 1$, і повторюємо

Алгоритм 8 (Адаптивний Маліцький—Tam)**. Ініціалізація.** Вибираємо елементи $x_1, x_0 \in H$, $\lambda_1, \lambda_0 > 0$, $\tau \in (0, \frac{1}{2})$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A x_n - \lambda(Ax_n - Ax_{n-1})). \tag{2.19}$$

Якщо $x_{n+1} = x_n = x_{n-1}$ то зупиняємо алгоритм і x_n — розв'язок, інакше переходимо на

Крок 2. Обчислюємо

$$\lambda_{n+1} = \min\left\{\lambda_n, \tau\frac{\|x_{n+1} - x_n\|}{\|Ax_{n+1} - Ax_n}\right\}, \tag{2.20}$$

покладаємо $n := n + 1$ і переходимо на **Крок 1.**

3 Задачі

Для порівняння алгоритмів нам знадобляться тестові задачі різної складності та різних розмірів. У якості таких задачі розглянемо:

3.1 Перша задача

Класичний приклад. Допустимою множиною є увесь простір: $C = \mathbb{R}^m$, а $F(x) = Ax$, де A — квадратна $m \times m$ матриця, елементи якої визначаються наступним чином:

$$a_{i,j} = \begin{cases} -1, & j = m - 1 - i > i, \\ 1, & j = m - 1 - i < i, \\ 0, & \text{інакше.} \end{cases} \tag{3.1}$$

Зауваження. Тут і надалі нуметрація рядків/стовпчиків матриць, а також елементів масивів починається з нуля. Якщо у вашій мові програмування нумерація починається з одиниці то у виразах вище замість $m - 1$ має бути $m + 1$.

Це визначає матрицю, чия бічна діагональ складається з половини одиниць і половини мінус одиниць, а решта елементів якої нульові. Для наглядності наведемо кілька преших матриць, для $m = 2, 4$:

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{3.2}$$

Для парних m нульовий вектор є розв'язком відповідної варіаційної нерівності (1.41). Для усіх чисельних есперементів ми брали $x_0 = (1, \ldots, 1)$, $\varepsilon = 10^{-3}$, $\lambda = 0.4$.

Зауваження. Ця задача є узагальненням 1.13. Як ми вже знаємо, звичайні градієнтні методи для неї не збігаються.

Зауваження. Для цієї задачі $P_C = \mathrm{Id}$, а тому алгоритми Корпелевич і Tseng'а еквівалентні. Втім, некешована версія алгоритму Tseng'а буде працювати повільніше, у чому ми переконаємося у результатах.

3.2 Друга задача

Візьмемо $F(x) = Mx + q$, де випадкова матриця M генерується наступним чином:

$$M = AA^{\intercal} + B + D, \tag{3.3}$$

де всі елементи $m \times m$ матриці A і $m \times m$ кососиметричної матриці B обираються рівномірно випадково з $(-5, 5)$, а усі елементи діагональної матриці D вибираються рівномірно випадково з $(0, 0.3)$ (як наслідок, матриця M додатно визначена), а кожен елемент q обирається рівномірно випадково з $(-500, 0)$. Допустимою множиною є

$$C = \{x \in \mathbb{R}^m_+ | x_1 + x_2 + \cdots + x_m = m\}. \tag{3.4}$$

Допустима множина цієї задачі — так званий *probability symplex* (з точністю до константи m). Для проектування $\vec{y}$ на нього ми використовували наступний явний

Алгоритм 9.

Крок 1. Відсортувати елементи $\vec{y}$ і зберегти в $\vec{u}$: $u_1 \geq \cdots \geq u_m$.

Крок 2. Знайти $k = \max j$: $u_j + \frac{1}{j}\left(m - \sum_{i=1}^{j} u_i\right) > 0$.

Крок 3. Видати вектор з елементами $x_i = \max\{y_i + \lambda, 0\}$, $\lambda = \frac{1}{k}\left(m - \sum_{i=1}^{k} u_i\right)$.

Цей алгоритм взято із статті [2].

Для усіх чисельних експериментів ми брали $x_0 = (1, \ldots, 1)$, $\varepsilon = 10^{-3}$, $L = \|M\|$. також, для кожного розмірку задачі було згенеровано кілька різних M.

3.3 Третя задача

Нелінійна доповлююча задача Коджими—Шиндо була розглянута у [6] і [3], де $m = 4$, а функція F визначається наступним чином:

$$F(x_1, x_2, x_3, x_4) = \begin{bmatrix} 3x_1^2 + 2x_1x_2 + 2x_2^2 + x_3 + 3x_4 - 6 \\ 2x_1^2 + x_1 + x_2^2 + 10x_3 + 2x_4 - 2 \\ 3x_1^2 + x_1x_2 + 2x_2^2 + 2x_3 + 9x_4 - 9 \\ x_1^2 + 3x_2^2 + 2x_3 + 3x_4 - 3 \end{bmatrix}. \tag{3.5}$$

Допустимою множиною знову є ймовірнісний симплекс

$$C = \left\{ x \in \mathbb{R}^4 \middle| x_1 + x_2 + x_3 + x_4 \right\}.$$

Для чисельних експериментів ми спершу взяли кілька конкретних стартових точок $x_1 = (1, 1, 1, 1)$ і $x_1 = (0.5, 0.5, 2, 1)$, а потім ще декілька випадкових точок із C. Для усіх стартових точок проводилося два тести: один із $\varepsilon = 10^{-3}$, второй с $\varepsilon = 10^{-6}$.

Зауваження. У цієї задачі неєдиний розв'язок, хоча здебільшого алгоритми збігалися до $x_\star = (1.225, 0, 0, 2.775)$. Втім, не завжди, що і пояснює значну різницю у кількості ітерацій між деякими алгоритмами на деяких тестах.

