

Міністерство освіти і науки України  
Державний заклад  
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут фізики, математики та інформаційних  
технологій

Кафедра інформаційних технологій та систем

**Семенов Микола Анатолійович**

**РОЗРОБКА ТА ДОСЛІДЖЕННЯ СПОСОБІВ ПРЕДСТАВЛЕННЯ  
СИСТЕМ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ ЗА ДОПОМОГОЮ  
ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ**

**кваліфікаційна робота**

**здобувача вищої освіти другого (магістерського) рівня**

**освітньої програми «Мультимедійні системи»**

**за спеціальністю 121 Інженерія програмного забезпечення**

Особистий підпис \_\_\_\_\_ Микола СЕМЕНОВ

Науковий керівник \_\_\_\_\_ Геннадій МОГИЛЬНИЙ,  
кандидат технічних наук, доцент  
кафедри інформаційних технологій  
та систем

Завідувач кафедри \_\_\_\_\_ Микола СЕМЕНОВ,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій  
та систем

Полтава – 2023

## **АНОТАЦІЯ**

**Семенов М.А.**

**Тема:** Розробка та дослідження способів представлення систем лінійних алгебраїчних рівнянь за допомогою об'єктно-орієнтованого програмування

**Спеціальність:** 121 «Інженерія програмного забезпечення».

**Установа:** ЛНУ імені Тараса Шевченка, 2023р.

**Магістерська робота містить:** 114 с., 29 рис., 13 табл., 65 джерел.

**Об'єкт дослідження** – оптимізація представлення інформації в пам'яті комп'ютера для обчислень лінійної алгебри.

**Предмет дослідження** – розробка об'єктно-орієнтованих бібліотек класів для ефективного зберігання та обробки матриць у пам'яті та розв'язання систем лінійних алгебраїчних рівнянь.

**Мета роботи** - розробка структур даних, у яких реалізовані ефективні методи зберігання та обробки матриць та векторів у пам'яті комп'ютера, а також реалізація алгоритмів розв'язання СЛАР на їх основі.

**Результати роботи** – в роботі виконано огляд практичної реалізації структур даних для широкого класу матриць, які дозволяють ефективно зберігати та обробляти матрицю у пам'яті. Реалізовано проекційні методи вирішення СЛАР підпростору Крилова на основі структур даних, які розроблені за допомогою об'єктно-орієнтованого програмування. Досліджено ефективність проекційних методів рішення з різними передумовами.

**Ключові слова:** СИСТЕМИ ЛІНІЙНИХ АЛГЕБРАІЧНИХ РІВНЯНЬ, ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ; СТРУКТУРИ ДАНИХ, МЕТОДИ ПРЕДСТАВЛЕННЯ ДАНИХ, МЕТОДИ ЗБЕРІГАННЯ ДАНИХ, МЕТОДИ ОПТИМІЗАЦІЇ ЗБЕРІГАННЯ ДАНИХ

## ANNOTATION

**Semenov Mykola**

**Theme:** Development and research of methods of representing systems of linear algebraic systems using object-oriented programming.

**Speciality:** 121 " Software engineering ".

**Institution:** Luhansk Taras Shevchenko National University (LTSNU), 2023 year.

**Master's work of:** 114 p., 29 im, 13 tbl, 65 sources.

**The object of the research** is the optimization of the representation of information in the computer memory for linear algebra calculations.

**The subject of research** is the development of object-oriented class libraries for efficient storage and processing of matrices in memory and solving systems of linear algebraic equations.

**The purpose of the thesis** is the development of data structures that implement effective methods of storing and processing matrices and vectors in computer memory, as well as the implementation of algorithms for solving systems of linear algebraic equations based on them.

**The results** include an overview of the practical implementation of data structures for a wide class of matrices, which allow efficient storage and processing of the matrix in memory. Projective methods for solving the systems of linear algebraic equations of the Krylov subspace based on data structures developed using object-oriented programming have been implemented. The effectiveness of projection methods of the solution with different prerequisites was investigated.

**Keywords:** SYSTEMS OF LINEAR ALGEBRAIC EQUATIONS, OBJECT-ORIENTED PROGRAMMING; DATA STRUCTURES, DATA REPRESENTATION METHODS, DATA STORAGE METHODS, DATA STORAGE OPTIMIZATION METHODS.

## **ЗМІСТ**

<b>ВСТУП .....</b>	<b>6</b>
<b>РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ ТЕХНОЛОГІЙ ЗБЕРІГАННЯ МАТРИЦЬ.....</b>	<b>9</b>
1.1. Формулювання критерій подання матриць.....	9
1.2. Зберігання масивів, списків, стеків та черг .....	12
1.3. Зберігання цілих списків .....	16
1.4. Подання та зберігання графів .....	19
1.5. Схеми зберігання матриць великої розмірності.....	23
1.6. Символічна обробка та динамічні схеми зберігання .....	37
Висновки до першого розділу .....	41
<b>РОЗДІЛ 2. МАТЕМАТИЧНЕ ПРЕДСТАВЛЕННЯ МАТРИЧНИХ АЛГОРИТМІВ .....</b>	<b>43</b>
2.1. Принципи побудови ітераційних процесів .....	43
2.2. Побудова проєкційних методів.....	46
2.3. Підпростори Крилова .....	52
2.3.1. Метод спряжених градієнтів.....	54
2.3.2. Метод біспряжених градієнтів.....	56
2.4. Передумовлення .....	57
2.5. ILU-факторизація.....	62
Висновки до другого розділу .....	66
<b>РОЗДІЛ 3. РОЗРОБКА БІБЛІОТЕКИ КЛАСІВ ДЛЯ ПРЕДСТАВЛЕННЯ СЛАР .....</b>	<b>68</b>
3.1. Використання ООП підходу до подання СЛАР .....	68
3.2. Реалізація моделі бібліотеки класів.....	69
3.3. Програмна реалізація класів .....	72
3.4. Дослідження ефективності методів вирішення представлених у розробленій бібліотеці.....	85
Висновки до третього розділу .....	88
<b>ВИСНОВКИ .....</b>	<b>89</b>

<b>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....</b>	<b>93</b>
<b>ДОДАТКИ .....</b>	<b>99</b>
Додаток А. Реалізація класу MV_Vector.....	99
Додаток Б. Реалізація класу CompCol_Mat_double .....	104
Додаток В. Реалізація класу CompRow_Mat_double.....	108
Додаток Г. Дані дослідження .....	112

## ВСТУП

Актуальність теми магістерської роботи обумовлена необхідністю ефективного (з високою точністю та мінімальними витратами ресурсів) чисельного вирішення систем лінійних алгебраїчних рівнянь (СЛАР). Чисельне рішення СЛАР – одне з найпоширеніших завдань у науково-технічних дослідженнях. Таке завдання виникає у математичній фізиці (чисельне рішення диференціальних та інтегральних рівнянь), економіці, статистиці. При цьому прикладні завдання часто вимагають вирішення великих і надвеликих СЛАР з числом невідомих більше 10000. До таких СЛАР, наприклад, наводить чисельне рішення двовимірних і особливо тривимірних задач математичної фізики, в яких умови фізичної та геометричної апроксимації двовимірної та тривимірної області диктують використання досить дрібної розрахункової сітки з великою кількістю розрахункових вузлів за лінійним розміром [40].

У зв'язку з цим основна причина розробки та дослідження нових форматів зберігання даних, особливо якщо йдеться про багатовимірні матриці з розрідженою структурою, полягає в тому, що класичні схеми зберігання даних не забезпечують високої продуктивності обчислень для деяких структур і топологій матриць. Нові схеми зберігання можуть бути використані для програм, написаних для різних мов програмування. Особливо слід підкреслити, що хороші результати оптимізації коду можуть бути отримані для сучасних компіляторів, в яких реалізовані ефективні алгоритми оптимізації коду [55].

У разі реалізації алгоритму в класичному стилі, коли кількість ітерацій заздалегідь відома і залежить тільки від порядку матриці, алгоритм буде ефективним лише у роботі з невеликими матрицями  $N \sim 1000$ . Однак у разі застосування алгоритму до матриці близько 100000 і більше алгоритм буде неефективним. Наприклад, на розрахунок такої матриці методом Холецького піде чимало часу, оскільки число арифметичних операцій множення для чисельного розв'язання СЛАР розмірністю  $N$  за допомогою прямого методу

$\sim N^3$ . Якщо дану матрицю зберігати в звичайному двовимірному масиві, то при використанні речовинного типу з подвійною точністю (double) вона займатиме в пам'яті  $100000 \times 100000 \times 8 = 8000000000$  байт – 800 Мб, що є досить ресурсомістким завданням навіть для сучасних ЕОМ. А якщо врахувати, що розмір матриці зростає пропорційно квадрату порядку матриці, то для матриці близько 400000 це буде в 16 разів більше, тобто 12.8 Гб.

В даний час відсутні бібліотеки структур даних широкого призначення для зберігання та обробки матриць великої розмірності, які можуть застосовуватись в алгоритмах для чисельного вирішення великих та надвеликих СЛАР. Таким чином, розробка структур даних, які дозволяють ефективно зберігати та обробляти широкого класу матриці великої розмірності та як наслідок брати участь у реалізації алгоритмів розв'язання, систем лінійних алгебраїчних рівнянь є актуальним завданням.

Дослідженням методів вирішення систем лінійних алгебраїчних рівнянь займалися: Д. Мак-Кракен, С. Годунов, В. Воєводін, А. Островський, Дж. Форсайт, К. Молер, Жегульська.

Об'єктно-орієнтований підхід у реалізації чисельних методів використовували: Р. Беретт, М. Бері, Тоні Ф. Чан, Д. Демл, Д. Донато, Д. Донгарра, Ст Ейджкаут, Р. Позо, Ч. Ромін.

**Об'єкт дослідження** – оптимізація представлення інформації в пам'яті комп'ютера для обчислень лінійної алгебри.

**Предмет дослідження** – розробка об'єктно-орієнтованих бібліотек класів для ефективного зберігання та обробки матриць у пам'яті та розв'язання систем лінійних алгебраїчних рівнянь.

**Метою роботи** є розробка структур даних, у яких реалізовані ефективні методи зберігання та обробки матриць та векторів у пам'яті комп'ютера, а також реалізація алгоритмів розв'язання СЛАР на їх основі.

**Відповідно до мети роботи поставлені такі завдання:**

1. Реалізація структур даних для широкого класу матриць, які дозволяють ефективно зберігати та обробляти матрицю у пам'яті.

2. Реалізація проєкційних методів вирішення СЛАР підпростору Крилова на основі розроблених структур даних.

3. Розробка структури та програмних модулів бібліотеки класів мовою C++ для вирішення СЛАР.

4. Дослідження ефективності у вигляді порівняння часу виконання проєкційних методів рішення з різними передумовами використовуючи реалізовані структури даних для зберігання широкого класу матриць.

**Гіпотеза** – об'єктно-орієнтовані бібліотеки класів можуть ефективно використовуватися у вирішенні систем лінійних алгебраїчних рівнянь.



## РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ ТЕХНОЛОГІЙ ЗБЕРІГАННЯ МАТРИЦЬ

### 1.1. Формулювання критерій подання матриць

У багатьох галузях людської діяльності інформацію часто подають у формі матриць. Матриця - це регулярний числовий масив. Головними характеристиками представлення матриць є щільність чи розрідженість її структури, дані характеристики залежать від значення елементів, що містяться в матриці [45]. Найбільш важливим є поняття розріджених матриць, особливо якщо йдеться про матриці великої розмірності. У спеціальній літературі є кілька визначень розрідженої матриці. Суть їх полягає в тому, що матриця розріджена, якщо в ній розміщено велику кількість нульових елементів. У [33] використовують поняття, пов'язані з граничним переходом: матрицю порядку  $n$  називають розрідженою, якщо число її ненульових елементів є  $O(n)$ . Це означає, що кількість елементів, що відрізняються від нуля, при досить великих  $n$  пропорційно  $n$ . Однак для заданої матриці  $n$  не є достатньо великим, а цілком конкретним числом. Відповідно до [26] критерієм розрідженості пропонується вважати обмеженість числа ненульових елементів у рядку, у типовому випадку від 2 до 10. Інше визначення полягає в тому, що матриця порядку  $n$  вважається розрідженою, якщо число її ненульових елементів виражається як  $n^{1+\nu}$ , де  $\nu < 1$ . Типові значення  $\nu$ : 0.2 для електричних задач; 0.5 для стрічкових матриць, асоційованих із сітками. Матриця порядку 1000 при  $\nu = 0,9$  має 501187 ненульових елементів, в цьому випадку виникає питання, чи варто вважати таку матрицю розрідженою.

