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

ЗМІСТ

Зміст

1 Вступ **3**

1.1 Варіаційна нерівність . . . 3

1.2 Зв'язок із задачами оптимізації . . . 3

1.3 Зв'язок із сідловими точками . . . 4

1.4 Ерроу та Гурвіц . . . 5

1.5 Сильно опуклі функції . . . 6

1.6 Монотонні оператори . . . 7

1.7 Регуляризація . . . 9

1.8 Зв'язок із мережевими іграми і рівновагою Неша . . . 9

1.9 Подальші припущення . . . 11

2 Прості алгоритми **12**

2.1 Алгоритми . . . 12

2.2 Реалізація простих алгоритмів . . . 14

3 Перша задача **20**

3.1 Задача . . . 20

3.2 Результати . . . 22

3.3 Розріджені матриці . . . 23

4 Друга задача **25**

4.1 Задача . . . 25

4.2 Результати . . . 26

5 Адаптивні алгоритми **28**

5.1 Алгоритми . . . 28

5.2 Реалізація адаптивних алгоритмів . . . 30

6 Результати адаптивних алгоритмів **36**

6.1 Перша задача . . . 36
6.2 Перша задача із розрідженими матрицями . . . 37
6.3 Друга задача . . . 39

7 Ще одна задача **41**
7.1 Задача . . . 41
7.2 Результати . . . 42
7.3 Розріджені матриці . . . 43

8 TODO **45**

1 Вступ

1.1 Варіаційна нерівність

Розглянемо абстрактний[1] 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

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

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

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

[1] Тобто поки що не накладаємо на нього ніяких обмежень і не вимагаємо від нього ніяких властивостей.

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[2] 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'ємоною 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[3] функції L.

[2] З опуклості C випливає, що $x + \varepsilon z \in C$, а отже можемо підставляти таке y.

[3] 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.

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

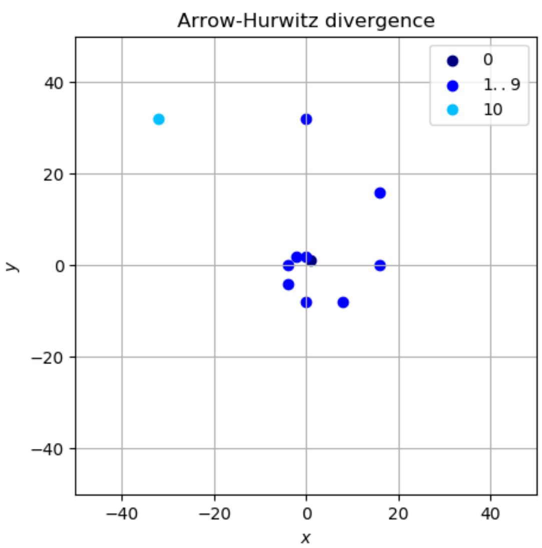
Приклад. Розглянемо тепер цілком конкретну функцію $L(x, y) = x \cdot y$ і спробуємо знайти її сідлову точку.

Розв'язок. Розглянемо алгоритм

$$\begin{aligned} x_{k+1} &\coloneqq x_k - \rho_k \nabla_1 L(x_k, y_k) = x_k - \rho_k y_k, \\ y_{k+1} &\coloneqq y_k + \rho_k \nabla_2 L(x_k, y_k) = y_k + \rho_k x_k, \end{aligned} \tag{1.13}$$

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 $|x_{k+1}|^2 + |y_{k+1}|^2$:

$$\begin{aligned} |x_{k+1}|^2 + |y_{k+1}|^2 = |x_k - \rho_k y_k|^2 + |y_k + \rho_k x_k|^2 = \\ = |x_k|^2 + \rho_k^2 \left(|x_k|^2 + |y_k|^2\right) + |y_k|^2 > |x_k|^2 + |y_k|^2, \end{aligned} \tag{1.14}$$

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

$$f(\alpha \cdot x + (1-\alpha) \cdot y) \le \alpha \cdot f(x) + (1-\alpha) \cdot f(y) - \mu \cdot \alpha \cdot (1-\alpha) \cdot \|x-y\|^2 . \tag{1.15}$$

Приклад. Довільна опукла (у звичайному розумінні) функція є 0-сильно опуклою.

Про всяк введемо також трохи більш узагальнене

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

$$f(\alpha \cdot x + (1-\alpha) \cdot y) \le \alpha \cdot f(x) + (1-\alpha) \cdot f(y) - \alpha \cdot (1-\alpha) \cdot g(\|x-y\|). \tag{1.16}$$

Можемо записати також альтернетивне визначення μ-сильно опуклості:

$$f(y) \ge f(x) + \langle \nabla f(x), y - x \rangle + \mu \cdot \|x-y\|^2 . \tag{1.17}$$

Зауваження. Просто цікаво, чи володіють якимись гарними властивостями “майже опуклі” функції, тобто μ-опуклі функції з $\mu < 0$.

Так от виявляється, що для задач із сильно опуклою функцією f метод Ерроу-Гурвіца збігається.

Доведення наведиться нижче у більш загальному випадку, а поки що останнє

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

$$(\tilde{x}_n, \tilde{y}_n) = \left(\frac{x_0 + \ldots + x_n}{n+1}, \frac{y_0 + \ldots + y_n}{n+1}\right) \tag{1.18}$$

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

$$\langle Ax, y - x\rangle \geq 0 \quad \forall y \in C, \tag{1.19}$$

де оператор A, взагалі кажучи, не нерозтягуючий.

Подивимося на цю задачу як на задачу знаходження нерухомої точки оператора

$$T : x \mapsto P_C\left(x - \rho A x\right), \tag{1.20}$$

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

$$x_{k+1} \coloneqq P_C\left(x_k - \rho_k A x_k\right), \tag{1.21}$$

збіжність якого ми зараз і проаналізуємо.

Взагалі хотілося б[4] щоб оператор T був нерозтягуючим. Маємо:

$$\begin{aligned} \|Tx - Ty\|^2 \leq \|x - y - \rho(Ax - Ay)\|^2 \leq \\ \leq \|x - y\|^2 - 2\rho\left\langle Ax - Ay, x - y\right\rangle + \rho^2\left\|Ax - Ay\right\|^2. \end{aligned} \tag{1.22}$$

Для подальших оцінок нам знадобиться поняття монотонного оператора.

[4] Відомо багато теорем щодо збіжності описаного алгоритму за таких умов.