3.4 Четверта задача

Цей приклад був розглянутий Суном у [7]:

$$\begin{aligned} F(x) &= F_1(x) + F_2(x), \\ F_1(x) &= (f_1(x), f_2(x), \ldots, f_m(x)), \\ F_2(x) &= Dx + c, \\ f_i(x) &= x_{i-1}^2 + x_i^2 + x_{i-1}x_i + x_i x_{i+1}, \quad m = 1, 2, \ldots, m, \\ x_0 &= x_{m+1} = 0, \end{aligned} \tag{3.6}$$

де D — квадратна $m \times m$ матриця з наступними елементами:

$$d_{i,j} = \begin{cases} 1, & j = i - 1, \\ 4, & j = i, \\ -2, & j = i + 1, \\ 0, & \text{інакше}, \end{cases} \tag{3.7}$$

$c = (-1, -1, \ldots, -1)$. Допустимою множиною є $C = \mathbb{R}_+^m$, а початкова точка $x_1 = (0, 0, \ldots, 0)$.

Для кращого розуміння наведемо матрицю D для кількох перших $m = 3, 4, 5$:

$$\begin{pmatrix} 4 & -2 & 0 \\ 1 & 4 & -2 \\ 0 & 1 & 4 \end{pmatrix} \qquad \begin{pmatrix} 4 & -2 & 0 & 0 \\ 1 & 4 & -2 & 0 \\ 0 & 1 & 4 & -2 \\ 0 & 0 & 1 & 4 \end{pmatrix} \qquad \begin{pmatrix} 4 & -2 & 0 & 0 & 0 \\ 1 & 4 & -2 & 0 & 0 \\ 0 & 1 & 4 & -2 & 0 \\ 0 & 0 & 1 & 4 & -2 \\ 0 & 0 & 0 & 1 & 4 \end{pmatrix} \tag{3.8}$$

Для усіх чисельних експериментів ми брали $x_1 = (0, \ldots, 0)$. Знову ж таки, було розглянуто два значення ε: 10^{-3} і 10^{-6}.

Зауваження. Матриця тридіагональна, ми цим скористаємося для ефективного зберігання і множення.

4 Результати

Тестування відбувалося на машині із процесором Intel Core i7-8550U 1.99GHz під 64-бітною версією операційної системи Windows 10.

4.1 Перша задача

Неадаптивні алгоритми

m	Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
1000	0.10	132	0.11	132	0.08	132	0.05	89	0.03	89	0.08	91	0.03	91
2000	0.58	137	0.92	137	0.53	137	0.36	92	0.18	92	0.54	94	0.19	94
5000	4.08	144	6.39	144	4.34	144	2.85	96	1.46	96	4.37	98	1.53	98
10000	17.43	148	26.73	148	17.82	148	13.14	99	6.56	99	18.94	101	5.85	101

Таблиця 4.1: Результати неадаптивних алгоритмів для першої задачі

Бачимо що алгоритм Корпелевич і кешований алгоритм Tseng'а справді показують майже однакові результати, що є непрямим доказом їхньої еквівалентності. Однакова кількість ітерацій некешованих і кешованих версій усіх алгоритмів слугує непрямим доказом їхньої еквівалентності. Втім, кешовані версії передбачувано виграють (у 1.5–3 рази) у некешованих версій по часу роботи. З точки зору часу роботи найкращі результати показує кешована версія алгоритму Маліцького—Tam'а, з точки зору числа ітерацій — алгоритм Попова, хоча алгоритм Маліцького—Tam'а майже йому не поступається.

m	Корп.		кеш. Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
1000	0.26	132	0.07	132	0.24	132	0.08	132	0.13	89	0.04	89	0.17	91	0.03	91
2000	1.74	137	0.57	137	2.01	137	0.57	137	1.09	92	0.19	92	1.39	94	0.24	94
5000	12.66	144	4.29	144	15.40	144	4.24	144	8.16	96	1.38	96	10.25	98	1.65	98
10000	51.37	148	17.61	148	66.98	148	17.16	148	34.25	99	5.85	99	40.75	101	5.84	101

Таблиця 4.2: Результати адаптивних алгоритмів для першої задачі

Бачимо що алгоритм Корпелевич і алгоритм Tseng'а справді показують майже однакові результати, що є непрямим доказом їхньої еквівалентності. Однакова кількість ітерацій некешованих і кешованих версій усіх алгоритмів слугує непрямим доказом їхньої еквівалентності. Втім, кешовані версії передбачувано виграють (у 3–7 разів) у некешованих версій по часу роботи. З точки зору часу роботи найкращі і майже еквівалентні результати показують кешовані версії алгоритмів Маліцького—Tam'а і Попова, з точки зору числа ітерацій — алгоритм Попова, хоча алгоритм Маліцького—Tam'а майже йому не поступається. Задача доволі проста, а тому адаптивні алгоритми не випереджають звичайні.

Розріджені матриці, неадаптивні алгоритми

Нескладно помітити, що матриця A дуже розріджена, що наводить на ідею скористатися модулем scipy.sparse для ефективної роботи з розрідженими матрицями. Це дозволить нам розв'язувати задачу для значно більших m.

m	Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
50000	0.15	159	0.25	159	0.16	159	0.10	106	0.08	106	0.13	108	0.08	108
100000	0.51	164	0.74	164	0.39	164	0.37	109	0.33	109	0.43	111	0.30	111
200000	2.32	169	3.06	169	2.53	169	1.56	112	1.22	112	2.13	114	1.41	114
500000	6.21	175	8.26	175	6.87	175	4.14	117	3.33	117	5.74	119	3.78	119

Таблиця 4.3: Результати неадаптивних алгоритмів для першої задачі із розрідженими матрицями

Бачимо що алгоритм Корпелевич і алгоритм Tseng'а справді показують майже однакові результати, що є непрямим доказом їхньої еквівалентності. Однакова кількість ітерацій некешованих і кешованих версій усіх алгоритмів слугує непрямим доказом їхньої еквівалентності. Тут перевага кешування вже не така очевидна, адже ми значно здешевили обчислення оператора A, хоча все ще присутня (у 1.5 рази). З точки зору часу роботи найкращі і майже еквівалентні результати показують кешовані версії алгоритмів Маліцького—Tam'а і Попова, з точки зору числа ітерацій — алгоритм Попова, хоча алгоритм Маліцького—Tam'а майже йому не поступається. Зауважимо, що розміри задачі зросли у 50–100 разів від використання розріджених матриць.