Більш практичний підхід до визначення розрідженої матриці спирається на взаємозв'язок трьох основних інгредієнтів: даної матриці, даного алгоритму та даної обчислювальної машини. Це поняття за своєю природою є евристичним: матриця розріджена, якщо має сенс отримувати вигоду з наявності в ній багатьох нулів. Будь-яку розріджену матрицю можна

обробляти так, ніби вона була щільною: навпаки, будь-яку щільну, або заповнену, матрицю можна обробляти алгоритмами для розріджених матриць. У обох випадках вийдуть правильні чисельні результати, але обчислювальні витрати зростуть [45]. Приписування матриці якості розрідженості еквівалентно твердженню про існування алгоритму, що використовує її розрідженість і робить обчислення з нею дешевше в порівнянні зі стандартними алгоритмами. Розріджену матрицю можна розглядати як числовий масив, у загальному випадку не регулярний. Однак найчастіше асоціюють з цими числами позиції в значно більшому регулярному масиві, оскільки зручніше аналізувати в термінах регулярних масивів і виходити з них при конструюванні алгоритмів [64]. Розглянемо наступну систему восьми лінійних рівнянь із вісьмома невідомими:

$$x_3 = 3$$

$$x_1 + 2x_6 = 16$$

$$x_1 + x_2 = 3$$

$$x_2 - x_8 = -6$$

$$x_4 - 2x_5 + x_7 = 1$$

$$-x_3 + x_6 = 3$$

$$-x_5 + x_7 = 2$$

$$-2x_4 + x_8 = 0$$

Цій системі можна поставити у відповідність таку матрицю коефіцієнтів:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & -2 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & -2 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Ця система має лише 16 коефіцієнтів, а використовується 64 позиції. У цьому випадку не є ефективним заповнювати несуттєві позиції нулями, а потім намагатися отримати вигоду з наявності настільки великої кількості нулів. У загальному випадку, розріджену матрицю слід представляти не як матрицю, а скоріше як граф, де кожна пара рівняння – невідоме асоціюється з вершиною, а кожен коефіцієнт – з ребром. Тому теорія графів відіграє важливу роль у технології розріджених матриць. Розріджена матриця, будучи безліччю чисел, що не мають регулярності, не може бути представлена в пам'яті машини тим самим простим способом, що і повна матриця. Якщо зберігаються чисельні значення коефіцієнтів системи рівнянь, необхідно разом із значенням кожного коефіцієнта зберігати номер відповідного рівняння та номер відповідного невідомого. Тобто потрібно зберігати значення ненульових елементів плюс індексна інформація, що вказує на розташування кожного елемента в регулярному масиві. Ця додаткова інформація становить накладні витрати – ціну, яку доводиться сплачувати за відмову від зберігання нулів [45].

Матричні алгоритми повинні проектуватися таким чином, щоб оброблялися тільки ненульові елементи і щоб на підставі попереднього знання розташування ненульових елементів уникалися операції типу складання з нулем або множення на нуль. Таким чином, число операцій, що робляться машиною при виконанні алгоритму, пропорційно числу ненульових елементів, а не числу всіх елементів матриці. Слід зазначити, що було б неправильно зберігати всі елементи, включаючи нулі, а потім оминати операції з нулями за допомогою оператора if. Цей оператор без необхідності виконуватиметься  $n^2$  раз чи більше, перетворюючи алгоритм на квадратичний по порядку  $n$  матриці. Хороший алгоритм для розріджених матриць використовує наявні в нього відомості про позиції ненульових елементів, щоб робити необхідні операції. Якщо для розрідженої матриці

число ненульових елементів у рядку постійно, то в багатьох випадках порядок оптимального алгоритму може зменшуватися до  $n$  [54].

Розріджена матриця, що представляє задану інформацію, має певну задану розрідженість. Проте якщо розріджена матриця породжується якимось алгоритмом як проміжний результат у межах ширшого рахунку, виникає питання: чи немає способу збільшити її розрідженість шляхом генерування меншого числа ненульових елементів? Найважливіший випадок, коли відповідь ствердна, — це гауссовий виняток. У гауссовому винятку розрідженість матриці зменшується порівняно з вихідною, оскільки виникають нові ненульові елементи. Результируюча матриця менш розріджена, ніж вихідна, але ступінь її заповнення залежить від порядку, в якому вибираються основні елементи. Хороший алгоритм для розріджених матриць намагається зберегти розрідженість, наскільки можна зменшуючи заповнення [45].

Таким чином, можна сформулювати три основні критерії, які спрямовували розвиток більшої частини технології розріджених матриць: зберігати лише ненульові елементи, оперувати лише з ненульовими елементами та зберігати розрідженість. Не всі алгоритми для розріджених матриць досягають цих цілей, лише найбільш витончені. Багато схем зберігання допускають певну частку нулів, і алгоритм обробляє їх, якби вони були нулями. Алгоритм, що зберігає і обробляє меншу кількість нулів, складніший у реалізації і доцільний лише досить великих матриць. Існує повний спектр алгоритмів, розрахованих на різні типи матриць, від щільних до дуже розріджених, з різними необхідними для практики ступенями ефективності, складності чи простоти.

## **1.2. Зберігання масивів, списків, стеків та черг**

Технологія представлення матриць часто вимагає зберігання та обробки списків, елементами яких можуть бути числа, цілі, речові або комплексні або об'єкти більш складної структури, такі, як матриця, масив або вершина графа разом з відповідними ребрами. Основними операціями, які

зазвичай виконуються над списками є: додавання елемента в кінець списку, видалення елемента з кінця списку, вставка або видалення елемента в середині або на початку списку, визначення позиції деякого елемента або елемента, наступного за даними, сортування, впорядкування. Вибір схеми зберігання залежить від операцій, які передбачається проводити, оскільки ефективність виконання цієї операції не регулярна щодо різних схем зберігання [45].

Найпростіша структура даних – це масив. Його особливістю є те, що крім зберігання чисел він може містити покажчики на елементи складнішої природи, що зберігаються насправді десь в іншому місці. Всі елементи масиву доступні безпосередньо за час, що не залежить від його розміру. Але з накладними витратами пов'язане використання подвійних чи кратних індексів, оскільки машинна пам'ять одномірна. У машинах з віртуальною пам'яттю масив може знаходитися на периферійному пристрої, що запам'ятовує, і це призводить до великих тимчасових витрат для доступу по індексу [55].

Лінійний зв'язковий список - це послідовність осередків, пов'язаних у певному порядку. Кожна комірка містить елемент списку та покажчик, що повідомляє положення наступної комірки. Припустимо, що необхідно зберігати числа  $a, b, c, d$  у вказаному порядку в масиві  $A(I)$ . Схема зберігання може бути представлена наступним чином (символом  $x$  відзначені несуттєві значення):

позиція =	1	2	3	4	5	6	7	8
$A(I) =$	$x$	$b$	$x$	$b$	$a$	$x$	$c$	$x$
$next(I) =$	$x$	7	$x$	0	2	$x$	4	$x$
$IP =$	5							
ознака кінця =	0							

Рис. 1.1. Схема зберігання зв'язкового списку

У масиві  $A(I)$  зберігаються елементи списку, а  $next(I)$  — покажчики позицій наступних елементів.  $IP$  є вказівником входу, що показує

розташування першого елемента. У цьому випадку він дорівнює 5. У позиції 5 знаходимо перший елемент  $A(5) = a$ , а  $next(5) = 2$  повідомляє, що наступний елемент потрібно шукати в позиції 2. У такий спосіб можна переглянути весь список. В останню клітинку повинно бути поміщено - ознака кінця, що вказує закінчення списку; в даному прикладі такою ознакою служить 0. Ще один спосіб полягає в тому, щоб зберігати загальну кількість елементів списку та за допомогою цього числа визначати, коли список вичерпаний. Порожній список зручно задавати, присвоюючи покажчику входу значення, яке може адресувати будь-яку позицію масивів, наприклад неперитивне число [58].

Досить легко вставляти або видаляти елементи в будь-якому місці. Якщо між  $b$  і  $c$  необхідно вставити число  $e$ , і відомо, що комірка 3 порожня, а  $b$  знаходиться в позиції 2. Наступна процедура виконує необхідну операцію:

$$\begin{aligned} A(3) &\leftarrow e \\ next(3) &\leftarrow next(2) \\ next(2) &\leftarrow 3 \end{aligned}$$

Процедура видалення елемента значно простіша:

$$next(2) \leftarrow next(7)$$

Для виконання цієї процедури необхідно знати, що елемент, що передус, розташований в позиції 2, таким чином, відбувається видалення елемента, наступного за  $b$ , а не самого  $c$ . Якщо потрібно вставити або видалити елемент на початку списку, слід перевизначити покажчик входу. Вставлення або видалення елементів не змінює порядок, у якому зберігаються інші елементи [55].

Якщо список зберігається у масиві, важливо мати інформацію про вільні позиції останнього. Їх також пов'язують у список, що потребує ще одного покажчика входу. Обидва списки не перетинаються, тому можуть бути збережені в тих самих масивах. Нижче наведена структура даних вийде, якщо зв'язати вільні позиції. (IE – покажчик входу для нового списку):

позиція = 1   2   3   4   5   6   7   8

$A(I) =$	$x$	$b$	$x$	$d$	$a$	$x$	$c$	$x$
$next(I) =$	$3$	$7$	$6$	$0$	$2$	$8$	$4$	$0$
$IP = 5$								
$IE = 1$								
$ознака\ кінця = 0$								

Рис. 1.2. Зв'язковий список з інформацією про вільні позиції

Зв'язковий список стає кільцевим зв'язковим списком, якщо в його останню позицію замість ознаки кінця помістити покажчик на початкову позицію. Кільцевий список не має початку і кінця, він вимагає окремого покажчика входу, що зберігається окремо, який може вказувати на будь-яку зайняту позицію. У кільцевому списку елементи можна видаляти чи додавати, не змінюючи порядку інших, а вільні позиції пов'язувати так само, як у лінійному списку [45].

Двонаправлений зв'язковий список, лінійний або кільцевий, можна отримати у разі, якщо ввести масив, що містить для кожної комірки адресу попередньої комірки. Двонаправлений список можна переглядати в обох напрямках; одним з головних його переваг є те, що можна вставляти або видаляти елементи, не знаючи позиції попереднього елемента. Лінійний двонаправлений список вимагає двох покажчиків входу – покажчик початку списку та покажчик кінця списку. Для кільцевого двонаправленого списку достатньо одного покажчика входу. Є можливість пов'язати між собою вільні позиції двонаправленого списку.

Стек – це список зі спрощеним способом зберігання та обробки [55]. У стеку елементи зберігаються у послідовних позиціях, чим усувається потреба у зв'язках. Для визначення позиції останнього елемента використовується спеціальний покажчик. Застосування стека зумовлене ситуаціями, коли елементи потрібно додавати або видаляти тільки через верх стека. Щоб увімкнути або опустити новий елемент у стек, необхідно збільшити значення покажчика на одиницю, перевіривши вільну кількість пам'яті, а потім

виконується запис елемента у позицію, що повідомляється покажчиком. Щоб виключити або підняти останній елемент з верху стека, слід зменшити на одиницю значення вказівника. Порожній стек розпізнається за нульовим значенням покажчика. Стеки використовують у кількох алгоритмах для розріджених матриць. Прикладами можуть бути алгоритм Джорджа для деревоподібного розбиття, що обговорюється в [57], і алгоритм Тар'яна для блокової тріангуляризації матриці, що розглядається в [57].

Черга - це список елементів, що зберігаються в послідовних позиціях, причому включення елементів проводиться через початок черги, а виняток - через кінець [55]. Для зазначення позицій початку та кінця черги використовуються два покажчики. Порожня черга пізнається завдяки тому, що для неї значення покажчика кінця на одиницю більше значення покажчика початку. Для черги з єдиним елементом значення обох покажчиків збігаються. Якщо ділянка пам'яті, відведена для зберігання черги, вичерпана, нові елементи можна записувати на початок цієї ділянки, закріплюючи тим самим чергу; при цьому початок черги не повинен перетинатися з її кінцем.

Усі вище описані схеми зберігання спираються на масив як єдину структуру даних, підтримувану мовою програмування C++. Властивості зв'язкових списків можна ефективно реалізувати, використовуючи таку структуру даних, як запис. У цьому випадку не потрібно непрямої адресації, і пам'ять може динамічно розподілятися ціною деяких системних витрат пам'яті [41].