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

$$\langle Ax - Ay, x - y \rangle \geq 0 \quad \forall x\, \forall y. \tag{1.23}$$

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 $A\colon H \to H$ називається *μ-сильно монотонним* зі сталою $\mu > 0$ якщо

$$\langle Ax - Ay, x - y \rangle \geq \mu \cdot \|x - y\|^2 \quad \forall x\, \forall y. \tag{1.24}$$

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

$$\begin{aligned} &\|x - y\|^2 - 2\rho \langle Ax - Ay, x - y \rangle + \rho^2 \|Ax - Ay\|^2 \leq \\ &\qquad \leq \|x - y\|^2 - 2\rho\mu \|x - y\|^2 + \rho^2 \|Ax - Ay\|^2 . \end{aligned} \tag{1.25}$$

Якщо ж при цьому оператор A ще й L-ліпшицевий[5], то можемо продовжити ще:

$$\begin{aligned} &\|x - y\|^2 - 2\rho\mu \|x - y\|^2 + \rho^2 \|Ax - Ay\|^2 \leq \\ &\qquad \leq \|x - y\|^2 - 2\rho\mu \|x - y\|^2 + \rho^2 L^2 \|x - y\|^2 = \\ &\qquad\qquad = (1 - 2\rho\mu + \rho^2 L^2) \|x - y\|^2 = \kappa(\rho) \cdot \|x - y\|^2 . \end{aligned} \tag{1.26}$$

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

Розв'язуючи отриману квадратну нерівність знаходимо:

$$\rho \in \left(0, \frac{2\mu}{L^2}\right), \tag{1.27}$$

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

[5] Ліпшицевий з константою L, тобто $\|Ax - Ay\| \leq L \cdot \|x - y\|$.

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

$$\tilde{\rho} = \frac{\mu}{L^2}, \tag{1.28}$$

і чому він дорівнює:

$$\kappa(\tilde{\rho}) = 1 - 2\rho\mu + \rho^2 L^2 = 1 - 2\frac{\mu}{L^2}\mu + \left(\frac{\mu}{L^2}\right)^2 L^2 = 1 - \frac{\mu^2}{L^2}. \tag{1.29}$$

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[6] на ε-сильно опуклу функцію $f_\varepsilon = f + \varepsilon \|x\|^2$, тому може здатися, що всі наші роблеми 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

$$\langle A_\varepsilon x - A_\varepsilon y, x - y\rangle = \underbrace{\langle Ax - Ay, x - y\rangle}_{\leq 0} + \varepsilon \cdot \|x - y\|^2 \geq \varepsilon \cdot \|x - y\|^2, \tag{1.30}$$

тобто оператор A_ε є ε-сильно опуклим.

Це наштовхує на думки про побудову алгоритму з ітераціями вигляду

$$x_{k+1} := P_C\left(x_k - \rho_k A_{\varepsilon_k} x_k\right), \tag{1.31}$$

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 [Asu Özdaglar, 2018].

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

[6] Цей процес називається регуляризацією.

- 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 $G \in \mathbb{R}^{n \times n}$:

$$\begin{cases} G_{i,j} > 0 & \text{вплив } j \text{ на } i, \\ G_{i,i} = 0 & \text{без петель.} \end{cases} \tag{1.32}$$

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:

$$x^i_{\text{br}}(z^i) \coloneqq \underset{x^i \in \mathcal{X}^i}{\operatorname{argmin}} J^i(x^i, z^i). \tag{1.33}$$

Нагадаємо

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

$$J^i\left(\bar{x}^i, \sum_{j=1}^n G_{i,j} \bar{x}^j\right) \leq J^i\left(x^i, \sum_{j=1}^m G_{i,j} \bar{x}^i\right), \quad \forall x^i \in \mathcal{X}^i. \tag{1.34}$$

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:

$$\mathcal{X} \coloneqq \mathcal{X}^1 \times \cdots \times \mathcal{X}^n; \tag{1.35}$$

$$F(x) \coloneqq [F^i(x)]_{i=1}^n \coloneqq \begin{bmatrix} \nabla_{x^1} J^1(x^1, z^1(x)) \\ \vdots \\ \nabla_{x^n} J^n(x^n, z^n(x)) \end{bmatrix} \tag{1.36}$$

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

1.9 Подальші припущення

Надалі будемо розв'язувати наступну задачу:

$$\text{знайти } x \in C : \quad \langle Ax, y - x \rangle \geq 0, \quad \forall y \in C. \tag{1.37}$$

Також будемо вважати, що виконані наступні умови:

- множина $C \subseteq H$ — опукла і замкнена;
- оператор $A : H \to H$ — монотонний і ліпшицевий (L — константа Ліпшиця);
- множина розв'язків (1.37) непорожня.

2 Прості алгоритми

2.1 Алгоритми

Серед різноманіття алгоритмів розв'язування (1.37) 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 зупиняємо алгоритм 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 зупиняємо алгоритм 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.**

2.2 Реалізація простих алгоритмів

Наведемо реалізацію цих алгоритмів на мові програмування python. Необхідні бібліотеки:

```
import numpy as np
import time

# pythonic way to implement generics (known as templates in C++)
from typing import Callable, TypeVar
T = TypeVar('T')
```

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.

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,
    A=lambda x: a.dot(x), ProjectionOntoC=lambda x: x,
    tolerance=1e-3, max_iterations=1e4, debug=True)
```

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 знадобиться користуватися scipy.optimize або аналогічним модулем для обчислення проекції.

Корпелевич

```
def korpelevich(x_initial: T,
                lambda_: float,
                A: Callable[[T], T],
                ProjectionOntoC: Callable[[T], T],
                tolerance: float = 1e-5,
                max_iterations: int = 1e4,
                debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial

    while True:
        # step 1
        y_current = ProjectionOntoC(x_current - lambda_ * A(x_current))

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

        # step 2
        x_next = ProjectionOntoC(x_current - lambda_ * A(y_current))

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

Tseng

```python
def tseng(x_initial: T,
          lambda_: float,
          A: Callable[[T], T],
          ProjectionOntoC: Callable[[T], T],
          tolerance: float = 1e-5,
          max_iterations: int = 1e4,
          debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial

    while True:
        # step 1
        y_current = ProjectionOntoC(x_current - lambda_ * A(x_current))

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

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

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

Кешований Tseng

```python
def cached_tseng(x_initial: T,
                 lambda_: float,
                 A: Callable[[T], T],
                 ProjectionOntoC: Callable[[T], T],
                 tolerance: float = 1e-5,
                 max_iterations: int = 1e4,
                 debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial

    while True:
        # step 1
        A_x_current = A(x_current)
        y_current = ProjectionOntoC(x_current - lambda_ * A_x_current)