m	Корп.		кеш. Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
50000	0.89	159	0.61	159	0.63	159	0.45	159	0.53	106	0.16	106	0.34	108	0.15	108
100000	1.53	164	0.95	164	1.13	164	0.54	164	0.89	109	0.35	109	0.75	111	0.31	111
200000	6.21	169	3.58	169	6.18	169	3.57	169	4.05	112	2.00	112	4.11	114	2.03	114
500000	16.27	175	9.29	175	16.35	175	9.29	175	10.96	117	5.21	117	11.03	119	5.22	119

Таблиця 4.4: Результати адаптивних алгоритмів для першої задачі із розрідженими матрицями

Бачимо що алгоритм Корпелевич і алгоритм Tseng'а справді показують майже однакові результати, що є непрямим доказом їхньої еквівалентності. Однакова кількість ітерацій некешованих і кешованих версій усіх алгоритмів слугує непрямим доказом їхньої еквівалентності. Тут перевага кешування вже не така очевидна, адже ми значно здешевили обчислення оператора A, хоча все ще присутня (у 2 рази). З точки зору часу роботи найкращі і майже еквівалентні результати показують кешовані версії алгоритмів Маліцького—Tam'а і Попова, з точки зору числа ітерацій — алгоритм Попова, хоча алгоритм Маліцького—Tam'а майже йому не поступається. Задача доволі проста, а тому адаптивні алгоритми не випереджають звичайні. Зауважимо, що розміри задачі зросли у 50–100 разів від викорситання розріджених матриць.

4.2 Друга задача

Неадаптивні алгоритми

m	Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
100	0.62	967	0.48	967	0.40	967	0.55	967	0.43	967	0.47	967	0.29	967
100	0.71	1151	0.58	1151	0.47	1151	0.68	1151	0.58	1151	0.59	1151	0.36	1151
100	0.63	1086	0.54	1086	0.42	1086	0.62	1086	0.50	1086	0.55	1086	0.34	1086
200	1.41	1849	1.07	1849	0.90	1849	1.39	1849	1.26	1849	1.08	1849	0.72	1849
200	1.30	1672	1.01	1672	0.85	1672	1.30	1672	1.10	1672	1.03	1672	0.69	1672
200	1.35	1715	1.13	1715	0.91	1715	1.32	1715	1.16	1715	1.03	1715	0.71	1715
500	3.67	2523	2.34	2523	2.07	2523	3.63	2523	3.30	2523	2.53	2523	1.88	2523
500	4.33	2543	2.74	2543	2.29	2543	3.77	2543	3.27	2543	2.39	2543	1.94	2543
500	4.07	2725	2.62	2725	2.36	2725	4.10	2725	3.48	2725	2.58	2725	2.16	2725
1000	10.09	3542	8.29	3542	7.23	3542	9.62	3542	8.05	3542	7.95	3543	5.07	3543
1000	9.18	3570	7.37	3570	6.16	3570	10.57	3570	8.05	3570	7.35	3570	4.75	3570
1000	9.40	3471	7.55	3471	6.67	3471	9.68	3471	7.76	3471	7.34	3471	5.11	3471

Таблиця 4.5: Результати неадаптивних алгоритмів для другої задачі

Однакова кількість ітерацій некешованих і кешованих версій усіх алгоритмів слугує непрямим доказом їхньої еквівалентності. Втім, кешовані версії все ще передбачувано виграють (у 1.5 рази) у некешованих версій по часу роботи. З точки зору часу роботи найкращі результати показує кешована версія алгоритму Маліцького—Tam'a. У цій задачі обчислення проекції вже більш складне, тому алгоритм Tseng'a має певну перевагу над алгоритмом Попова.

m	Корп.		кеш. Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
100	1.14	967	0.64	967	0.99	967	0.44	967	0.97	967	0.51	967	0.89	967	0.35	967
100	1.16	1151	0.73	1151	1.10	1151	0.52	1151	1.15	1151	0.58	1151	1.09	1151	0.42	1151
100	1.10	1086	0.69	1086	1.02	1086	0.49	1086	1.08	1086	0.56	1086	1.01	1086	0.41	1086
200	2.30	1849	1.59	1849	2.01	1849	1.10	1849	2.37	1849	1.56	1849	2.33	1849	0.90	1849
200	2.06	1672	1.38	1672	1.74	1672	1.01	1672	2.06	1672	1.21	1672	1.88	1672	0.81	1672
200	2.07	1715	1.46	1715	1.82	1715	0.94	1715	2.05	1715	1.27	1715	1.83	1715	0.86	1715
500	5.30	2523	3.88	2523	3.85	2523	2.32	2523	4.68	2523	3.25	2523	3.71	2523	2.03	2523
500	5.08	2543	4.13	2543	3.88	2543	2.50	2543	5.14	2543	3.29	2543	4.16	2543	2.14	2543
500	5.36	2725	3.99	2725	3.99	2725	2.43	2725	5.26	2725	3.50	2725	4.15	2725	2.33	2725
1000	15.93	3542	10.31	3542	11.56	3542	6.54	3542	14.59	3542	8.05	3542	12.11	3543	5.45	3543
1000	14.28	3570	9.68	3570	11.33	3570	6.51	3570	15.09	3570	7.89	3570	11.13	3570	4.89	3570
1000	14.43	3471	9.42	3471	11.47	3471	6.40	3471	14.79	3471	8.50	3471	11.99	3471	5.37	3471

Таблиця 4.6: Результати адаптивних алгоритмів для другої задачі

Однакова кількість ітерацій некешованих і кешованих версій усіх алгоритмів слугує непрямим доказом їхньої еквівалентності. Втім, кешовані версії передбачувано виграють (у 3–7 разів) у некешованих версій по часу роботи. З точки зору часу роботи найкращі результати показує кешована версія алгоритму Маліцького—Tam'a. У цій задачі обчислення проекції вже більш складне, тому алгоритм Tseng'a має певну перевагу над алгоритмом Попова. Задача доволі проста, а тому адаптивні алгоритми не випереджають звичайні. Можна додати якийсь із матричних розкладів M для пришвидшення множення Mx.