### **1.3. Зберігання цілих списків**

Цілі списки є особливо важливими технологіями розріджених матриць [43]. Елементи списку належать множині  $(1, 2, \dots, n)$ ; деякі з них можуть повторюватись. Нехай  $m$  - кількість елементів списку;  $n$  називається розмахом списку. Список називають розрідженим, якщо  $m \ll n$ . Цілий список можна зберігати у компактній формі, користуючись масивом довжини, не меншою за  $m$ ; так, для списку 3, 11, 7, 4:



$$\begin{array}{l} \text{позиція} = 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \\ \text{JA} = 3 \quad 11 \quad 7 \quad 4 \\ \text{M} = 4 \end{array}$$

Рис. 1.3 Зберігання списку у компактній формі

Крім масиву JA необхідно зберігати (за допомогою змінної M) число  $m$  елементів списку; у вище поданому випадку  $m = 4$ . Порожній список упізнається за нульовим значенням M. Щоб включити до списку новий елемент, достатньо збільшити M на одиницю і потім записати новий елемент у позицію з номером M. Щоб виключити елемент із позиції  $i$ , необхідно перенести в цю позицію останній елемент і зменшити M на одиницю. Якщо потрібно зберігати впорядкованість цілих чисел, то включення або виключення елемента в якійсь позиції викликає зсув всіх елементів, що знаходяться праворуч від неї.

Цілі списки з повтореннями або без них можуть зберігатися як зв'язні лінійні або кільцеві списки за допомогою масиву JA довжини не менше  $m$  і додаткового масиву NEXT; Цей тип зберігання також компактний, тому підходить для розріджених списків.

Для цілих списків із різними елементами існує альтернатива. Такий список може зберігатися одним масивом довжини  $n$  або більшої у формі зв'язного списку, лінійного або кільцевого. Якщо у наведеному вище прикладі використовувати кільцевий зв'язковий список, то отримаємо наступний опис матриці, представлений (рис. 1.4).

$$\begin{array}{l} \text{позиція} = 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \\ \text{JA} = x \quad x \quad 11 \quad 3 \quad x \quad x \quad 4 \quad x \quad x \quad x \quad 7 \quad x \\ \text{IP} = 3 \end{array}$$

Рис. 1.4. Кільцевий зв'язковий список

Число, яке зберігається в кожній позиції JA, дає одночасно значення елемента списку та адресу наступного елемента; тому додатковий масив next стає зайвим. Для входу в ланцюг необхідний вказівник IP, що задає позицію першого числа у списку. Цей тип зберігання називають розширеним, на

противагу компактному, тому що для нього потрібний масив довжини, принаймні  $n$ . У будь-якому місці списку нескладно увімкнути або виключити елемент, не змінюючи порядку, де зберігаються інші елементи. Наприклад, якщо  $i, k$  - два послідовні елементи списку, що знаходяться в масиві  $JA$ , так що  $JA(i) = k$  і тепер потрібно вставити  $j$  між  $i$  і  $k$ . Тоді вважаємо наступне:

$$JA(j) \leftarrow JA(i)$$

$$JA(i) \leftarrow j$$

Якщо, навпаки, необхідно виключити  $k$ , то

$$JA(j) \leftarrow JA(k)$$

Якщо необхідно увімкнути елемент перед першим елементом або видалити перший елемент, слід перевизначити покажчик входу. Список одного елемента, припустимо 5, при використанні розширеної схеми виглядає наступним чином:

$$\begin{array}{cccccccccc} \text{позиція} & = & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ JA & = & x & x & x & x & 5 & x & x & x & x \\ IP & = & 5 \end{array}$$

Рис. 1.5. Розширена схема зберігання

Порожній список можна впізнати за нульовим чи негативним значенням покажчика входу.

Одне з головних застосувань розширеної схеми - це зберігання декількох списків, що не перетинаються, тобто списків, що не мають загальних елементів [54]. У більшості випадків всі списки мають однаковий розмах, але в загальному випадку вважатимемо  $n$  максимальним з розмахів окремих списків. Усі списки можна зберігати в тому самому масиві, довжини  $n$  або більшої. Також потрібен масив покажчиків входу. Проілюструємо описуваний спосіб зберігання прикладом. Розглянемо три цілих списки, що не перетинаються:

1. Список: 2, 8, 6, 3;

2. Список: 5, 4, 9;

3. Список: 7, 1, 10;

Дані списки можуть бути розміщені як кільцеві зв'язкові списки в одному масиві довжини 10:

позиція	=	1	2	3	4	5	6	7	8	9	10
JA	=	10	8	2	9	4	3	1	6	5	7
IP	=	2	5	7							

Рис. 1.6. Кільцеві зв'язкові списки в одному масиві

При використанні цього типу зберігання легко виконуються такі операції, як розбиття списку на два або конкатенація двох списків з утворенням нового [55].

#### 1.4. Подання та зберігання графів

Графи є важливою частиною технології розріджених матриць, оскільки існує відповідність між матрицями і графами, що зберігає деякі властивості. На рис. 1.7 (а) представлений граф, що складається з п'яти вершин і семи ребер. Точніше, граф  $G = (U, E)$  складається з безлічі вершин  $U$  та безлічі ребер  $E$ ; ребро можна ототожнити з парою  $(u, v)$  різних вершин із  $U$ . Наприклад, на рис. 1.7 (а) 3 є вершина, а  $(3, 5)$  - ребро, що зображується відрізком, що з'єднує вершини 3 і 5.

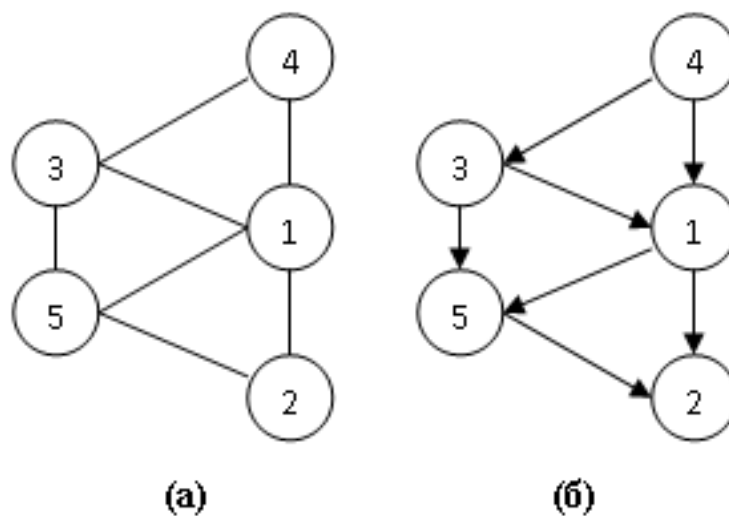


Рис. 1.7. Неорієнтований граф (а) та орієнтований граф (б)

Якщо пари  $(u, v)$  і  $(v, u)$  не відрізняються, то кажуть, що ребра отожднюються з неупорядкованими парами, а граф називають неорієнтованим. Якщо пари, що представляють ребра, упорядковані, то граф називають орієнтованим графом. Якщо в орієнтованому графі  $(u, v) \in E$ , то кажуть, що  $v$  суміжна з  $u$ . Якщо  $(u, v)$  — ребро неорієнтованого графа, то кажуть, що  $v$  суміжна з  $u$  і  $u$  суміжна з  $v$ . Неорієнтований граф можна розглядати як орієнтований, у якому, якщо  $(u, v)$  — ребро, то  $(v, u)$  також ребро. Щоб зобразити граф з вершинами  $n$ , зручно помітити його, встановивши відповідність між вершинами і цілими числами  $1, 2, n$ . На рис. 1.7 показані приклади орієнтованого та неорієнтованого графа.

У пам'яті машини граф можна представити, зберігаючи для кожної вершини список вершин, суміжних із нею. Така множина списків називається структурою суміжності графа [40]. Якщо списки зберігаються у компактній формі (масив `list`), то необхідний ще масив покажчиків (`IP`) початку кожного списку. Для орієнтованого графа (рис. 1.7 б) структура суміжності може бути представлена наступним чином:

позиція =	1	2	3	4	5	6	7	8
list =	2	5	1	5	1	3	2	
IP =	1	3	3	5	7	8		

Рис. 1.8. Структура суміжності орієнтованого графа

Список суміжності, наприклад, для вершини 4 починається з 5 позиції масиву `list`, що впливає зі значення 5 покажчика  $IP(4)$ , і закінчується в 6 позиції, це пов'язано з тим, що наступний список, тобто список для вершини 5, починається з позиції  $IP(5) = 7$ . Таким чином, з вершиною 4 суміжні вершини 1 і 3. Список для вершини 2 порожній, оскільки значення покажчиків  $IP(2)$  та  $IP(3)$  збігаються. У кінець масиву `IP` включено додатковий покажчик; він є першою вільною позицією масиву `list`. Цей прийом є зручним у програмній реалізації. Неорієнтований граф (рис. 1.7 а) представляє наступна структура суміжності:

позиція =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-----------	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

```
list = 2  3  4  5  1  5  1  4  5  1  3  1  2  3
IP = 1  5  7  10  12  15
```

Рис. 1.9. Структура суміжності неорієнтованого графа

Кожне ребро представлене двічі. Компактне зберігання списків суміжності ідентично малої схеми зберігання портрета розрідженої матриці, асоційованої з цим графом [26].

При використанні цього типу зберігання може бути зручним зв'язати обидві копії ребра за допомогою додаткового масиву. Так, якщо те саме ребро зберігається в позиціях  $i$  і  $j$  масиву `list`, то  $\text{link}(i) = j$  і  $\text{link}(j) = i$ .

Недолік компактного зберігання структури суміжності в тому, що важко додавати та виключати ребра. Ці труднощі долаються, якщо зберігати структуру суміжності як безліч зв'язкових списків, лінійних чи кільцевих, разом із масивом `IP` покажчиків входу кожен список [43]. Серед списків зазвичай існують такі що перетинаються, тому неможливе застосування техніки зберігання, яка описувалася в § 1.3, і потрібен додатковий масив `next`. Використовуючи кільцеві списки, орієнтований граф (рис. 1.7, б) можна подати так (несуттєві позиції позначені символом  $x$ ):

```
позиція = 1  2  3  4  5  6  7  8  9
list = 3  5  1  2  x  5  1  x  2
next = 3  7  1  6  x  4  2  x  9
IP = 4  0  7  3  9
```

Рис. 1.10. Орієнтований граф у кільцевому списку

Зв'язне подання неорієнтованого графа (рис. 1.7, а) може бути подане таким чином:

```
позиція = 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
list = 5  x  5  1  4  x  3  5  2  2  1  1  4  3  1  3  x
next = 4  x  13  1  14  x  11  5  12  8  7  16  15  10  3  9  x
IP = 8  4  13  11  9
```

Рис. 1.11. Неорієнтований граф у кільцевому списку

Кожне ребро зберігається двічі. Наприклад, ребро (1,5) зберігається в позиціях 8 і 12 спочатку як (1, 5), потім як (5, 1). Як зазначено вище, для зв'язування обох копій кожного ребра можна використовувати масив (link). Якщо з неорієнтованого графа необхідно зробити видалення ребра, то у цьому випадку можна скористатися двонаправленим списком. У всіх випадках вільні позиції можуть бути пов'язані між собою з метою подальшого використання.

Ще одна поширена схема зберігання - це таблиця зв'язків [45]. Якщо граф має  $n$  вершин і  $m$  - максимальна ступінь вершини (тобто число суміжних з нею вершин), то таблиця зв'язків є масивом з  $n$  рядків і  $m$  стовпців; при цьому в 1-му рядку зберігається список суміжності вершини 1. Для графа на рис. 1.7 (б) таблиця зв'язків складається з 5 рядків та двох стовпців:

$$\begin{bmatrix} 2 & 5 \\ 0 & 0 \\ 1 & 5 \\ 1 & 3 \\ 2 & 0 \end{bmatrix}$$

Для представлення графа можна використовувати матрицю суміжності [26]. Для графа з  $n$  вершинами матриця суміжності - це квадратна матриця  $A$  порядку  $n$ , яка визначається таким чином:  $a_{ij} = 1$  тоді і лише тоді, коли  $(i, j)$  - ребро даного графа; в іншому випадку  $a_{ij} = 0$ . Матриця суміжності для графа на рис. 1.7 (б) має вигляд

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Матриця суміжності неорієнтованого графа симетрична. Цей тип зберігання не є неекономічним, якщо матриця повністю записується в

пам'ять машини. Але він може бути зручний для алгоритмів, яким часто потрібна інформація про наявність чи відсутність будь-яких ребер. Матрицю суміжності можна зберігати і як розріджену матрицю, наприклад, рядками, не запам'ятовуючи числові значення її елементів. І тут вийде компактна схема зберігання структури суміжності.