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

        # step 2
        x_next = y_current - lambda_ * (A(y_current) - A_x_current)

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

Попов

```python
def popov(x_initial: T,
          y_initial: T,
          lambda_: float,
          A: Callable[[T], T],
          ProjectionOntoC: Callable[[T], T],
          tolerance: float = 1e-5,
          max_iterations: int = 1e4,
          debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial
    y_previous = y_initial

    while True:
        # step 1
        y_current = ProjectionOntoC(x_current - lambda_ * A(y_previous))

        # step 2
        x_next = ProjectionOntoC(x_current - lambda_ * A(y_current))

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance and
            np.linalg.norm(x_next - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

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

Кешований Попов

```python
def cached_popov(x_initial: T,
                 y_initial: T,
                 lambda_: float,
                 A: Callable[[T], T],
                 ProjectionOntoC: Callable[[T], T],
                 tolerance: float = 1e-5,
                 max_iterations: int = 1e4,
                 debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial
    y_previous = y_initial
    A_y_previous, A_y_current = A(y_previous), None

    while True:
        # step 1
        y_current = ProjectionOntoC(x_current - lambda_ * A_y_previous)

        # step 2
        A_y_current = A(y_current)
        x_next = ProjectionOntoC(x_current - lambda_ * A_y_current)

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance and
            np.linalg.norm(x_next - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

        # next iteration
        iteration_n += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
        A_y_previous, A_y_current = A_y_current, None
```

3 Перша задача

3.1 Задача

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

Задача 1. 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 декілька преших матриць, для $m = 2, 4, 6$:

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

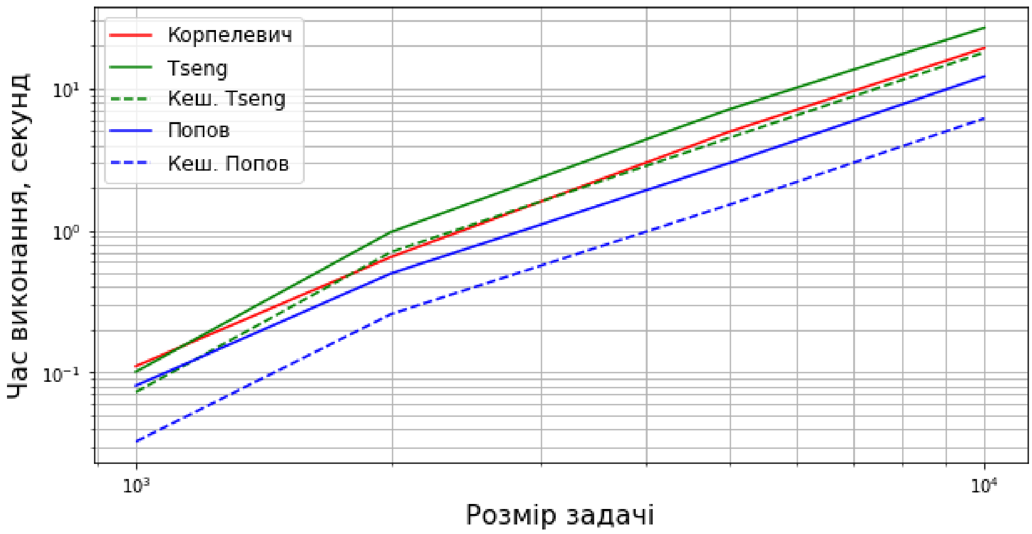
Для парних m нульовий вектор є розв'язком відповідної варіаційної нерівності (1.37).

Для усіх алгоритмів у якості початкового наближення ми брали $x_1 = (1, \ldots, 1)$, $\varepsilon = 10^{-3}$, $\lambda = 0.4$ (константа Ліпшиця цієї задачі дорівнює одиниці: $L = 1$).

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, що ми скоро і побачимо.

3.2 Результати

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



І справді, бачимо що алгоритм Корпелевич і кешований алгоритм Tseng'a справді показують майже однакові результати, а також обидві некешовані версії програють кешованим. Та сама інформація у табличці, для зручності:

Розмір задачі	1000	2000	5000	10000
Корпелевич	0.11	0.65	4.95	19.31
Tseng	0.10	0.98	7.13	26.82
Кеш. Tseng	0.07	0.71	4.49	17.98
Попов	0.08	0.50	2.98	12.18
Кеш. Попов	0.03	0.26	1.52	6.16

Таблиця 3.1: Час виконання, секунд

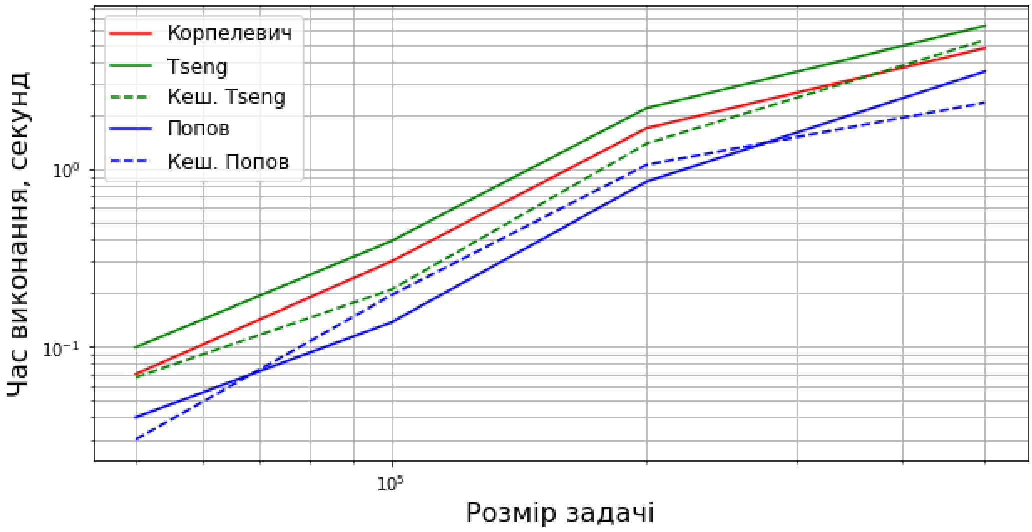
Розмір задачі	1000	2000	5000	10000
Корпелевич = Tseng	132	137	144	148
Попов	89	92	96	99

Таблиця 3.2: Число ітерацій

Зауваження. Наша реалізація приблизно у 50 разів швидша за результати наведені у статті [Yura Malitsky, 2015].

3.3 Розріджені матриці

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.



Та сама інформація у табличці, для зручності:

Розмір задачі	50000	100000	200000	500000
Корпелевич	0.07	0.30	1.69	4.77
Tseng	0.10	0.39	2.19	6.35
Кеш. Tseng	0.07	0.21	1.39	5.27
Попов	0.04	0.14	0.85	3.53
Кеш. Попов	0.03	0.19	1.06	2.35

Таблиця 3.3: Час виконання, секунд

Розмір задачі	50000	100000	200000	500000
Корпелевич = Tseng	159	164	169	175
Попов	106	109	112	117

Таблиця 3.4: Число ітерацій

Зауваження. Тут перевага кешування вже не така очевидна, адже ми значно здешевили обчислення оператора A, хоча все ще присутня.

4 Друга задача

4.1 Задача

Задача 2. Візьмемо $F(x) = Mx + q$, де матриця M генерується наступний чином:

$$M = AA^{\intercal} + B + D, \tag{4.1}$$

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

$$C = \left\{x \in \mathbb{R}^m_+ | x_1 + x_2 + \cdots + x_m = m\right\}, \tag{4.2}$$

а за початкове наближення береться $x_1 = (1, \ldots, 1)$. Для цієї задачі $L = \|M\|$, $\varepsilon = 10^{-3}$.

Алгоритм проектування

Алгоритм 4.

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

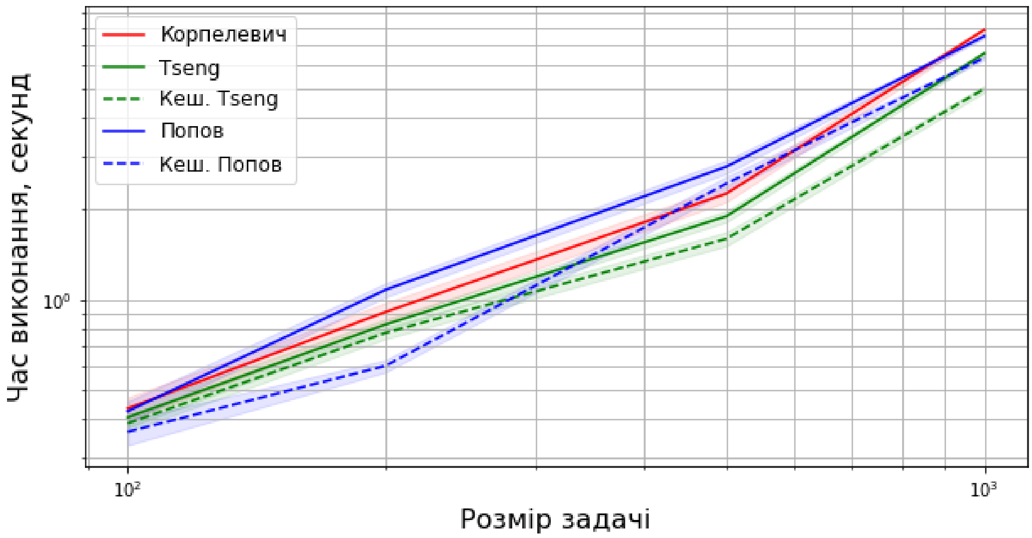
Цей алгоритм взято із статті [Weiran Wang, Miguel A. Carreira-Perpiñán, 2013], хоча він зустрічається у літературі і раніше.

Приклад клієнтського коду