4.3 Третя задача

Адаптивні алгоритми

x_1	ε	Корп.		кеш. Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
		час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
$(1,1,1,1)$	10^{-3}	0.02	30	0.01	30	0.05	156	0.03	156	0.01	29	0.01	29	0.02	49	0.01	49
	10^{-6}	0.02	58	0.01	58	0.15	475	0.08	475	0.02	56	0.01	56	0.04	142	0.02	142
$(\frac{1}{2},\frac{1}{2},2,1)$	10^{-3}	0.02	51	0.01	51	0.33	980	0.20	980	0.02	51	0.01	51	0.02	69	0.01	69
	10^{-6}	0.04	104	0.03	104	2.00	6503	1.13	6503	0.03	104	0.02	104	0.05	156	0.02	156
random	10^{-3}	0.01	37	0.01	37	0.14	444	0.09	444	0.01	36	0.01	36	0.02	68	0.01	68
	10^{-6}	0.03	72	0.02	72	0.52	1602	0.31	1602	0.03	71	0.02	71	0.05	154	0.03	154
random	10^{-3}	0.01	39	0.01	39	0.12	359	0.07	359	0.02	55	0.01	55	0.05	148	0.02	148
	10^{-6}	0.04	121	0.03	121	0.40	1231	0.24	1231	0.05	141	0.03	141	0.20	632	0.10	632
random	10^{-3}	0.02	46	0.01	46	0.37	1160	0.23	1160	0.02	46	0.01	46	0.02	69	0.01	69
	10^{-6}	0.03	94	0.02	94	—	—	—	—	0.03	94	0.02	94	0.05	165	0.02	165

Таблиця 4.7: Результати адаптивних алгоритмів для третьої задачі

З певних причин на цій задачі суттєво просідає алгоритм Tseng'а. Що стосується решти, то тут алгоритми Корпелевич і Попова випереджають алгоритм Маліцького—Tam'а за числом ітерацій. Щоправда, якщо почати рахувати за числом обчислень оператора і проектора, то алгоритм Маліцького—Tam'а не буде їм поступатися. Розв'язок цієї задачі неєдиний: для деяких випадкових стартових точок алгоритми збігаються до різних розв'язків, що пояснює велику різницю то числу ітерацій між ними.

4.4 Четверта задача

Адаптивні алгоритми

m	ε	Корп.		кеш. Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
		час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
500	10^{-3}	0.05	35	0.02	35	0.25	175	0.09	175	0.04	34	0.01	34	0.04	31	0.01	31
	10^{-6}	0.09	65	0.03	65	0.52	343	0.17	343	0.08	63	0.02	63	0.13	60	0.03	60
1000	10^{-3}	0.09	37	0.03	37	0.53	183	0.17	183	0.07	35	0.01	35	0.07	32	0.01	32
	10^{-6}	0.23	67	0.06	67	1.16	352	0.43	352	0.13	65	0.03	65	0.15	62	0.03	62
2000	10^{-3}	0.48	38	0.20	38	3.29	192	0.82	192	0.46	37	0.11	37	0.56	34	0.08	34
	10^{-6}	0.85	69	0.36	69	5.23	360	1.49	360	0.80	66	0.14	66	0.89	63	0.13	63
5000	10^{-3}	4.10	40	1.20	40	21.45	203	6.37	203	3.36	39	0.69	39	3.89	36	0.53	36
	10^{-6}	5.95	71	1.98	71	36.11	371	10.37	371	5.64	68	1.00	68	6.34	65	0.93	65

Таблиця 4.8: Результати адаптивних алгоритмів для четвертої задачі

З певних причин на цій задачі просідає алгоритм Tseng'а. Кешовані версії передбачувано виграють (у 3–7 разів) у некешованих версій по часу роботи. Як з точки зору числа ітерацій, так і з точки зору часу роботи найкращим є алгоритм Маліцького—Tam'а, хоча алгоритм Попова і не дуже сильно йому поступається.

Розріджені матриці, адаптивні алгоритми

Нескладно помітити, що матриця A дуже розріджена, що наводить на ідею скористатися модулем scipy.sparse для ефективної роботи з розрідженими матрицями. Це дозволить нам розв'язувати задачу для значно більших m.

m	ε	Корп.		кеш. Корп.		Tseng		кеш. Tseng		Попов		кеш. Попов		Маліц.		кеш. Маліц.	
		час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.	час	ітер.
20000	10^{-3}	0.20	43	0.09	43	1.12	220	0.44	220	0.20	42	0.06	42	0.20	39	0.05	39
	10^{-6}	0.35	74	0.15	74	1.92	388	0.76	388	0.34	71	0.11	71	0.41	68	0.12	68
50000	10^{-3}	0.83	45	0.39	45	4.14	231	1.68	231	0.79	44	0.21	44	0.80	41	0.20	41
	10^{-6}	1.21	76	0.54	76	7.87	399	2.49	399	1.18	73	0.28	73	1.31	70	0.29	70
100000	10^{-3}	2.00	47	0.68	47	11.05	240	3.52	240	1.80	45	0.44	45	2.05	42	0.37	42
	10^{-6}	2.70	77	1.16	77	18.07	408	6.91	408	2.86	74	0.71	74	3.38	71	0.55	71
200000	10^{-3}	7.36	48	3.00	48	42.02	248	15.85	248	7.15	47	1.99	47	7.39	44	1.76	44
	10^{-6}	12.98	79	4.94	79	72.83	416	25.79	416	11.79	76	3.24	76	12.50	73	2.92	73

Таблиця 4.9: Результати адаптивних алгоритмів для четвертої задачі із розрідженими матрицями

З певних причин на цій задачі просідає алгоритм Tseng'а. Кешовані версії передбачувано виграють (у 3–7 разів) у некешованих версій по часу роботи. Як з точки зору числа ітерацій, так і з точки зору часу роботи найкращим є алгоритм Маліцького—Tam'а, хоча алгоритм Попова і не дуже сильно йому поступається. Зауважимо, що розміри задачі зросли у 50–100 разів від використання розріджених матриць.