Кліка - це граф, у якому кожна пара вершин з'єднана ребром. Для завдання кліки достатньо зберігати список посилань на її вершини, і немає потреби запам'ятовувати будь-яку інформацію про її ребра. Ця властивість знайшла важливе застосування у гауссовому виключенні [29].

### 1.5. Схеми зберігання матриць великої розмірності

З стрічковими матрицями пов'язана найпростіша стратегія використання нульових елементів матриці. Матрицю  $A$  називають стрірковою, якщо її ненульові елементи укладені всередині стрічки, утвореної діагоналями, паралельними головній діагоналі. Таким чином,  $a_{ij} = 0$  якщо  $|i - j| > \beta$  та  $a_{k,k-\beta} \neq 0$  або  $a_{k,k+\beta} \neq 0$  хоч би для одного значення  $k$ . Тут  $\beta$  – півширина, а  $2\beta + 1$  – ширина стрічки. Стріркою матриці  $A$  називається безліч елементів, для яких  $|i - j| \leq \beta$ . Іншими словами, для будь-якого рядка  $i$  стрічці належать всі елементи зі стовпцевими індексами від  $i - \beta$  до  $i + \beta$ , тобто  $2\beta + 1$  елементів. Це число може бути набагато менше порядку матриці [64].

Якщо матриця симетрична, достатньо зберігати її напівстрічку. Верхня напівстрічка складається з елементів, що знаходяться у верхній частині стрічки, тобто таких, що  $0 < j - i \leq \beta$ ; нижня напівстрічка складається з елементів нижньої частини стрічки, тобто таких, що  $0 < i - j \leq \beta$ ; в обох випадках у рядку  $\beta$  елементів.

$$A = \begin{bmatrix} 1 & & & & & & \\ & 2 & 8 & 9 & & & \\ & 8 & 5 & & & & \\ & 9 & & 4 & 10 & & \\ & & & 10 & 5 & 11 & 12 \\ & & & & 11 & 6 & \\ & & & & 12 & & 7 \end{bmatrix} \quad AN(I,J) = \begin{bmatrix} & & & & & & 1 \\ & & & & & 0 & 2 \\ & & & & 0 & 8 & 3 \\ & & & 9 & 0 & 4 & \\ & & 0 & 10 & 5 & & \\ & 0 & 11 & 6 & & & \\ 12 & 0 & 7 & & & & \end{bmatrix}$$

(а) (б)

Рис. 1.12. Симетрична стрічкова матриця 7x7 з шириною стрічки 5 (а); діагональна схема зберігання в прямокутному масиві AN(I,J) (б)

Діагональна схема зберігання симетричної стрічкової матриці за допомогою масиву  $AN(i, j)$  представлена на рис. 1.12. Для матриці порядку  $n$  та півширини стрічки  $\beta$  масив має розміри  $n \times (\beta + 1)$ . Головна діагональ зберігається в останньому стовпці, а нижні кодіагоналі - в інших стовпцях зі зсувом на одну позицію вниз при кожному зміщенні вліво. У цьому прикладі  $n = 7$ ,  $\beta = 2$ . Для масиву необхідно 21 осередок пам'яті. Для несиметричної матриці  $A$  необхідний масив розміром  $n \times (2\beta + 1)$ ; нижня напівстрічка і головна діагональ зберігаються як і раніше, а верхні кодіагоналі – у правій частині масиву зі зсувом однією позицією вгору при кожному зміщенні вправо. Діагональна схема зручна, якщо  $\beta \ll n$ . Вона є схемою прямого доступу в тому сенсі, що є проста взаємно однозначна відповідність між положенням елемента в матриці  $A$  та її положенням у масиві:  $a_{ij}$  зберігається компонентом  $AN(i, j - i + \beta + 1)$ .

У стрічкових матрицях ширина стрічки залежить від порядку, в якому розташовані рядки та стовпці. Тому можна шукати перестановки рядків та стовпців, що призводять до зменшення ширини стрічки. Мінімальна ширина стрічки означає зниження запитів до пам'яті; у обчисленнях, пов'язаних із матрицями, вона зазвичай зменшує і роботу. У разі симетричної матриці



цінна властивість симетрії буде збережена, якщо для стовпців і рядків використовуються однакові перестановки [45].

Якщо система лінійних рівнянь має стрічкову матрицю коефіцієнтів і вирішується за допомогою гауссового виключення, причому головні елементи вибираються на головній діагоналі, вся арифметика обмежена стрічкою, поза якою ніяких нових ненульових елементів не виникає. Гауссовий виняток можна проводити на місці, оскільки для будь-якого ненульового елемента, якщо він з'явиться, вже зарезервована позиція у схемі зберігання.

Власні значення і власні вектори стрічкової матриці, а також власні значення і власні вектори узагальненої спектральної задачі з двома стрічковими матрицями однакової ширини, можна обчислити, не використовуючи додаткову пам'ять [21].

Стрічкова матриця високого порядку може мати широку стрічку з великою кількістю нулів. Для такої матриці діагональна схема не є неекономним рішенням. У цьому випадку оптимальним є застосування схеми, яка називається профільною схемою, або схемою змінної стрічки. Для кожного рядка  $i$  симетричної матриці  $A$  буде

$$\beta_i = i - j_{\min}(i)$$

де  $j_{\min}(i)$  - мінімальний стовпцевий індекс рядка  $i$ , для якого  $a_{ij} \neq 0$ .

Таким чином, перший ненульовий елемент рядка  $i$  знаходиться на позиції ліворуч від головної діагоналі. Певна півширина стрічки  $\beta \in \max(\beta_i)$ .

Оболонка матриці  $A$  - це безліч елементів  $a_{ij}$  для яких  $0 < i - j \leq \beta_i$ . У рядку  $i$  оболонці належать всі елементи зі стовпцевими індексами від  $j_{\min}(i)$  до  $i - 1$ , всього  $\beta_i$  елементів. Діагональні елементи не входять до оболонки. Профіль матриці  $A$  визначається як кількість елементів в оболонці:

$$\text{profile}(A) = \sum_i \beta_i$$

При використанні профільної схеми всі елементи оболонки, впорядковані рядками, зберігаються, включаючи нулі, в одновимірному масиві (AN) [54]. Діагональний елемент даного рядка поміщається у його кінець. Довжина масиву AN дорівнює сумі профілю та порядку A. Необхідний ще масив показчиків (IA); елементи цього масиву вказують на розташування діагональних елементів AN. Так, при  $i > 1$  елементи рядка  $i$  знаходяться в позиціях від  $IA(i-1)+1$  до  $IA(i)$ . Єдиний елемент  $a_{11}$  1-го рядка зберігається у  $AN(1)$ . Елементи мають послідовні стовпцеві індекси, що легко обчислюються. Наприклад, для матриці на рис. 1.12 (а) профіль дорівнює 7, а профільна схема виглядає так:

позиція=	1	2	3	4	5	6	7	8	9	10	11	12	13	14
AN =	1	2	8	3	9	0	4	10	5	11	6	12	0	7
IA =	1	2	4	7	9	11	14							

Рис. 1.13. Профільна схема зберігання

Можливий варіант профільної схеми із зберіганням оболонки по стовпчикам. І тут стовпці матриці зберігають свої довжини, цю схему часто називають вертикальною. Ще одне з понять, яке використовується при конструюванні схем зберігання симетричних матриць можна виділити при розгляді рядка  $i$  матриці A. Кажуть, що стовпець  $j, j > i$  активний у цьому рядку, якщо він містить ненульовий елемент у рядку  $i$  або вище за неї. Нехай  $\omega_i$ , - Число стовпців, активних у рядку  $i$ . Тоді,  $\max \omega_i$  - називають хвильовим фронтом або шириною фронту A. У прикладі (рис. 1.12, а) стовпець 4 активний у рядку 3, а ширина фронту A дорівнює 2.

Як і у разі стрічкових матриць, профіль зазвичай змінюється при перестановках рядків та стовпців. Менший профіль означає меншу пам'ять і менше операцій у обчисленнях, виконуваних з матрицею, тому алгоритми мінімізації профілю відіграють важливу роль технології розріджених матриць [45].

Розглядаючи елементи, укладені в оболонці, можна виявити, що у гауссовому винятку значення  $\beta_i$  кожного рядка не зміниться, якщо головні елементи вибрати на головній діагоналі. Нові ненульові елементи не можуть виникнути у позиціях, зовнішніх для оболонки. Вони можуть з'явитися всередині оболонки, де їм вже виділено місце; звідси випливає, що виняток можна проводити за статичної схеми зберігання. Профільну схему можна з вигодою використовувати і в інших випадках. Наприклад, вона добре пристосована для ітераційних алгоритмів, де потрібна ефективність при обчисленні добутків матриці на вектори. До цієї категорії належать алгоритми Ланцоша та сполучених градієнтів. У цьому контексті поняття оболонки можна поширити на несиметричні матриці [34].

Схема змінної стрічки рядково орієнтована, будь-який рядок матриці в цьому випадку сканується ефективно; в той же час, сканування стовпців буде неефективним. Крім того, схема є статичною, тому що включення нового елемента, що лежить поза оболонкою, вимагає зміни всієї структури (якщо тільки не використовуються записи змінної довжини) [36].

Вище описані схеми зберігання матриць дають високу ефективність у багатьох важливих практичних додатках. Розріджена матриця  $A$ , яку можна переупорядкувати так, щоб вона мала вузьку стрічку або малу оболонку, вимагає набагато меншої пам'яті, ніж при зберіганні у двовимірному масиві, як це реалізовано для щільних матриць.

Нулі, що знаходяться всередині стрічки або оболонки, зберігатимуться і беруть участь в операціях (або принаймні у перевірках на нульове значення), простота, властива цим схемам і пов'язаному з ними програмуванню, може значно переважити цю незручність. Порівняння двох методів зберігання показує, що при використанні профільної схеми зберігається менше нулів; ціна, яку доводиться платити за це зменшення, — необхідність мати додатковий масив, що містить допоміжну інформацію, і внаслідок цього збільшується складність програми.

Труднощі реалізації і накладна пам'ять зростають паралельно з ускладненням схеми зберігання. Високо складні схеми вимагають професійної програмної реалізації, інакше їх потенційні переваги будуть втрачені [45]. З цього погляду проста схема може бути більш підходящою для завдання малого або середнього розміру. З іншого боку, можливість застосування високо складної схеми може визначати, чи можна вирішити взагалі велике завдання на даній машині.

Нижче представлені схеми, в яких нулі не зберігаються або зберігаються тільки деякі нулі. Розглянемо розріджену матрицю  $A$ ; без втрати спільності припустимо, що  $A$  є матрицею великої розмірності. Її ненульові елементи, які представлені в малій кількості порівняно з числом нулів, розпорошені по всій матриці, утворюючи портрет матриці або шаблон нулів – не нулів.

$$A = \begin{bmatrix} & 6 & \\ 9 & 4 & 7 \\ 5 & & \\ & 2 & 8 \end{bmatrix}$$

Однією зі схем, що дозволяє зберігати не нульові елементи в компактній формі і в довільному порядку в одновимірному масиві ( $AN$ ) є схема Д. Кнута. Інформація про положення ненульових елементів може зберігатися двома додатковими паралельними одновимірними масивами ( $I$  та  $J$ ); В даному випадку для кожного ненульового елемента містяться його рядковий та стовпцевий індекси. Так, для кожного  $a_{ij} \neq 0$  у пам'яті знаходиться  $a_{ij}, i, j$ . Далі, для того щоб проводити швидкий пошук елементів довільного рядка або стовпця матриці, необхідна пара покажчиків для кожної  $a_{ij}, i, j$ , а також покажчики входу для рядків і стовпців, що повідомляють початок кожного рядкового або стовпцевого списку. Нехай  $NR$  - масив, що зберігає малі покажчики, а  $NC$  - масив стовпцевих покажчиків.