```
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(...
    A=lambda x: M.dot(x) + q,
    ProjectionOntoC=ProjectionOntoProbabilitySymplex, ...)
```

4.2 Результати

Для кожного алгоритму і кожного розміру задачі було проведено 5 запусків (із різними матрицями), у таблиці і на графіку наведені середні значення та середньоквадратичні відхилення.



Та сама інформація у табличці, для зручності:

Розмір задачі	100	200	500	1000
Корпелевич	0.43 ± 0.19	0.90 ± 0.33	2.24 ± 0.68	7.86 ± 0.83
Tseng	0.40 ± 0.15	0.82 ± 0.17	1.89 ± 0.40	6.56 ± 0.68
Кеш. Tseng	0.38 ± 0.11	0.77 ± 0.17	1.59 ± 0.42	5.00 ± 0.63
Попов	0.42 ± 0.17	1.07 ± 0.28	2.76 ± 0.70	7.49 ± 0.46
Кеш. Попов	0.36 ± 0.17	0.60 ± 0.14	2.42 ± 0.67	6.35 ± 0.46

Таблиця 4.1: Час виконання, секунд

У цій задачі основна складність все ще у обчисленні оператора A, хоча обчислення проекції вже більш складне, тому алгоритм Tseng'а має певну перевагу над алгоритмом Попова, який у свою чергу випереджає алгоритм Корпелевич. Щодо кількості ітерацій то усі три алгоритми демонструють практично ідентичні результати.

Розмір задачі	100	200	500	1000
Корпелевич	1176 ± 162	1743 ± 87	2694 ± 139	3681 ± 191
Tseng	1176 ± 162	1743 ± 87	2694 ± 139	3681 ± 191
Попов	1176 ± 162	1743 ± 87	2694 ± 139	3681 ± 191

Таблиця 4.2: Число ітерацій

Знову ж таки, кешування дає перевагу на великих задачах, хоча вона вже не у 1.5–2 рази.

Зауваження. З певних поки що незрозумілих містичних причин кількість ітерацій усіх алгоритмів виходить однакова, хоча вони ніби як не еквівалентні для цієї задачі.

Зауваження. Можна додати якийсь із матрчиних розкладів M для пришвидшення множення Mx.

Зауваження. Наша реалізація приблизно у 2000 разів швидша за результати наведені у статті [Yura Malitsky, 2015].

5 Адаптивні алгоритми

5.1 Алгоритми

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

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

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{5.1}$$

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

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda A y_n). \tag{5.2}$$

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } \langle Ax_n - Ay_n, x_{n+1} - y_n \rangle \le 0, \\ \min \left\{ \lambda_n, \dfrac{\tau}{2} \dfrac{\|x_n - y_n\|^2 + \|x_{n+1} - y_n\|^2}{\langle Ax_n - Ay_n, x_{n+1} - y_n \rangle} \right\}, \text{ інакше.} \end{cases} \tag{5.3}$$

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

Зауваження. У алгоритмі 5 можна робити і так:

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ax_n - Ay_n = 0, \\ \min \left\{ \lambda_n, \tau \dfrac{\|x_n - y_n\|}{\|Ax_n - Ay_n\|} \right\}, \text{ інакше.} \end{cases} \tag{5.4}$$

Алгоритм 6 (Адаптивний Tseng)**. Ініціалізація.** Вибираємо елементи x_1, $\tau \in (0, 1)$, $\lambda \in (0, +\infty)$. Покладаємо $n = 1$.

Крок 1. Обчислюємо

$$y_n = P_C(x_n - \lambda A x_n). \tag{5.5}$$

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

Крок 2. Обчислюємо

$$x_{n+1} = y_n - \lambda(Ay_n - Ax_n), \tag{5.6}$$

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ax_n - Ay_n = 0, \\ \min\left\{\lambda_n, \tau\dfrac{\|x_n - y_n\|}{\|Ax_n - Ay_n\|}\right\}, \text{ інакше.} \end{cases} \tag{5.7}$$

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

Алгоритм 7 (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

$$y_n = P_C(x_n - \lambda Ay_{n-1}). \tag{5.8}$$

Крок 2. Обчислюємо

$$x_{n+1} = P_C(x_n - \lambda Ay_n). \tag{5.9}$$

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

Крок 3. Обчислюємо

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } \langle Ay_{n-1} - Ay_n, x_{n+1} - y_n\rangle \le 0, \\ \min\left\{\lambda_n, \dfrac{\tau}{2}\dfrac{\|y_{n-1} - y_n\|^2 + \|x_{n+1} - y_n\|^2}{\langle Ay_{n-1} - Ay_n, x_{n+1} - y_n\rangle}\right\}, \text{ інакше.} \end{cases} \tag{5.10}$$

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

Зауваження. У алгоритмі 7 можна робити і так:

$$\lambda_{n+1} = \begin{cases} \lambda_n, \text{ якщо } Ay_{n-1} - Ay_n = 0, \\ \min\left\{\lambda_n, \tau\dfrac{\|y_{n-1} - y_n\|}{\|Ay_{n-1} - Ay_n\|}\right\}, \text{ інакше.} \end{cases} \tag{5.11}$$

5.2 Реалізація адаптивних алгоритмів

Адаптивний Корпелевич