5 ВИСНОВОК

В рамках дипломної роботи були виконані наступні завдання:

- розглянуто широкий спектр алгоритмів розв’язання варіаційної нерівності: алгоритми Корпелевич, Tseng’а, Попова, Маліцького—Tam’а;
- розроблено програмний засіб розв’язування варіаційної нерівності за допомогою алгоритмів Корпелевич, Tseng’а, Попова, Маліцького—Tam’а;
- виконано тестування алгоримтмів на широкому спектрі модельних задач;
- проведено детальний аналіз швидкодії і ефективності алгоритмів.

Зроблено акцент на ефективності програмної реалізації.

Зокрема, використовується такий прийом як кешування уже обчислених значень оператора і проектора, що дозволяє пришвидшити алгоритм від 2 до 8 разів, залнжно від особливостей задачі й алгоритму.

Окрім цього, увага приділяється використанню розріджених матриць, що дозволяє на порядки підняти ефективність алгоритмів для задач із розрідженими матрицями (максимальний розмір задачі, яку можна розв’язати за прийнятний час зростає у *сотні* разів).

Нарешті, згадується можливість кешування матричних розкладів для пришвидшення матрично-векторних операцій для відповідних задач. За нашими оцінками, для складних задач які потребують тисяч ітерацій це може принести виграш у часі роботи приблизно ще на один порядок.

Автор переконаний, що у сучасному світі для конкретної практичної задачі грамотна програмна реалізація є настільки ж важливою, наскійки й оптимальний алгоритм.

Бібліографія

[1] Bauschke, H. H. Convex Analysis and Monotone Operator Theory in Hilbert Spaces / H. H. Bauschke, P. L. Combettes. — Springer, 2011.

[2] Efficient projections onto the ℓ_1-ball for learning in high dimensions / John Duchi, Shai Shalev-Shwartz, Yoram Singer, Tushar Chandra // ICML '08: Proceedings of the 25th international conference on Machine learning. — 2008. — P. 272–279. — https://stanford.edu/~jduchi/projects/DuchiShSiCh08.pdf.

[3] Kanzow, C. some equation-based methods for the nonlinear complementarity problem / Christian Kanzow // Optimization Methods and Software. — 1994. — Vol. 3, no. 4. — P. 327–340.

[4] Malitsky, Y. projected reflected gradient methods for monotone variational inequalities / Yura Malitsky // SIAM Journal on Optimization. — 2015. — Vol. 25. — P. 502–520.

[5] Özdaglar, A. A Variational Inequality Framework for Network Games: Existence, Uniqueness, Convergence and Sensitivity Analysis / Asu Özdaglar. — 2018. — https://simons.berkeley.edu/talks/asu-ozdaglar-3-28-18.

[6] Pang, J.-S. NE/SQP: A robust algorithm for the nonlinear complementarity problem / J.-S. Pang, S. A. Gabriel // Mathematical Programming. — 1993. — Vol. 60. — P. 295–337.

[7] Sun, D. A Projection and Contraction Method for the Nonlinear Complementarity Problem and Its Extensions / Defeng Sun // Mathematica Numerica Sinica. — 2012. — Vol. 16. — P. 183–194.

[8] Tseng, P. on linear convergence of iterative methods for the variational inequality problem / P. Tseng // Journal of Computational and Applied Mathematics. — 1995. — Vol. 60. — P. 237–252.

[9] Антипин, А.С. О методе выпуклого программирования, использующем симметрическую модификацию функции Лагранжа / Антипин, А.С. // Экономика и математические методы. — 1976. — Vol. 12. — P. 1164–1173.

[10] Гасников, А. В. Введение в математическое моделирование транспортных потоков / Гасников, А. В. — МФТИ, 2010.

[11] Корпелевич, Г.М. Экстраградиентный метод для поиска седловой точки и других задач / Корпелевич, Г.М. // Экономика и математические методы. — 1976. — Vol. 12. — P. 747–756.

[12] Ляшко, С.І., Семенов, В.В. Нова економічна модифікація методу Корпелевич для монотонних задач про рівновагу / Ляшко, С.І., Семенов, В.В. // Кібернетика і системний аналіз. — 2011. — Vol. 47. — P. 631–639.

[13] Маліцький, Ю.В., Семенов, В.В. Варіант екстраградієнтного алгоритму для монотонних варіаційних нерівностей / Маліцький, Ю.В., Семенов, В.В. // Кібернетика і системний аналіз. — 2014. — Vol. 50. — P. 271–277.

[14] Попов, Л.Д. Модификация метода Эрроу—Гурвица поиска седловых точек / Попов, Л.Д. // Математические заметки. — 1980. — Vol. 28. — P. 845–848.

Додаток А Реалізація

Наведемо реалізацію усіх згаданих алгоритмів на мові програмування python.

Зауваження. Ми обрали дизайн згідно з яким власне алгоритм знає мінімальний контекст задачі. Це означає, що для використання алгоритму користувач має визначити дві функції, одна з яких відповідатиме за обчислення оператора A, а друга — за обчислення оператора P_C. Це надає користувачеві гнучкість у плані вибору способу обчислення операторів, яка буде помітна вже з перших тестових запусків.

Загальний вигляд (за модулем назви і деяких параметрів) запуску алгоритма наступний:

```
solution, iteration_n, duration = korpelevich(
    x_initial=np.ones(size), lambda_=0.4,
    operator=lambda x: a.dot(x), projector=lambda x: x,
    tolerance=1e-3, max_iterations=1e4)
```

Як бачимо, визначення способу обчислення операторів A і P_C лягає на плечі користувача. У багатьох випадках це доволі просто, хоча у деяких користувачеві доведеться написати більше коду і знадобитсья користуватися scipy.optimize або аналогічним модулем для обчислення проекції.