Масиви AN, I, J, NR, NC мають однакову довжину, та їх однойменні позиції відповідають одна одній. Нехай JR і JC – масиви містять покажчики входу для рядків та стовпців розташовані відповідно до порядку рядків та стовпців матриці. Нижче представлена матриця A, що зберігається згідно зі схемою Кнута.

	1	2	3	4	5	6	7
AN =	6	9	4	7	5	2	8
I =	1	2	2	2	3	4	4
J =	2	1	2	4	1	2	4
NR =	0	3	4	0	0	7	0
NC =	3	5	6	7	0	0	0
JR =	1	2	5	6			
JC =	2	1	0	4			

Рис. 1.14. Схема зберігання Д. Кнута

Якщо необхідно, наприклад, елементи 2-го стовпця, то, визначивши, що  $JC(2) = 1$ , потрібно почати з 1 позиції масивів AN, I і NC. Так як  $AN(1) = 6$ , і  $I(1) = 1$ , то елемент 6. знаходиться у 2-му стовпці, 1-му рядку. Оскільки  $NC(1) = 3$ , переходимо в позицію 3 і знаходимо елемент 4 у 2-му рядку. Визначивши, що  $NC(3) = 6$ , переходимо у позицію 6 і знаходимо елемент 2. у 4-му рядку. Більше елементів у 2-му стовпці немає, тому що  $NC(6) = 0$ . Також слід зазначити, що  $JC(3) = 0$ ; це означає, що третій стовпець порожній. При користуванні даною схемою елемент  $a_{ij}$  можна знайти, тільки входячи до списку  $i$  рядка і переглядаючи його до тих пір, поки не буде знайдено стовпцевий індекс  $j$  [45].

Схема Кнута вимагає п'яти клітинок пам'яті для кожного ненульового елемента A та наявності покажчиків входу для рядків та стовпців. Внаслідок великої накладної пам'яті така схема неекономна. Переваги її в тому, що будь-де можна включити або виключити елемент, і можна ефективно сканувати рядки і стовпці. Для цього використовується описана в § 1.2

техніка обробки зв'язкових списків. Записи, що мають фіксовану довжину, можуть бути розміщені в довільних сферах фізичної пам'яті і потім пов'язані між собою; вільні записи також зв'язуються для подальшого використання. Схема пристосована для випадків, коли матриця  $A$  будується алгоритмом, у якому не можна передбачити кінцеве число та позиції ненульових елементів. Найважливішим прикладом такого алгоритму є гауссовий виняток для матриць загального виду [30].

Існує модифікація схеми Кнута, яка зберігає її цінні властивості, і дозволяє використовувати значно менше накладної пам'яті. Ця модифікація називається схемою Кнута-Рейн-болдта-Містен'ї або кільцевою КРМ-схемою. Зв'язкові списки рядків і стовпців закріплюються, а початкові позиції списків входять у покажчики входу. Списки, асоційовані з рядками (стовпцями), попарно не перетинаються тому можуть спільно зберігатися одним масивом  $NR$  (для стовпців  $NC$ ). Нижче наведено приклад кільцевої КРМ-схеми:

	1	2	3	4	5	6	7
$AN =$	6	9	4	7	5	2	8
$NR =$	1	3	4	2	5	7	6
$NC =$	3	5	6	7	2	1	4
$JR =$	1	2	5	6			
$JC =$	2	1	0	4			

Рис. 1.15. Кільцева КРМ-схема

Подана схема більш щільна, порівняно зі схемою Кнута. Однак якщо необхідно переглядати елементи деякого рядка (або стовпця), то не можна отримати інформацію про стовпцеві (маленькі) індекси цих елементів. Для того щоб знайти елемент  $a_{ij}$  спочатку необхідно зробити сканування  $i$ -й рядка і визначити множину  $S_i$ , яка є позицією елементів рядка в масиві  $AN$ :

$$S_i = \{p | AN(p) \text{ є елемент } i\text{-й рядка}\}$$

Мається на увазі, що повторення відсутні, тобто кожній парі  $p, q$  рядкового  $i$  і стовпцевого  $j$  індексів відповідає найбільш одна позиція в кожному з масивів AN, NR і NC. Далі скануванням  $j$ -го стовпцевого списку визначається безліч позицій масиву AN, де зберігаються елементи  $j$ -го стовпця [45]. Після чого необхідно знайти  $S_{ij} = S_i \cap S_j$ ,  $S_{ij}$  є порожнім, або містить тільки один елемент. Якщо  $S_{ij}$  порожнє, то  $a_{ij}$ . Якщо  $S_{ij}$  містить одне число, припустимо,  $k$ , то  $a_{ij} = AN(k)$ . Зворотнє завдання, тобто за заданою позицією  $k$  в AN знайти відповідні рядковий і стовпцевий індекси  $i$  та  $j$  має рішення тільки у разі перегляду всієї матриці.

Нижче наведено варіант KPM-схеми, що дозволяє шукати необхідні індекси.

	1	2	3	4	5	6	7
AN =	6	9	4	7	5	2	8
NR =	-1	3	4	-2	-3	7	-4
VC =	3	5	6	7	-1	-2	-4
JR =	1	2	5	6			
JC =	2	1	0	4			

Рис. 1.16. Модифікована KPM-схема

Вказівники входу для рядків та стовпців включаються до відповідних кільцевих списків. І тому існує кілька способів; ілюстрований вище спосіб, використовує негативні числа посилань на покажчики входу рядків і стовпців. У цій схемі достатньо переглядати ряд доти, доки не буде знайдено негативне посилання. Абсолютна її величина є номером ряду; в той же час вона відсилає до відповідного покажчика входу, тому перегляд ряду при необхідності може продовжуватися. Припустимо, необхідно знайти рядковий індекс елемента AN(3). Так як  $NR(3) = 4$  і  $NR(4) = -2$ , то шуканий індекс дорівнює 2. Крім того,  $JR(2) = 2$ , тому AN(2) містить наступний елемент 2-го рядка. Щоб знайти в масиві AN елемент  $a_{ij}$  можна застосувати описану

вище процедуру. Інша можливість полягає в тому, щоб переглянути  $i$  рядок і знайти стовпцевий індекс кожного елемента, скануючи відповідні стовпцеві списки, поки не зустрінеться  $j$ .

Існує ще один варіант схеми Кнута він використовується для зберігання симетричних позитивно визначених матриць, елементами яких можуть бути як числа, так і підматриці. Називається ця схема вперед рядком - назад по стовпцю. Зберігаються тільки діагональ та верхній трикутник матриці; потрібний лише один масив покажчиків входу, які відсилають до діагональних елементів [54].

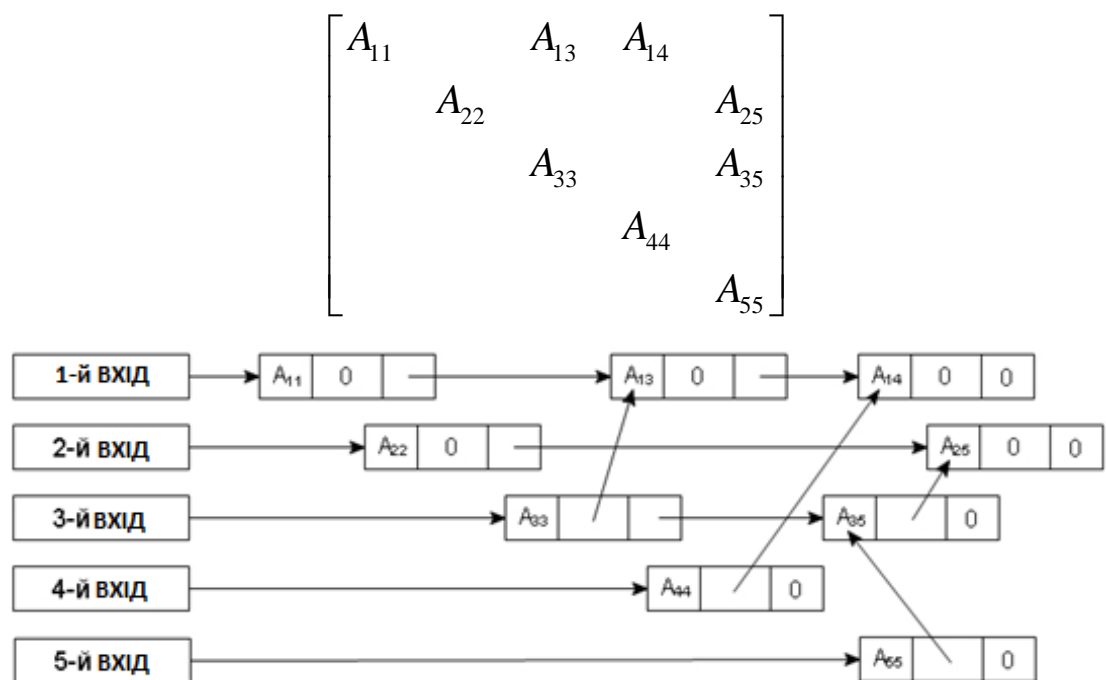


Рис. 1.17. Модифікація схеми Кнута для зберігання симетричних матриць з ненульовими діагональними елементами

Від кожного діагонального елемента починаються два списки: один описує частину відповідного рядка праворуч від діагоналі і проходить у прямому напрямку, інший описує частину відповідного стовпця над діагоналлю і проходить у зворотному напрямку, тобто знизу вгору. Це пояснює назву схеми. Приклад показано на рис. 1.17. Ідею схеми досить легко поширити на матриці загального вигляду. Якщо є діагональ з ненульових елементів, то від кожного діагонального елемента можуть



починатися чотири списки. З іншого боку, може вистачити двох неупорядкованих списків: оскільки, під час роботи з розрідженими матрицями впорядковані представлення не завжди бажані [14].

Одна зі схем зберігання розріджених матриць, що найбільш широко використовуються, являє собою розріджено малий формат. Ця схема пред'являє мінімальні вимоги до пам'яті і водночас є зручною для кількох важливих операцій над розрідженими матрицями: додавання, множення, перестановок рядків і стовпців, транспонування, рішення лінійних систем з розрідженими матрицями коефіцієнтів як прямими, так і ітераційними методами та іншими. Значення ненульових елементів матриці та відповідні стовпцеві індекси зберігаються у цій схемі по рядках у двох масивах; назовемо їх відповідно AN та JA. Використовується також масив покажчиків (IA), що відзначає позиції масивів AN та JA, з яких починається опис чергового рядка. Додаткова компонента в IA містить покажчик першої вільної позиції JA і AN. Нижче наведено приклад розрідженого рядкового зберігання для матриці A.

$$A = \begin{bmatrix} 0 & 0 & 1 & 3 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Матриця A представляється так:

позиція = 1   2   3   4   5   6

IA = 1   4   4   6

JA = 3   4   8   6   8

AN = 1   3   5   7   1

Рис. 1.18. Розріджений малий формат

Опис 1-го рядка матриці починається з позиції IA(1) = 1 масивів AN та JA. Оскільки опис 2-го рядка починається з IA(2) = 4, це означає, що 1-му рядку в AN і JA відповідають позиції 1, 2 і 3. Наведеному прикладі IA(1) = 1 перший рядок починається з JA(1) ) та AN(1). IA(2) = 4 другий рядок починається з JA(4) та AN(4). IA(3) = 4 третій рядок починається з JA(4) та

AN(4). Оскільки це ті ж позиції, з яких починається другий рядок, це означає, що 2-й рядок порожній.  $IA(4) = 6$  це перша вільна позиція у JA та AN. Таким чином, опис 3-го рядка закінчується в позиції  $6 - 1 = 5$  масивів JA та AN.

У загальному випадку опис  $r$ -го рядка матриці A зберігається в позиціях з  $IA(r)$  до  $IA(r + 1) - 1$  масивів JA та AN, за винятком рівності  $IA(r + 1) = IA(r)$ , що означає, що  $r$ -й рядок порожній. Якщо матриця має  $m$  рядків, то IA містить  $m + 1$  позицій.

Даний спосіб подання називають повним, оскільки представлена вся матриця A, і впорядкованим, оскільки елементи кожного рядка зберігаються відповідно до зростання стовпцевих індексів. Таким чином, це рядкове подання, повне та впорядковане, або скорочено RR(C)O [26].

Масиви IA і JA представляють портрет матриці A, що задається як безліч списків суміжності асоційованого з графом матриці A. Якщо в алгоритмі розмежовані етапи символічної обробки та чисельної обробки, то масиви IA і JA заповнюються на першому етапі, а масив AN – на другому.