```python
def adaptive_korpelevich(x_initial: T,
                         tau: float,
                         lambda_initial: float,
                         A: Callable[[T], T],
                         ProjectionOntoC: Callable[[T], T],
                         tolerance: float = 1e-5,
                         max_iterations: int = 1e4,
                         debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial
    lambda_current = lambda_initial

    while True:
        # step 1
        y_current = ProjectionOntoC(x_current - lambda_current * A(x_current))

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

        # step 2
        x_next = ProjectionOntoC(x_current - lambda_current * A(y_current))

        # step 3
        if (A(x_current) - A(y_current)).dot(x_next - y_current) <= 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) /
                    (A(x_current) - A(y_current)).dot(x_next - y_current))

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

Кешований адаптивний Корпелевич

```python
def cached_adaptive_korpelevich(x_initial: T,
                                tau: float,
                                lambda_initial: float,
                                A: Callable[[T], T],
                                ProjectionOntoC: Callable[[T], T],
                                tolerance: float = 1e-5,
                                max_iterations: int = 1e4,
                                debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial
    lambda_current = lambda_initial
    A_x_current, A_y_current = None, None

    while True:
        # step 1
        A_x_current = A(x_current)
        y_current = ProjectionOntoC(x_current - lambda_current * A_x_current)

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

        # step 2
        A_y_current = A(y_current)
        x_next = ProjectionOntoC(x_current - lambda_current * A_y_current)

        # step 3
        product = (A_x_current - A_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_n += 1
        x_current, x_next = x_next, None
        y_current = None
        lambda_current, lambda_next = lambda_next, None
        A_x_current, A_y_current = None, None
```

Адаптивний Tseng

```python
def adaptive_tseng(x_initial: T,
                   tau: float,
                   lambda_initial: float,
                   A: Callable[[T], T],
                   ProjectionOntoC: Callable[[T], T],
                   tolerance: float = 1e-5,
                   max_iterations: int = 1e4,
                   debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial
    lambda_current = lambda_initial

    while True:
        # step 1
        y_current = ProjectionOntoC(x_current - lambda_current * A(x_current))

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

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

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

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

Кешований адаптивний Tseng

```python
def cached_adaptive_tseng(x_initial: T,
                          tau: float,
                          lambda_initial: float,
                          A: Callable[[T], T],
                          ProjectionOntoC: Callable[[T], T],
                          tolerance: float = 1e-5,
                          max_iterations: int = 1e4,
                          debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial
    lambda_current = lambda_initial
    A_x_current = None
    A_y_current = None

    while True:
        # step 1
        A_x_current = A(x_current)
        y_current = ProjectionOntoC(x_current - lambda_current * A_x_current)

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

        # step 2
        A_y_current = A(y_current)
        x_next = y_current - lambda_current * (A_y_current - A_x_current)

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

        # next iteration
        iteration_n += 1
        x_current, x_next = x_next, None
        y_current = None
        lambda_current, lambda_next = lambda_next, None
        A_x_current, A_y_current = None, None
```

Адаптивний Попов

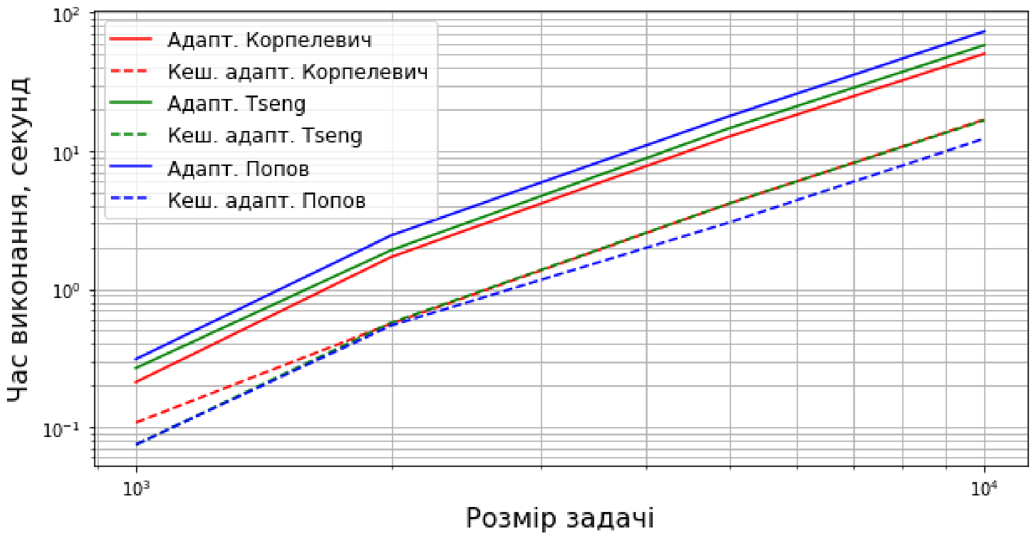
```python
def adaptive_popov(x_initial: T,
                   y_initial: T,
                   tau: float,
                   lambda_initial: float,
                   A: Callable[[T], T],
                   ProjectionOntoC: Callable[[T], T],
                   tolerance: float = 1e-5,
                   max_iterations: int = 1e4,
                   debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial
    y_previous = y_initial
    lambda_current = lambda_initial

    while True:
        # step 1
        y_current = ProjectionOntoC(x_current - lambda_current * A(y_previous))