Ось, наприклад, клієнтський код для другої задачі:

```
def ProjectionOntoProbabilitySymplex(x: np.array) -> np.array:
    dimensionality = x.shape[0]
    x /= dimensionality
    sorted_x = np.flip(np.sort(x))
    prefix_sum = np.cumsum(sorted_x)
    to_compare = sorted_x + (1 - prefix_sum) / np.arange(1, dimensionality + 1)
    k = 0
    for j in range(1, dimensionality): if to_compare[j] > 0: k = j
    return dimensionality * np.maximum(np.zeros(dimensionality),
                                       x + (to_compare[k] - sorted_x[k]))

solution, iteration_n, duration = korpelevich(...
    operator=lambda x: M.dot(x) + q,
    projector=ProjectionOntoProbabilitySymplex, ...)
```

A.1 Класичні алгоритми

Корпелевич

```
9  def korpelevich(x_initial: T,
10                 lambda_: float,
11                 tolerance: float = 1e-6,
12                 max_iterations: int = 1e4,
13                 operator: Callable[[T], T] = lambda x: x,
14                 projector: Callable[[T], T] = lambda x: x,
15                 **kwargs) -> Tuple[T, int, float]:
16     start = time.time()
17 
18     # initialization
19     iteration_number = 1
20     x_current, x_next = x_initial, None
21     y_current = None
22 
23     while True:
24         # step 1
25         y_current = projector(x_current - lambda_ * operator(x_current))
26 
27         # stopping criterion
28         if norm(x_current - y_current) < tolerance or \
29             iteration_number == max_iterations:
30             end = time.time()
31             duration = end - start
32             return x_current, iteration_number, duration
33 
34         # step 2
35         x_next = projector(x_current - lambda_ * operator(y_current))
36 
37         # next iteration
38         iteration_number += 1
39         x_current, x_next = x_next, None
40         y_current = None
```

Tseng

```python
def tseng(x_initial: T,
          lambda_: float,
          tolerance: float = 1e-6,
          max_iterations: int = 1e4,
          operator: Callable[[T], T] = lambda x: x,
          projector: Callable[[T], T] = lambda x: x,
          **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None

    while True:
        # step 1
        y_current = projector(x_current - lambda_ * operator(x_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        x_next = y_current - \
            lambda_ * (operator(y_current) - operator(x_current))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
```

Кешований Tseng

```
def cached_tseng(x_initial: T,
                 lambda_: float,
                 tolerance: float = 1e-6,
                 max_iterations: int = 1e4,
                 operator: Callable[[T], T] = lambda x: x,
                 projector: Callable[[T], T] = lambda x: x,
                 **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None

    while True:
        # step 1
        operator_x_current = operator(x_current)
        y_current = projector(x_current - lambda_ * operator_x_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        x_next = y_current - \
            lambda_ * (operator(y_current) - operator_x_current)

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
```

Попов

```
9  def popov(x_initial: T,
10           y_initial: T,
11           lambda_: float,
12           tolerance: float = 1e-6,
13           max_iterations: int = 1e4,
14           operator: Callable[[T], T] = lambda x: x,
15           projector: Callable[[T], T] = lambda x: x,
16           **kwargs) -> Tuple[T, int, float]:
17      start = time.time()
18
19      # initialization
20      iteration_number = 1
21      x_current, x_next = x_initial, None
22      y_previous, y_current = y_initial, None
23
24      while True:
25          # step 1
26          y_current = projector(x_current - lambda_ * operator(y_previous))
27
28          # step 2
29          x_next = projector(x_current - lambda_ * operator(y_current))
30
31          # stopping criterion
32          if norm(x_current - y_current) < tolerance and \
33              norm(x_next - y_current) < tolerance or \
34              iteration_number == max_iterations:
35              end = time.time()
36              duration = end - start
37              return x_current, iteration_number, duration
38
39          # next iteration
40          iteration_number += 1
41          x_current, x_next = x_next, None
42          y_previous, y_current = y_current, None
```

Кешований Попов

```python
def cached_popov(x_initial: T,
                 y_initial: T,
                 lambda_: float,
                 tolerance: float = 1e-6,
                 max_iterations: int = 1e4,
                 operator: Callable[[T], T] = lambda x: x,
                 projector: Callable[[T], T] = lambda x: x,
                 **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_previous, y_current = y_initial, None
    operator_y_previous, operator_y_current = operator(y_previous), None

    while True:
        # step 1
        y_current = projector(x_current - lambda_ * operator_y_previous)

        # step 2
        operator_y_current = operator(y_current)
        x_next = projector(x_current - lambda_ * operator_y_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance and \
            norm(x_next - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
        operator_y_previous, operator_y_current = operator_y_current, None
```

A.2 Адаптивні алгоритми

Адаптивний Корпелевич

```
9  def adaptive_korpelevich(x_initial: T,
10                           tau: float,
11                           lambda_initial: float,
12                           tolerance: float = 1e-6,
13                           max_iterations: int = 1e4,
14                           operator: Callable[[T], T] = lambda x: x,
15                           projector: Callable[[T], T] = lambda x: x,
16                           **kwargs) -> Tuple[T, int, float]:
17      start = time.time()
18
19      # initialization
20      iteration_number = 1
21      x_current, x_next = x_initial, None
22      y_current = None
23      lambda_current, lambda_next = lambda_initial, None
24
25      while True:
26          # step 1
27          y_current = projector(x_current - lambda_current * operator(x_current))
28
29          # stopping criterion
30          if norm(x_current - y_current) < tolerance or \
31              iteration_number == max_iterations:
32              end = time.time()
33              duration = end - start
34              return x_current, iteration_number, duration
35
36          # step 2
37          x_next = projector(x_current - lambda_current * operator(y_current))
38
39          # step 3
40          if (operator(x_current) - operator(y_current)).dot(x_next - y_current) <= 0:
41              lambda_next = lambda_current
42          else:
43              lambda_next = min(lambda_current, tau / 2 *
44                  (np.linalg.norm(x_current - y_current) ** 2 +
45                   np.linalg.norm(x_next - y_current) ** 2) /
46                      (operator(x_current) - operator(y_current)).dot(x_next - y_current))
47
48          # next iteration
49          iteration_number += 1
50          x_current, x_next = x_next, None
51          y_current = None
52          lambda_current, lambda_next = lambda_next, None
```