Існує ще один варіант малої схеми зберігання, орієнтований для додатків, де доводиться виконувати операції і над рядками, і над стовпцями. Матриця A зберігається рядками, як описано вище; крім того, визначають портрет  $A^T$  і зберігають по рядках. Термінове уявлення портрета  $A^T$  рівносильне стовпцевому уявленню портрета A. Його можна отримати транспонуванням рядкового портрета A. Ця схема застосовувалася під час вирішення завдань лінійного програмування [26].

Для несиметричних матриць була запропонована інша, спрощена рядкова схема [45]. Ненульові елементи зберігають у двовимірному масиві з розмірами  $n \times m$ , де  $n$  – порядок матриці,  $m$  – максимальна кількість ненульових елементів у рядку. Ця схема припускає просту обробку; недолік її в тому, що  $m$  може бути невідомо заздалегідь або виявитися занадто великим.

Ще одним із варіантів схем зберігання матриць великої розмірності є блокова схема зберігання. Метод розбиття великої матриці на підматриці чи блоки виникає природним чином. З блоками можна поводитися, ніби вони були елементами матриці, і блокова матриця перетворюється на матрицю з матриць. Розбиття на блоки грає важливу роль технології розріджених матриць, оскільки багато алгоритмів, сконструйовані спочатку для числових матриць, можна узагальнити у разі матриць з матриць. Велика гнучкість, що з поняттям розбиття, може давати певні обчислювальні переваги. З іншого боку, розбиття можна розглядати просто як інструмент управління даними, що допомагає організувати обмін інформацією між оперативною пам'яттю і зовнішніми пристроями, що запам'ятовують.

Розбиття рядків і стовпців може бути однаковим, але якщо вони однакові, то діагональні блоки квадратні. Зберігання блочної матриці має на увазі зберігання множини її підматриць [38]. Розглянемо неявну схему зберігання, призначену головним чином для симетричних матриць з квадратними діагональними блоками. Діагональні блоки розглядаються так, ніби вони становили єдину матрицю, і зберігаються відповідно до профільної схеми. Чисельні значення елементів зберігаються рядками в масиві  $AN$ , а покажчики на діагональні елементи кожного рядка – загалом в масиві  $IA$ . Піддіагональні блоки, що розглядаються так, як якщо б вони становили єдину матрицю, зберігаються за допомогою розрідженого рядкового формату. Ненульові елементи розміщуються по рядках у дійсному масиві  $AP$ , а відповідні стовпцеві індекси – у паралельному масиві  $JA$ . Покажчики початку рядків в  $AP$  та  $JA$  зберігаються у масиві  $IP$ ; в останній для зручності програмної реалізації включений додатковий покажчик першої вільної позиції в  $AP$  та  $JA$ . Саме розбиття визначається номерами перших рядків кожного блоку, що зберігаються в  $LP$ , де також є додаткова компонента. Масив  $LP$  можна використовувати разом із масивами  $IA$  та  $IP$ . Ця гібридна схема зберігання успішно застосовувалася для гауссового виключення у

Нижче наведено приклад симетричної блокової матриці  $A$ . Рядки та стовпці матриці  $A$  згруповані в підмножини  $(1, 2, 3, 4)$ ,  $(5, 6, 7)$  та  $(8, 9, 10)$ .

$$A = \left[ \begin{array}{cccc|cccc|cccc} 1 & & 3 & & & 17 & & & & & & \\ & 2 & 4 & 6 & & & & & & 21 & & \\ 3 & 4 & 5 & & & 18 & & & & & 24 & \\ & 6 & & 7 & & 19 & 20 & & & & & \\ - & - & - & - & & - & - & - & & - & - & - \\ 17 & & & & & 8 & & 10 & & & 23 & \\ & & 18 & 19 & & & 9 & & & 22 & & \\ & & & 20 & & 10 & & 11 & & & & 25 \\ - & - & - & - & & - & - & - & & - & - & - \\ & 21 & & & & & 22 & & & 12 & 13 & 15 \\ & & & & & & & 23 & & & 13 & 14 \\ & & 24 & & & & & & 25 & & 15 & 16 \end{array} \right]$$

[illegible]

Рис. 1.19. Неявна схема зберігання блокової матриці

36

послідовно в дійсному масиві разом з інформацією про початок кожного блоку і блоковий стовпчик, якому належить даний блок.

Гіперматрична схема виходить, якщо матриця  $A$  зберігається як матриця, елементами якої є матриці [32]. Для зберігання матриці можна використовувати будь-яку з представлених схем зберігання; відмінність буде тільки в тому, що замість чисельних значень елементів потрібно зберігати інформацію про розташування та розміри підматриць. Самі підматриці, елементами яких будуть числа, зберігаються відповідно до деякого стандартного формату. У випадку, коли матриця  $A$ , розріджена можна використовувати розріджене подання, що зберігається в оперативній пам'яті, в той час як підматриці, так само як у деякому розрідженому поданні, розміщуються в периферійній пам'яті і вносяться в оперативну тільки тоді, коли цього вимагає виконання алгоритму. Такий варіант називається понад розрідженою схемою [54]. Всі гіперматричні схеми зберігання дозволяють проводити обробку великих завдань за помірних запитів до пам'яті. Їхня головна перевага полягає в легкості, з якою користувач може зменшити пам'ять або час (ціною збільшення іншої з цих характеристик), для чого достатньо, в залежності від наявної пам'яті, задати більш дрібне або грубіше розбиття. Ця техніка успішно використовувалася в гауссовому виключенні та при вирішенні спектральних завдань. Вона передбачає, що з розріджених матриць (прямокутних, квадратних і трикутних) і векторів ефективно реалізовані такі операції, як додавання, множення, перестановки, і навіть факторизація квадратних діагональних блоків.

### **1.6. Символічна обробка та динамічні схеми зберігання**

Алгоритм для розріджених матриць може породжувати нові ненульові елементи та змінювати значення тих, що вже є в даній матриці; або він може користуватися цією матрицею, не змінюючи її, як у випадку ітераційних алгоритмів, яким потрібні лише добутки матриці з векторами. Безліч нових ненульових елементів, що додаються до вже існуючої розрідженої матриці, називається заповненням. Алгоритм може будувати і зовсім нову матрицю,

але концептуально це рівнозначно генеруванню ненульових елементів спочатку нульової матриці.

Перш ніж розпочнеться виконання такого алгоритму, необхідно переконатися, що в пам'яті є місце для нових елементів. Повинні бути встановлені закони управління пам'яттю, що визначають внутрішнє подання даних, або структуру даних; від них залежить, де і як зберігатиметься кожен новий елемент. Для структури даних є вибір: вона може бути сформована до того, як почнеться чисельна обробка, або будуватися паралельно з обробкою відповідно до потоку елементів, що обчислюються.

Структура даних, заготовлена до початку чисельної обробки, називається статичною структурою [55]. Її формування вимагає знання числа ненульових елементів та його становища в матриці ще до того, як вони реально обчислені. Ряд матричних алгоритмів дозволяє передбачення необхідної інформації. У цю категорію потрапляють додавання та множення матриць, а також гауссовий виняток, коли послідовність головних елементів відома, як буде у разі симетричних позитивно визначених матриць. Кожен такий алгоритм природно розпадається на дві частини: символічну частину, де виконується символічна обробка чи розподіл пам'яті, чи формування структури даних, і чисельну частину, де виконуються реальні обчислення чи чисельна обробка. Результат символічного етапу – статична структура даних; результат чисельного етапу – значення ненульових елементів, розміщених у клітинках пам'яті, які були заготовлені на символічному етапі [26].

Деякі статичні схеми деяких конкретних додатків можуть вимагати явного символічного етапу. Стрічкові та профільні схеми були сконструйовані таким чином, що гауссовий виняток з вибором головних елементів на діагоналі може породжувати ненульові елементи лише всередині стрічки або оболонки; отже, ці схеми є потрібними статичними структурами. Те саме справедливо при приведенні стрічкової матриці до тридіагонального виду в ході вирішення спектральних завдань; але до профільної схеми зберігання це стосується. Додавання двох матриць з

однаковою стрічкою або оболонкою зберігає стрічку або оболонку, але множення двох стрічкових матриць розширює стрічку. Якщо гауссовий виняток проводиться для стрічкової матриці і при виборі головного елемента допускаються тільки перестановки рядків, то нижня напівстрічка зберігається, а верхня півстрічка може в гіршому випадку подвоїти ширину [2].

З іншого боку, використання методу такого алгоритму мінімізації стрічки або профілю повинно розглядатися як символічний етап; так як такі алгоритми лише розподіляють пам'ять, маючи на увазі її зменшення та відповідне зменшення роботи, але не знаходять жодних чисельних результатів. До тієї ж категорії належать деякі інші методи – перерізи, деревоподібне розбиття, блочна тріангуляризація. Якщо матриця зберігається в розрідженому рядковому форматі, необхідним символічним етапом буде обчислення портрета матриці.

Статичні схеми заохочують модульність, оскільки символічний та чисельний етапи виконуються окремо і, отже, можуть бути оптимізовані незалежно. Що стосується прямих методів для лінійних систем існують ефективні процедури, що дозволяють виконувати символічний етап набагато швидше, ніж чисельний. Вони використовують фіксовану пам'ять, яка лише трохи перевищує ту, що потрібно для зберігання матриці, проте дають суттєву для користувача інформацію: кількість пам'яті і обчислювальної роботи, необхідні для чисельного етапу. Ще одна важлива перевага модульності виявляється у додатках, що вимагають повторного використання даного алгоритму при різних чисельних значеннях. Прикладами можуть служити ітераційні методи, які потребують вирішення багатьох лінійних систем з однією і тією ж матрицею та різними правими частинами, або з матрицями, що мають (при різних чисельних значеннях) однаковий портрет. У таких випадках символічний етап може бути проведений лише один раз, і у всіх наступних обчисленнях може використовуватися та сама статична структура даних.

Статичні структури даних можуть застосовуватися не завжди. Найпомітнішим винятком є метод Гаусса із вибором головного елемента для матриць загального виду [45]. Щоб запобігти сильному зростанню помилок, основні елементи вибираються під час винятку відповідно до їх чисельних значень; при цьому використовуються такі стратегії вибору головного елемента, як повний вибір, частковий вибір або вибір бар'єру. Вибір головних елементів рівносильний перестановкам рядків і стовпців, які у свою чергу впливають на розташування заповнення, що виходить. Наслідком цього є непередбачуваність портрета остаточної матриці, і рішення про те, де і як розмістити кожен новий ненульовий елемент, слід приймати, коли цей елемент вже обчислений і готовий до запису в пам'ять. Така процедура називається динамічним розподілом пам'яті та породжує динамічну структуру даних.

Якщо гауссовий виняток проводити для стрічкової матриці загального вигляду і при виборі головного елемента користуватися лише перестановками рядків, то верхня напівстрічка подвоюється, а нижня не змінюється. Суворо кажучи, ця схема статична, оскільки може бути сформована заздалегідь. Вона має недолік, властивий кожній статичній схемі: у процесі виключення не можна заощадити пам'ять, необхідно зберігати та обробляти (або піддавати перевіркам) велику кількість нулів. Справді динамічні структури вимагають схем, які не залежать від статичного по суті характеру машинної пам'яті. Ця сильна вимога зазвичай задовольняється за допомогою ефективної системи зв'язок та покажчиків. Схема Кнута та КРМ-схема добре пристосовані для цієї мети. Кожен ненульовий елемент матриці має внутрішнє подання у вигляді запису фіксованої довжини, яка зберігається в довільному місці машинної пам'яті, а потім зв'язується з відповідними списками, малим і стовпцевим. Елементи можуть бути включені або виключені в будь-якій позиції матриці без переміщення інших даних, і коли алгоритм цього вимагатиме, можна переглядати як рядки, так і стовпці.



Для динамічного зберігання розріджених матриць [41] було запропоновано використовувати записи змінної довжини. Запис розуміється як двовимірний масив спеціального типу, в який можна динамічно додавати рядки і стовпцями якого можуть бути прості змінні різних типів. Рядки, що не використовуються, розпізнаються автоматично і при необхідності схема зберігання стискається. Не потрібно зберігати покажчики та зв'язки; також немає потреби у непрякій адресації. Зв'язкові списки формуються, якщо привласнити змінній, що зберігається в рядку, значення адреси наступного рядка. Вартість цього термінах накладної пам'яті рівнозначна використанню цілої змінної на рядок. Ця схема підтримується сучасними трансляторами; її застосування вимагає від транслятора здатності обробляти зв'язки та покажчики. Програма, що спирається на цю структуру даних, витримує порівняння з програмою, яка використовує впорядковані спискові схеми [45].