        # step 2
        x_next = ProjectionOntoC(x_current - lambda_current * A(y_current))

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance and
            np.linalg.norm(x_next - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

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

        # next iteration
        iteration_n += 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,
                          A: Callable[[T], T],
                          ProjectionOntoC: Callable[[T], T],
                          tolerance: float = 1e-5,
                          max_iterations: int = 1e4,
                          debug: bool = False) -> T:
    start = time.time()

    # initialization
    iteration_n = 1
    x_current = x_initial
    y_previous = y_initial
    lambda_current = lambda_initial
    A_y_previous, A_y_current = A(y_previous), None

    while True:
        # step 1
        y_current = ProjectionOntoC(x_current - lambda_current * A_y_previous)

        # step 2
        A_y_current = A(y_current)
        x_next = ProjectionOntoC(x_current - lambda_current * A_y_current)

        # stopping criterion
        if (np.linalg.norm(x_current - y_current) < tolerance and
            np.linalg.norm(x_next - y_current) < tolerance or
            iteration_n == max_iterations):
            if debug:
                end = time.time()
                duration = end - start
                print(f'Took {iteration_n} iterations '
                      f'and {duration:.2f} seconds to converge.')
                return x_current, iteration_n, duration
            return x_current

        # step 3
        product = (A_y_previous - A_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(y_previous - y_current) ** 2 +
                 np.linalg.norm(x_next - y_current) ** 2) / product)