Кешований адаптивний Корпелевич

```
                                tau: float,
                                lambda_initial: float,
                                tolerance: float = 1e-6,
                                max_iterations: int = 1e4,
                                operator: Callable[[T], T] = lambda x: x,
                                projector: Callable[[T], T] = lambda x: x,
                                **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None
    lambda_current, lambda_next = lambda_initial, None
    operator_x_current, operator_y_current = None, None

    while True:
        # step 1
        operator_x_current = operator(x_current)
        y_current = projector(x_current - lambda_current * operator_x_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        operator_y_current = operator(y_current)
        x_next = projector(x_current - lambda_current * operator_y_current)

        # step 3
        product = (operator_x_current - operator_y_current).dot(x_next - y_current)
        if product <= 0:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau / 2 *
                (np.linalg.norm(x_current - y_current) ** 2 +
                 np.linalg.norm(x_next - y_current) ** 2) / product)

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
        lambda_current, lambda_next = lambda_next, None
        operator_x_current, operator_y_current = None, None
```

Адаптивний Tseng

```python
def adaptive_tseng(x_initial: T,
                   tau: float,
                   lambda_initial: float,
                   tolerance: float = 1e-6,
                   max_iterations: int = 1e4,
                   operator: Callable[[T], T] = lambda x: x,
                   projector: Callable[[T], T] = lambda x: x,
                   **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_current = None
    lambda_current, lambda_next = lambda_initial, None

    while True:
        # step 1
        y_current = projector(x_current - lambda_current * operator(x_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 2
        x_next = y_current - \
            lambda_current * (operator(y_current) - operator(x_current))

        # step 3
        if norm(operator(x_current) - operator(y_current)) < tolerance:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau *
                np.linalg.norm(x_current - y_current) /
                np.linalg.norm(operator(x_current) - operator(y_current)))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_current = None
        lambda_current, lambda_next = lambda_next, None
```

Кешований адаптивний Tseng

```
55  def cached_adaptive_tseng(x_initial: T,
56                            tau: float,
57                            lambda_initial: float,
58                            tolerance: float = 1e-6,
59                            max_iterations: int = 1e4,
60                            operator: Callable[[T], T] = lambda x: x,
61                            projector: Callable[[T], T] = lambda x: x,
62                            **kwargs) -> Tuple[T, int, float]:
63      start = time.time()
64
65      # initialization
66      iteration_number = 1
67      x_current, x_next = x_initial, None
68      y_current = None
69      lambda_current, lambda_next = lambda_initial, None
70      operator_x_current, operator_y_current = None, None
71
72      while True:
73          # step 1
74          operator_x_current = operator(x_current)
75          y_current = projector(x_current - lambda_current * operator_x_current)
76
77          # stopping criterion
78          if norm(x_current - y_current) < tolerance or \
79              iteration_number == max_iterations:
80              end = time.time()
81              duration = end - start
82              return x_current, iteration_number, duration
83
84          # step 2
85          operator_y_current = operator(y_current)
86          x_next = y_current - \
87              lambda_current * (operator_y_current - operator_x_current)
88
89          # step 3
90          if norm(operator_x_current - operator_y_current) < tolerance:
91              lambda_next = lambda_current
92          else:
93              lambda_next = min(lambda_current, tau *
94                  np.linalg.norm(x_current - y_current) /
95                  np.linalg.norm(operator_x_current - operator_y_current))
96
97          # next iteration
98          iteration_number += 1
99          x_current, x_next = x_next, None
100         y_current = None
101         lambda_current, lambda_next = lambda_next, None
102         operator_x_current, operator_y_current = None, None
```

Адаптивний Попов

```python
def adaptive_popov(x_initial: T,
                   y_initial: T,
                   tau: float,
                   lambda_initial: float,
                   tolerance: float = 1e-6,
                   max_iterations: int = 1e4,
                   operator: Callable[[T], T] = lambda x: x,
                   projector: Callable[[T], T] = lambda x: x,
                   **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_previous, y_current = y_initial, None
    lambda_current, lambda_next = lambda_initial, None

    while True:
        # step 1
        y_current = projector(x_current - lambda_current * operator(y_previous))

        # step 2
        x_next = projector(x_current - lambda_current * operator(y_current))

        # stopping criterion
        if norm(x_current - y_current) < tolerance and \
            norm(x_next - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 3
        if (operator(y_previous) - operator(y_current)).dot(x_next - y_current) <= 0:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau / 2 *
                (norm(y_previous - y_current) ** 2 +
                 norm(x_next - y_current) ** 2) /
                    (operator(y_previous) - operator(y_current)).dot(x_next - y_current))

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
        lambda_current, lambda_next = lambda_next, None
```

Кешований адаптивний Попов

```python
def cached_adaptive_popov(x_initial: T,
                          y_initial: T,
                          tau: float,
                          lambda_initial: float,
                          tolerance: float = 1e-6,
                          max_iterations: int = 1e4,
                          operator: Callable[[T], T] = lambda x: x,
                          projector: Callable[[T], T] = lambda x: x,
                          **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_current, x_next = x_initial, None
    y_previous, y_current = y_initial, None
    lambda_current, lambda_next = lambda_initial, None
    operator_y_previous, operator_y_current = operator(y_previous), None

    while True:
        # step 1
        y_current = projector(x_current - lambda_current * operator_y_previous)

        # step 2
        operator_y_current = operator(y_current)
        x_next = projector(x_current - lambda_current * operator_y_current)

        # stopping criterion
        if norm(x_current - y_current) < tolerance and \
            norm(x_next - y_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # step 3
        product = (operator_y_previous - operator_y_current).dot(x_next - y_current)
        if product <= 0:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau / 2 *
                (norm(y_previous - y_current) ** 2 +
                 norm(x_next - y_current) ** 2) / product)

        # next iteration
        iteration_number += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
        lambda_current, lambda_next = lambda_next, None
        operator_y_previous, operator_y_current = operator_y_current, None
```