### **Висновки до першого розділу**

1. Визначено основні критерії, які дозволяють ефективно зберігати та обробляти матриці великої розмірності. Такими критеріями є:
  - a. зберігання тільки ненульових елементів матриці;
  - b. оперування тільки з ненульовими елементами;
  - c. збереження розрідженості матриці.
2. Проаналізовано зберігання масивів, списків, стеків та черг. Виділено різні схеми зберігання матриць на основі розглянутих структур. Визначено, що ефективність застосування різних схем зберігання залежить від операцій, які передбачаються в алгоритмі обробки.
3. Розглянуто діагональну схему зберігання стрічкової матриці. Визначено, що у стрічкових матрицях ширина стрічки залежить від порядку, в якому розташовані рядки та стовпці. Це дозволяє проводити пошук перестановок рядків і стовпців, що призводить до зменшення ширини стрічки. У свою чергу мала ширина стрічки призводить до зниження запитів до пам'яті, що збільшує швидкість роботи.

4. Розглянуто профільну схему зберігання симетричних матриць. Визначено, що одним із напрямків використання даної схеми зберігання є стрічкова матриця високого порядку з широкою стрічкою і великою кількістю нулів, оскільки використання діагональної схеми зберігання не є неекономним рішенням.
5. Проаналізовано зв'язкові схеми розрідженого зберігання та зберігання блокових матриць. Виявлено, що проблеми реалізації і накладна пам'ять зростають паралельно з ускладненням схеми зберігання. Високо складні схеми вимагають професійної програмної реалізації, інакше їх потенційні переваги будуть втрачені. Проста схема може бути ефективнішою для завдання малого або середнього розміру. З іншого боку, можливість застосування високо складної схеми може визначати, чи можна вирішити взагалі велике завдання на даній машині.

## РОЗДІЛ 2. МАТЕМАТИЧНЕ ПРЕДСТАВЛЕННЯ МАТРИЧНИХ АЛГОРИТМІВ

### 2.1. Принципи побудови ітераційних процесів

Основні ітераційні процеси для вирішення лінійних систем можуть бути описані за допомогою наступної загальної схеми.

Нехай дана система лінійних рівнянь

$$Ax = b \quad (2.1)$$

з матрицею  $A$ . Будується послідовність векторів  $x^{(1)}, x^{(2)}, \dots, x^{(k)}, \dots$  за рекурентними формулами

$$x^{(k)} = x^{(k-1)} + H^{(k)}(b - Ax^{(k-1)}) \quad (2.2)$$

де  $H^{(1)}, H^{(2)}$  деяка послідовність матриць,  $x^{(0)}$  – початкове наближення.

Різний вибір послідовності матриць  $H^{(k)}$  призводить до різних ітераційних процесів [12].

Ітераційні процеси, що протікають за формулою (2.2), мають таку властивість, що для кожного з них точне рішення  $x^*$  є нерухомою точкою.

Це означає, що й за початкове наближення  $x^{(0)}$  взято  $x^*$ , всі наступні наближення також рівні  $x^*$ .

Зворотньо, будь-який ітераційний процес, для якого  $x^*$  є нерухомою точкою, що протікає за формулою

$$x^{(k)} = C^{(k)}x^{(k-1)} + Z^{(k)} \quad (2.3)$$

де  $C^{(k)}$  послідовність матриць,  $Z^{(k)}$  послідовність векторів. Може бути поданий у вигляді (2.2). Для  $x^*$  маємо  $x^* = C^{(k)}x^* + Z^{(k)}$  звідки

$$\begin{aligned} x^{(k)} &= x^* + C^{(k)}(x^{(k-1)} - x^*) = x^{(k-1)} + (C^{(k)} - E)(x^{(k-1)} - x^*) = \\ &= x^{(k-1)} + (E - C^{(k)})A^{-1}A(x^* - x^{(k-1)}) = x^{(k-1)} + H^{(k)}(b - Ax^{(k-1)}) \end{aligned}$$

при

$$H^{(k)} = (E - C^{(k)})A^{-1}$$

Неважко дати необхідні та достатні умови для того, щоб ітераційний процес (2.2) схилювався до рішення за будь-якого початкового вектора.

$$x^* - x^{(k)} = (E - H^{(k)}A)(E - H^{(k-1)}A)\dots(E - H^{(1)}A)(x^* - x^{(0)})$$

Для того, щоб  $x^* - x^{(k)} \rightarrow 0$  при будь-якому початковому векторі  $x^{(0)}$  необхідно і достатньо, щоб матриця

$$T^{(k)} = (E - H^{(k)}A)(E - H^{(k-1)}A)\dots(E - H^{(1)}A)$$

прагнула до нуля, навіщо, своєю чергою, достатньо, щоб будь-яка матриця  $T^{(k)}$  прагнула до нуля. Виділена умова дає лише загальну думку для побудови умов збіжності конкретних ітераційних процесів [26].

Найпростішими серед ітераційних процесів є стаціонарні процеси, в яких матриці  $H^{(k)}$  не залежать від номера кроку  $k$ . Зокрема, при  $H^{(k)} = E$  виходить класичний процес послідовних наближень. Будь-який стаціонарний процес  $H \neq E$  можна розглядати як процес послідовних наближень, застосований до рівносильної системи  $HAX = Hb$ , підготовленої до застосування методу послідовних наближень. Здійснювати таку підготовку зазвичай немає необхідності, і такого роду розгляд стаціонарних процесів лише дає зручний засіб для їхнього теоретичного дослідження. Близькими до стаціонарних ітераційних процесів є циклічні, в яких матриці  $H^{(k)}$  періодично повторюються через кілька  $p$  кроків. З кожного циклічного процесу можна отримати рівносильний йому стаціонарний [20], приймаючи за один крок стаціонарного процесу результат застосування повного циклу  $p$  кроків вихідного циклічного процесу. Нестаціонарні ітераційні процеси, своєю чергою, можна поділити на два типи.

1. Нестаціонарні ітераційні процеси зі зміною матриці здійснюються на кожному кроці.

2. Стаціонарні процеси з прискоренням збіжності у вигляді заміни час від часу стаціонарної матриці  $H$ , що визначає процес, на деякі спеціальним чином підібрані, матриці  $H^{(k)}$ .

Вибір матриці для стаціонарного процесу і матриць для нестаціонарного може здійснюватися багатьма різними способами виходячи з різних принципів [34].

Можлива побудова матриць так, щоб ітераційний процес схилювався до рішення для більш широкого класу систем рівнянь. Можлива протилежна точка зору, в силу якої при побудові матриць максимально використовуються приватні особливості даної системи для отримання ітераційного процесу, що має найшвидшу збіжність. Для застосування ітераційного процесу, побудованого виходячи з останнього принципу, потрібно мати більшу інформацію про матрицю коефіцієнтів системи, зокрема про розташування її власних значень.

Важливим принципом побудови ітераційних процесів є принцип релаксації. Під цим принципом розуміють вибір матриць  $H^{(k)}$  із деякого заздалегідь окресленого класу матриць так, щоб на кожному кроці процесу зменшувалася будь-яка величина, що характеризує точність розв'язання системи [25].

Серед релаксаційних методів найбільше розроблені координатні. В яких матриці  $H^{(k)}$  підібрані так, що на кожному кроці змінюються одна або декілька компонентів послідовних наближень, і градієнтні, в яких матриці  $H^{(k)}$  є скалярними [2].

Про точність наближення рішення  $X$  системи  $Ax = b$  природно судити за величиною вектора помилки  $Y = x^* - x$ . Проте вектор помилки може бути обчислений без знання точного рішення системи і може лише оцінюватися. Вектором, що характеризує точність наближеного рішення  $X$  системи  $Ax = b$ , може служити також вектор нев'язки [1]  $r = b - Ax$ .

Зрозуміло, що  $r = AY$ . Таким чином, релаксація може бути побудована на зменшенні будь-якої норми кожного з цих векторів.

При позитивно визначених матрицях  $A$  зручною мірою точності є так звана функція помилки

$$f(x) = (AY, Y) = (Y, r) = (A^{-1}r, r)$$

З огляду на позитивності визначеності  $A$  завжди  $f(x) \geq 0$ , причому  $f(x) = 0$  лише за  $x = x^*$ .

Функція помилки не може бути обчислена, якщо не відоме точне рішення. Однак її значення лише постійним доданком відрізняються від значень функціоналу

$$f_0(X) = (Ax, x) - 2(x, b)$$

які можуть бути обчислені без знання  $x^*$ . Тому можемо судити про зменшення функції помилки порівнюючи відповідні значення функціоналу  $f_0(x)$  [7].

Іншим важливим принципом побудови ітераційних процесів є принцип послідовного «придушення компонентів» вектора помилки в розкладанні його за власними векторами матриці коефіцієнтів системи.

## 2.2. Побудова проекційних методів

Розглянемо систему  $Ax = b$  та сформулюємо для неї наступне завдання. Нехай задані деякі два підпростори  $K \subset R^n$  та  $\alpha \subset R^n$ . Потрібно знайти такий вектор  $x \in K$ , який би гарантував рішення вихідної системи, оптимальне щодо підпростору  $\alpha$ , тобто щоб виконувалася умова

$$\forall l \in \alpha : (Ax, l) = (b, l)$$

звана умовою Петрова-Галеркіна. Згрупувавши обидві частини рівності за властивостями скалярного добутку і помітивши, що  $b - Ax = r_x$ , цю умову можна подати у вигляді

$$\forall l \in \alpha : (r_x, l) = 0 \quad (2.4)$$

тобто  $r_x = b - Ax \perp \alpha$ . Таке завдання називається завданням проектування рішення  $x$  на підпростір  $K$  ортогонально до підпростору  $\alpha$  [5].

У більш загальній постановці завдання виглядає так. Для вихідної системи (2.1) відомо деяке наближення  $x_0$  до рішення  $x^*$ . Потрібно уточнити його поправкою  $\sigma_x \in K$  таким чином, щоб  $b - A(x_0 + \sigma_x) \perp \alpha$ . Умову Петрова-Галеркіна у цьому разі можна записати як

$$\forall l \in \alpha : (r_{x_0 + \sigma_x}, l) = ((b - Ax_0) - A\sigma_x, l) = (r_0 - A\sigma_x, l) = 0$$

Нехай  $\dim K = \dim \alpha = m$ . Введемо у підпросторах  $K$  та  $\alpha$  базиси  $\{v_j\}_{j=1}^m$  та  $\{w_j\}_{j=1}^m$  відповідно. Неважко бачити, що (2.4) виконується тоді і тільки тоді, коли

$$\forall j(1 \leq j \leq m): (r_0 - A\sigma_x, w_j) = 0 \quad (2.5)$$

При введенні для базисів матричних позначень  $V = [v_1 | v_2 | \dots | v_m]$  та  $W = [w_1 | w_2 | \dots | w_m]$  можна записати  $\sigma_x = V_y$  де  $y \in R^m$  — вектор коефіцієнтів [1]. Тоді (2.5) може бути записано у вигляді

$$W^T(r_0 - AV_y) = 0 \quad (2.6)$$

звідки  $W^T AV_y = W_{r_0}^T$  та

$$y = (W^T AV)^{-1} W_{r_0}^T \quad (2.7)$$

Таким чином, рішення має уточнюватися відповідно до формули

$$x_1 = x_0 + V(W^T AV)^{-1} W_{r_0}^T \quad (2.8)$$

з якої відразу випливає важлива вимога: у практичних реалізаціях проєкційних методів підпростору  $K$  та  $\alpha$ , їх базиси повинні вибиратися

так, щоб матриця  $W^T AV$  або була малої розмірності, або мала просту структуру, зручну для обігу [15].

З (2.6) також випливає співвідношення

$$V_y = A^{-1}r_0 = A^{-1}(b - Ax_0) = x_* - x_0$$

Тобто  $Vy = \sigma_x$  являє собою проекцію на підпростір  $K$  різниці між точним рішенням та початковим наближенням.