        # next iteration
        iteration_n += 1
        x_current, x_next = x_next, None
        y_previous, y_current = y_current, None
        lambda_current, lambda_next = lambda_next, None
        A_y_previous, A_y_current = A_y_current, None
```

6 Результати адаптивних алгоритмів

6.1 Перша задача



Та сама інформація у табличці, для зручності:

Розмір задачі	1000	2000	5000	10000
Корпелевич	0.11	0.65	4.95	19.31
Tseng	0.10	0.98	7.13	26.82
Кеш. Tseng	0.07	0.71	4.49	17.98
Попов	0.08	0.50	2.98	12.18
Кеш. Попов	0.03	0.26	1.52	6.16
Адапт. Корпелевич	0.21	1.71	12.70	50.57
Кеш. адапт. Корпелевич	0.11	0.56	4.15	16.89
Адапт. Tseng	0.27	1.91	14.58	58.36
Кеш. адапт. Tseng	0.07	0.57	4.16	16.69
Адапт. Попов	0.31	2.45	17.84	73.44
Кеш. адапт. Попов	0.07	0.54	3.03	12.29

Таблиця 6.1: Час виконання, секунд

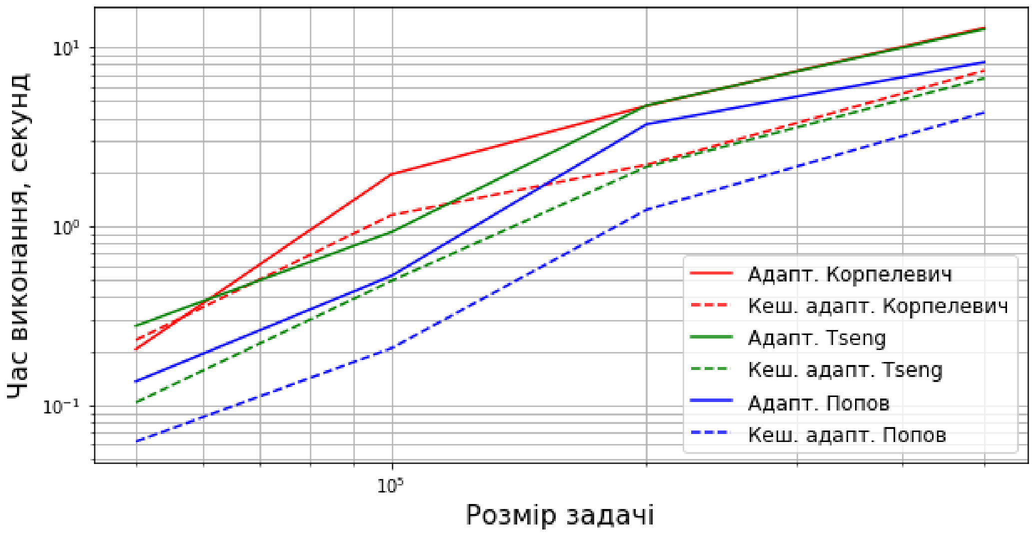
Алгоритма Попова програє своїй неадаптивній версії. Окрім цього, некешовані версії адаптивних алгоритмів явно програють кешованим. Кешовані версії адаптивних алгоритмів Корпелевич і Tseng'а не поступаються кешованим неадаптивним версіям.

Щодо кількості ітерацій ситуація схожа:

Розмір задачі	1000	2000	5000	10000
Корпелевич = Tseng	132	137	144	148
Попов	89	92	96	99
Адапт. Корпелевич	125	129	135	139
Адапт. Tseng	125	129	135	139
Адапт. Попов	179	185	194	201

Таблиця 6.2: Число ітерацій

6.2 Перша задача із розрідженими матрицями



Та сама інформація у табличці:

Розмір задачі	50000	100000	200000	500000
Корпелевич	0.07	0.30	1.69	4.77
Tseng	0.10	0.39	2.19	6.35
Кеш. Tseng	0.07	0.21	1.39	5.27
Попов	0.04	0.14	0.85	3.53
Кеш. Попов	0.03	0.19	1.06	2.35
Адапт. Корпелевич	0.20	1.94	4.69	12.81
Кеш. адапт. Корпелевич	0.23	1.15	2.20	7.38
Адапт. Tseng	0.28	0.93	4.72	12.61
Кеш. адапт. Tseng	0.10	0.49	2.14	6.69
Адапт. Попов	0.14	0.53	3.71	8.24
Кеш. адапт. Попов	0.06	0.21	1.23	4.31

Таблиця 6.3: Час виконання, секунд

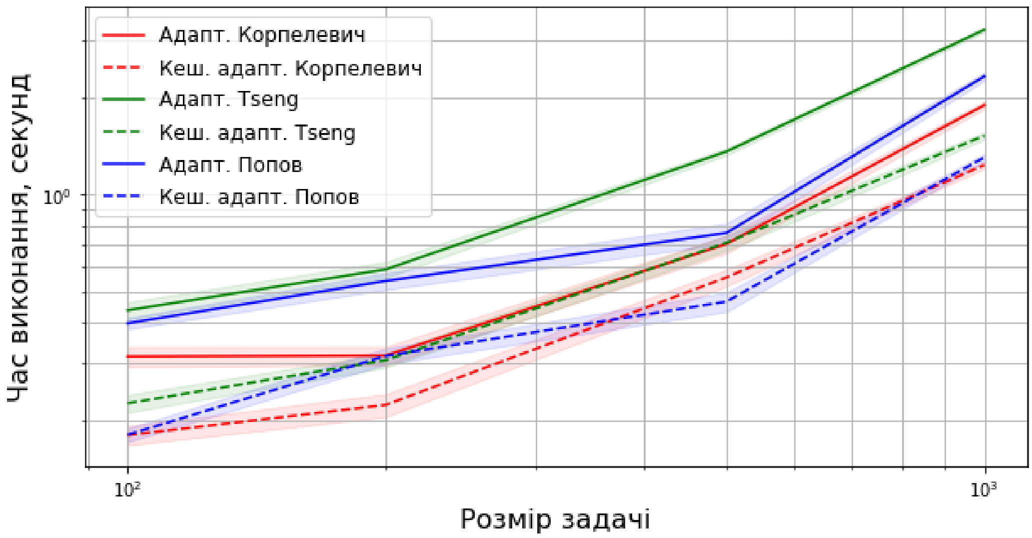
Розмір задачі	50000	100000	200000	500000
Корпелевич = Tseng	159	164	169	175
Попов	106	109	112	117
Адапт. Корпелевич	160	165	170	176
Адапт. Tseng	160	165	170	176
Адапт. Попов	108	111	114	118

Таблиця 6.4: Число ітерацій

Ситуація доволі схожа на попередню, за виключення того що алгоритм Попва тепер не так суттєво програє неадаптивній версії.

Зауваження. З невідомих містичних причин від переходу на розріджені матриці алгоритм Попова починає збігаатися за меншу кількість ітерацій для задачі більшого розміру.

6.3 Друга задача



Та сама інформація у табличці:

Розмір задачі	100	200	500	1000
Корпелевич	0.43 ± 0.19	0.90 ± 0.33	2.24 ± 0.68	7.86 ± 0.83
Tseng	0.40 ± 0.15	0.82 ± 0.17	1.89 ± 0.40	6.56 ± 0.68
Кеш. Tseng	0.38 ± 0.11	0.77 ± 0.17	1.59 ± 0.42	5.00 ± 0.63
Попов	0.42 ± 0.17	1.07 ± 0.28	2.76 ± 0.70	7.49 ± 0.46
Кеш. Попов	0.36 ± 0.17	0.60 ± 0.14	2.42 ± 0.67	6.35 ± 0.46
Адапт. Корпелевич	0.31 ± 0.11	0.31 ± 0.10	0.70 ± 0.22	1.89 ± 0.24
Кеш. адапт. Корпелевич	0.18 ± 0.06	0.22 ± 0.09	0.55 ± 0.16	1.23 ± 0.14
Адапт. Tseng	0.43 ± 0.13	0.58 ± 0.16	1.35 ± 0.16	3.24 ± 0.24
Кеш. адапт. Tseng	0.22 ± 0.07	0.30 ± 0.07	0.71 ± 0.18	1.52 ± 0.19
Адапт. Попов	0.39 ± 0.07	0.54 ± 0.16	0.76 ± 0.27	2.32 ± 0.33
Кеш. адапт. Попов	0.18 ± 0.04	0.31 ± 0.09	0.46 ± 0.16	1.30 ± 0.17

Таблиця 6.5: Час виконання, секунд

Адаптивні версії суттєво випереджають неадаптивні, причому за числом ітерацій також:

Розмір задачі	100	200	500	1000
Корпелевич	1176 ± 162	1743 ± 87	2694 ± 139	3681 ± 191
Tseng	1176 ± 162	1743 ± 87	2694 ± 139	3681 ± 191
Попов	1176 ± 162	1743 ± 87	2694 ± 139	3681 ± 191
Адапт. Корпелевич	317 ± 66	359 ± 42	410 ± 34	451 ± 46
Адапт. Tseng	504 ± 50	684 ± 38	872 ± 73	994 ± 68
Адапт. Попов	430 ± 93	507 ± 64	551 ± 48	606 ± 57

Таблиця 6.6: Число ітерацій

7 Ще одна задача

7.1 Задача

Задача 3.

$$\begin{aligned}
F(x) &= F_1(x) + F_2(x), \\
F_1(x) &= (f_1(x), f_2(x), \dots, f_m(x)), \\
F_2(x) &= Dx + c, \\
f_i(x) &= x_{i-1}^2 + x_i^2 + x_{i-1}x_i + x_i x_{i+1}, \quad m = 1, 2, \dots, m, \\
x_0 &= x_{m+1} = 0,
\end{aligned} \tag{7.1}$$

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

$$d_{i,j} = \begin{cases} 1, & j = i - 1, \\ 4, & j = i, \\ -2, & j = i + 1, \\ 0, & \text{інакше}, \end{cases} \tag{7.2}$$

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

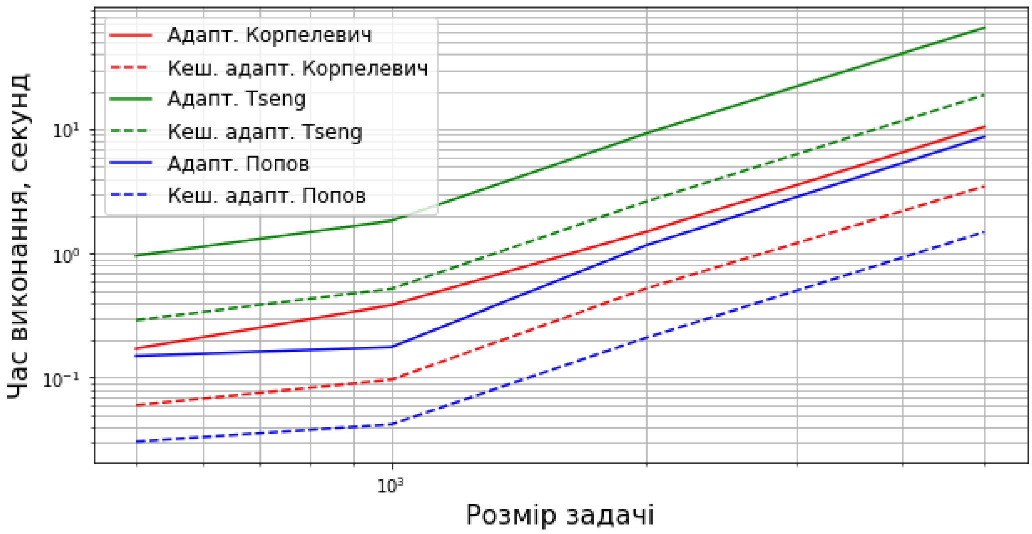
Для кращого розуміння наведемо матрицю D для кількох перших $m = 3, 4, 5$:

$$\begin{pmatrix} 4 & -2 & 0 \\ 1 & 4 & -2 \\ 0 & 1 & 4 \end{pmatrix} \qquad \begin{pmatrix} 4 & -2 & 0 & 0 \\ 1 & 4 & -2 & 0 \\ 0 & 1 & 4 & -2 \\ 0 & 0 & 1 & 4 \end{pmatrix} \qquad \begin{pmatrix} 4 & -2 & 0 & 0 & 0 \\ 1 & 4 & -2 & 0 & 0 \\ 0 & 1 & 4 & -2 & 0 \\ 0 & 0 & 1 & 4 & -2 \\ 0 & 0 & 0 & 1 & 4 \end{pmatrix} \tag{7.3}$$

Зауваження. Як бачимо, матриця тридіагональна, чим ми скоро скористаємося.

7.2 Результати

А цій задачі константа Ліпшиця мені невідома, тому тут наводяться результати лише адаптивних алгоритмів.



Та сама інформація у табличці:

Розмір задачі	500	1000	2000	5000
Адапт. Корпелевич	0.17	0.38	1.50	10.43
Кеш. адапт. Корпелевич	0.06	0.10	0.52	3.46
Адапт. Tseng	0.96	1.83	9.26	65.36
Кеш. адапт. Tseng	0.29	0.52	2.62	18.81
Адапт. Попов	0.15	0.18	1.17	8.68
Кеш. адапт. Попов	0.03	0.04	0.21	1.49

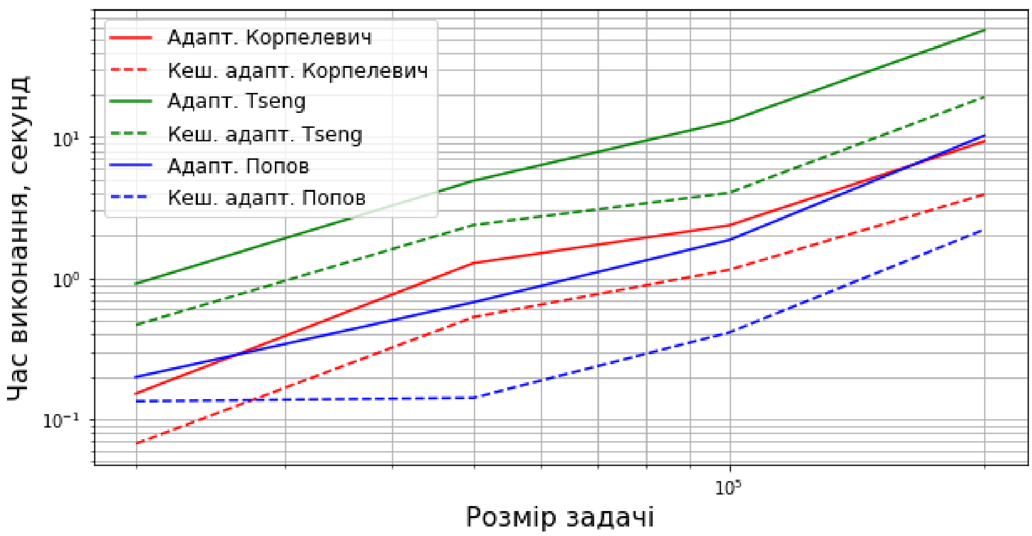
Таблиця 7.1: Час виконання, секунд

Зауваження. Наша реалізація приблизно у 100 разів швидша за результати наведені у статті [Yura Malitsky, 2015].

Розмір задачі	500	1000	2000	5000
Адапт. Корпелевич	111	113	116	119
Адапт. Tseng	558	572	587	605
Адапт. Попов	87	89	91	94

Таблиця 7.2: Число ітерацій

7.3 Розріджені матриці



Та сама інформація у табличці:

Розмір задачі	20000	50000	100000	200000
Адапт. Корпелевич	0.15	1.28	2.36	9.33
Кеш. адапт. Корпелевич	0.07	0.53	1.14	3.92
Адапт. Tseng	0.91	4.90	12.94	57.32
Кеш. адапт. Tseng	0.47	2.38	4.02	19.24
Адапт. Попов	0.20	0.67	1.86	10.24
Кеш. адапт. Попов	0.13	0.14	0.41	2.21

Таблиця 7.3: Час виконання, секунд

Розмір задачі	20000	50000	100000	200000
Адапт. Корпелевич	74	76	77	79
Адапт. Tseng	388	399	408	416
Адапт. Попов	71	73	74	76

Таблиця 7.4: Число ітерацій

Зауваження. Чомусь на цій задачі алгоритм Tseng'а явно просідає. Причини цього явища поки що не з'ясовані.

Зауваження. З містичних причин від переходу на розріджені матриці усі алгоритми починають збігаатися за меншу кількість ітерацій для задачі більшого розміру.

8 TODO

1. алгоритми Юри
2. задача з інтегральним оператором з NUMA-D-20-00210_reviewer.pdf
3. зв’язок з транспортними мережами
4. пройтися по всім зауваженням — або видалити або розприсати нормально
5. вирівняти криві графіки усередненням по 5 (?) запускам
6. реферат
7. література
8. репетиція презентації/доповіді