A.3 Алгоритм Маліцького—Tam'a

Маліцький—Tam

```
9  def malitskyi_tam(x0_initial: T,
10                   x1_initial: T,
11                   lambda_: float,
12                   tolerance: float = 1e-6,
13                   max_iterations: int = 1e4,
14                   operator: Callable[[T], T] = lambda x: x,
15                   projector: Callable[[T], T] = lambda x: x,
16                   **kwargs) -> Tuple[T, int, float]:
17      start = time.time()
18
19      # initialization
20      iteration_number = 1
21      x_previous, x_current, x_next = x0_initial, x1_initial, None
22
23      while True:
24          # step
25          x_next = projector(x_current - lambda_ * operator(x_current) -
26              lambda_ * (operator(x_current) - operator(x_previous)))
27
28          # stopping criterion
29          if norm(x_current - x_previous) < tolerance and \
30              norm(x_next - x_current) < tolerance or \
31              iteration_number == max_iterations:
32              end = time.time()
33              duration = end - start
34              return x_current, iteration_number, duration
35
36          # next iteration
37          iteration_number += 1
38          x_previous, x_current, x_next = x_current, x_next, None
```

Кешований Маліцький—Tam

```python
def cached_malitskyi_tam(x0_initial: T,
                         x1_initial: T,
                         lambda_: float,
                         tolerance: float = 1e-6,
                         max_iterations: int = 1e4,
                         operator: Callable[[T], T] = lambda x: x,
                         projector: Callable[[T], T] = lambda x: x,
                         **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_number = 1
    x_previous, x_current, x_next = x0_initial, x1_initial, None
    operator_x_previous, operator_x_current = \
        operator(x_previous), operator(x_current)

    while True:
        # step
        x_next = projector(x_current - lambda_ * operator_x_current -
            lambda_ * (operator_x_current - operator_x_previous))

        # stopping criterion
        if norm(x_current - x_previous) < tolerance and \
            norm(x_next - x_current) < tolerance or \
            iteration_number == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_number, duration

        # next iteration
        iteration_number += 1
        operator_x_previous, operator_x_current = \
            operator_x_current, operator(x_next)
        x_previous, x_current, x_next = x_current, x_next, None
```

Адаптивний Маліцький—Tam

```python
def adaptive_malitskyi_tam(x0_initial: T,
                           x1_initial: T,
                           tau: float,
                           lambda0_initial: float,
                           lambda1_initial: float,
                           tolerance: float = 1e-6,
                           max_iterations: int = 1e4,
                           operator: Callable[[T], T] = lambda x: x,
                           projector: Callable[[T], T] = lambda x: x,
                           **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_n = 1
    x_previous, x_current, x_next = x0_initial, x1_initial, None
    lambda_previous, lambda_current, lambda_next = \
        lambda0_initial, lambda1_initial, None

    while True:
        # step 1
        x_next = projector(x_current - lambda_current * operator(x_current) -
            lambda_previous * (operator(x_current) - operator(x_previous)))

        # stopping criterion
        if norm(x_current - x_previous) < tolerance and \
            norm(x_next - x_current) < tolerance or \
            iteration_n == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_n, duration

        # step 2
        if norm(operator(x_next) - operator(x_current)) < tolerance:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau *
                norm(x_next - x_current) /
                norm(operator(x_next) - operator(x_current)))

        # next iteration
        iteration_n += 1
        x_previous, x_current, x_next = x_current, x_next, None
        lambda_previous, lambda_current, lambda_next = \
            lambda_current, lambda_next, None
```

Кешований адаптивний Маліцький—Tam

```python
def cached_adaptive_malitskyi_tam(x0_initial: T,
                                  x1_initial: T,
                                  tau: float,
                                  lambda0_initial: float,
                                  lambda1_initial: float,
                                  tolerance: float = 1e-6,
                                  max_iterations: int = 1e4,
                                  operator: Callable[[T], T] = lambda x: x,
                                  projector: Callable[[T], T] = lambda x: x,
                                  **kwargs) -> Tuple[T, int, float]:
    start = time.time()

    # initialization
    iteration_n = 1
    x_previous, x_current, x_next = x0_initial, x1_initial, None
    lambda_previous, lambda_current, lambda_next = \
        lambda0_initial, lambda1_initial, None
    operator_x_previous, operator_x_current, operator_x_next = \
        operator(x_previous), operator(x_current), None

    while True:
        # step 1
        x_next = projector(x_current - lambda_current * operator_x_current -
            lambda_previous * (operator_x_current - operator_x_previous))

        # stopping criterion
        if norm(x_current - x_previous) < tolerance and \
            norm(x_next - x_current) < tolerance or \
            iteration_n == max_iterations:
            end = time.time()
            duration = end - start
            return x_current, iteration_n, duration

        # step 2
        operator_x_next = operator(x_next)
        if norm(operator_x_next - operator_x_current) < tolerance:
            lambda_next = lambda_current
        else:
            lambda_next = min(lambda_current, tau *
                norm(x_next - x_current) /
                norm(operator_x_next - operator_x_current))

        # next iteration
        iteration_n += 1
        x_previous, x_current, x_next = x_current, x_next, None
        lambda_previous, lambda_current, lambda_next = \
            lambda_current, lambda_next, None
        operator_x_previous, operator_x_current, operator_x_next = \
            operator_x_current, operator_x_next, None
```