Нехай є набір пар підпросторів  $\langle \kappa_i, \alpha_i \rangle_{i=1}^q$  таких, що  $\kappa_1 \oplus \kappa_2 \dots \oplus \kappa_q = R^n$  і  $\alpha_1 \oplus \alpha_2 \dots \oplus \alpha_q = R^n$ . Тоді очевидно, що послідовне застосування описаного раніше процесу до всіх таких пар призведе до рішення  $x_q$ , що задовольняє вихідну систему  $Ax = b$  [5]. Відповідно, у загальному вигляді алгоритм будь-якого методу проекційного класу може бути записаний таким чином:

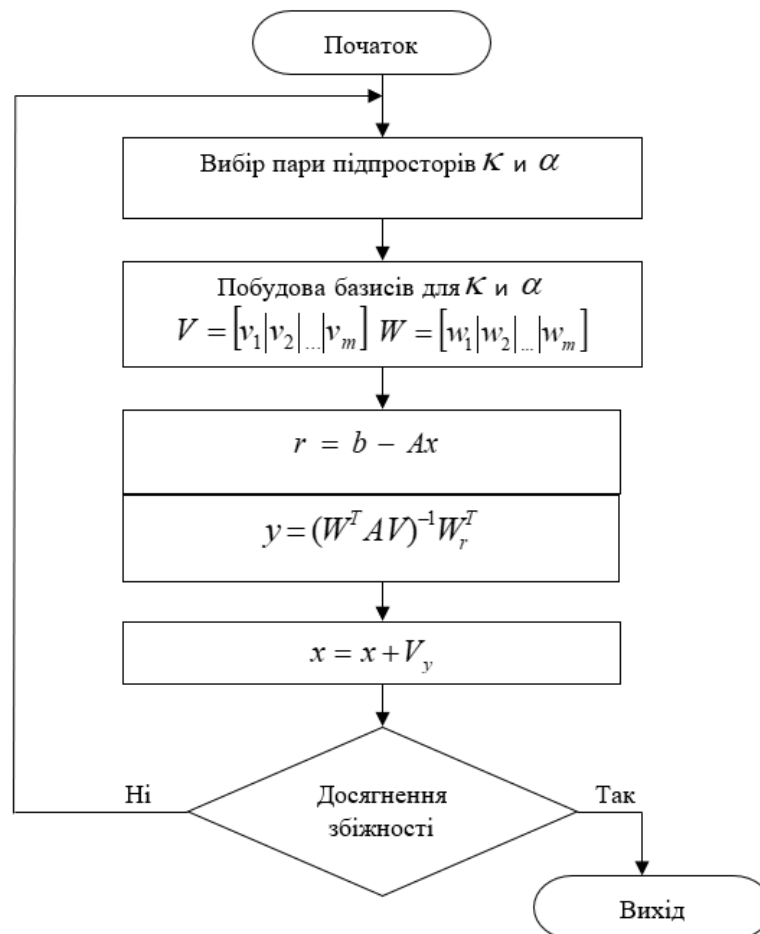


Рис. 2.1. Блок-схема методу розв'язання проекційного класу



Найбільш простою ситуацією є випадок, коли простори  $\mathcal{K}$  та  $\mathcal{A}$  одновимірні. Нехай  $\mathcal{K} = \text{span}\{v\}$  і  $\mathcal{A} = \text{span}\{w\}$  [1]. Тоді (2.8) набуває вигляду

$$x_{k+1} = x_k + \gamma_k v_k \quad (2.9)$$

причому коефіцієнт  $\gamma_k$  легко знаходиться з умови ортогональності

$$r_k - A(\gamma_k v_k) \perp w_k :$$

$$(r_k - \gamma_k A v_k, w_k) = (r_k, w_k) - \gamma_k (A v_k, w_k) = 0$$

Звідки  $\gamma_k = (r_k, w_k) / (A v_k, w_k)$  [5]. Якщо вибрати  $v_k = w_k = r_k$ .

Тоді (2.9) набуває вигляду

$$x_{k+1} = x_k + \frac{(r_k, r_k)}{(A r_k, r_k)} r_k \quad (2.10)$$

Оскільки вираз у знаменнику є квадратичною формою  $r_k^T A r_k$ , збіжність процесу гарантована, якщо матриця  $A$  є симетричною і позитивно визначеною [21].

Цей спосіб є спосіб якнайшвидшого спуску; можна показати, що з кожної ітерації у ньому мінімізується функціонал  $\Phi(x) = \|x - x_*\|_A^2$  у бік  $-\nabla \Phi(x)$  [58].

В силу позитивної визначеності  $A$ , квадратична форма  $(x - x_*)^T A (x - x_*) = \Phi(x)$  досягає свого мінімуму (рівного нулю)  $x = x_*$  та строго випукла. При цьому

$$\begin{aligned} \Phi(x) &= (A(x - x_*), x - x_*) = \\ &= (Ax, x) - (Ax, x_*) - (b, x) + (b, x_*) = \\ &= x^T A x - x_*^T A x - b_x^T + b_{x_*}^T \end{aligned}$$

В силу симетричності матриці  $A$  справедливо  $x_*^T Ax = b^T (A^{-1}) Ax = b^T x$ , і функціонал дорівнює

$$\Phi(x) = x^T Ax - 2b^T x + b^T x_* \quad (2.11)$$

його градієнт  $\nabla \Phi(x) = 2Ax - 2b = -2r_x$  [53]; отже,  $-\nabla \Phi(x) \uparrow r_x$ .

Покажемо тепер, що вибір  $\gamma = \|r_x\| / (r_x^T A r_x)$  доставляє мінімум  $\Phi(x)$  у цьому напрямі. Підставим в (2.11) вираз  $x + \gamma r$ ; останнім доданком  $b^T x_*$  при цьому можна знехтувати, так як він постійний і не впливає на процес мінімізації [5]. Знову враховуватимемо симетрію матриці  $A$

$$\begin{aligned} f(\gamma) &= \Phi(x + \gamma r) = (x + \gamma r)^T A(x + \gamma r) - 2b^T (x + \gamma r) = \\ &= x^T Ax + 2\gamma x^T Ar + \gamma^2 r^T Ar - 2b^T x - 2\gamma b^T r = \\ &= [x^T Ax - b^T x] - b^T x + 2\gamma [x^T Ar - b^T r] + \gamma^2 r^T Ar = \\ &= -(r + b)_x^T - 2\gamma r^T r + \gamma^2 r^T Ar \end{aligned}$$

Знайдемо  $\min_{\gamma} f(\gamma)$ ; враховуючи опуклість  $\Phi(x)$ , для цього достатньо знайти стаціонарну точку  $f(\gamma)$ :

$$f'(\gamma) = 2\gamma r^T Ar - 2r^T r = 0 \Rightarrow \gamma = \frac{r^T r}{r^T Ar} = \frac{(r, r)}{(Ar, r)}$$

що збігається з (2.10). У практичних завданнях метод якнайшвидшого спуску має досить повільну збіжність, відповідний аналіз наведено в [24].

Якщо вибрати  $v_k = A^T r_k$  і  $w_k = Av_k$ . Формула (2.9) набуває вигляду

$$\begin{aligned} x_{k+1} &= x_k + \frac{(r_k, AA^T r_k)}{(AA^T r_k, AA^T r_k)} A^T r_k = \\ &= x_k + \frac{(A^T r_k, A^T r_k)}{(A^T AA^T r_k, A^T r_k)} A^T r_k \end{aligned} \quad (2.12)$$

Цей спосіб є спосіб якнайшвидшого зменшення невязки; умовою його збіжності є невиродженість матриці  $A$ .

Порівнюючи (2.12) і (2.10), неважко переконатися, що метод якнайшвидшого зменшення незв'язки збігається з методом якнайшвидшого спуску, застосованим до системи  $A^T Ax = A^T b$ . Тоді на підставі співвідношення (2.10) можна стверджувати, що в методі (2.12) на кожному кроці мінімізується функціонал  $\Phi(x) = \|b - Ax\|_2^2$  у напрямку  $-\nabla\Phi(x)$ .

Якщо вибір на  $k$ -й ітерації  $v_k = w_k = A^T e_k$  це дає методи класу ABS [1]. Щоб різних ітерацій виконувалося умова  $\kappa_i \perp \kappa_j$ , можливе або зберігання всіх  $v_k$  із їх ортогоналізацією по мірі знаходження (схема Хуанга), або перерахунок матриці  $A$  (схема Абрамова). Перший варіант веде до збільшення витрат пам'яті реалізації алгоритму, другий – до зміни заповненості матриці. Отже, дані методи придатні лише вирішення щільних чи невеликих систем; з іншого боку, їх єдиною умовою збіжності є існування рішення як такого, тому даний клас придатний для СЛАР з виродженими і неквадратними матрицями [43].

Безпосередньо з визначення векторів  $v_k$  випливає, що у даних методах на  $k$ -й ітерації поправка до рішення обчислюється з умови звернення  $k$ -го рівняння у тотожність [5].

У найпростішому випадку, якщо припускати, що матриця  $A$  квадратна і невироджена, обчислення описуються як алгоритм представлений на рис. 2.2. і схема має такий вигляд:

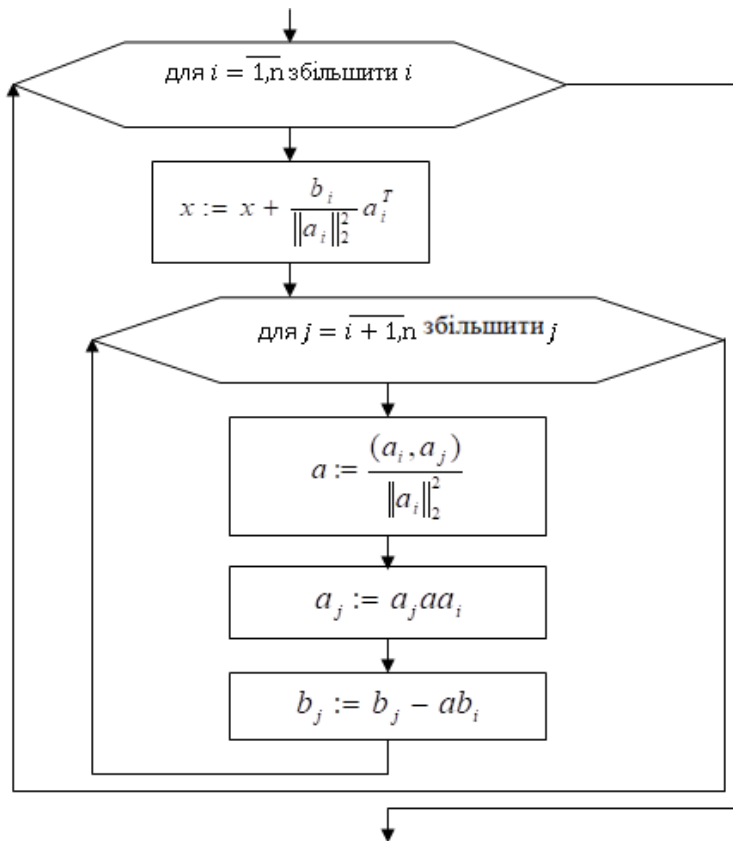


Рис. 2.2. Блок-схема методів розв'язання класу ABS

### 2.3. Підпростори Крилова

При побудові та реалізації проекційних методів важливу роль відіграють так звані підпростори Крилова, які часто обираються як  $\mathcal{K}$ . Підпростором Крилова розмірності  $m$ , породженим вектором  $v$  та матрицею  $A$  називається лінійний простір

$$K_m(v, A) \stackrel{\text{def}}{=} \text{span}\{v, Av, A^2v, \dots, A^{m-1}v\} \quad (2.13)$$

В якості вектора  $v$  зазвичай вибирається нев'язка початкового наближення  $r_0$ ; тоді вибір підпростору  $\mathcal{K}$  та спосіб побудови базисів підпросторів повністю визначає обчислювальну схему методу [5].

До ідеї використання підпросторів Крилова можна дійти таким чином. Оскільки відомо, що в побудові релаксаційних методів використовується уявлення матриці  $A$  як  $A = D - E - F$ , а методи Якобі і Гаусса-Зейделя є окремими випадками класу методів, заснованого на розщепленні  $A$  як різниці

$A = K - R$  двох матриць  $K$  і  $R$ . Тоді вихідна система (2.1) може бути записана у вигляді

$$Kx = b + Rx = b + (K - A)x$$

що дозволяє побудувати ітераційний процес

$$Kx_{k+1} = Kx_k + (b - Ax_k)$$

або, еквівалентний вираз,

$$x_{k+1} = x_k + K^{-1}r_k \quad (2.14)$$

Виберемо  $K = I$  і  $R = I - A$ , тоді процес (2.14) буде зведений до вигляду

$$x_{k+1} = x_k + r_k \quad (2.15)$$

звідки

$$x_k = x_0 + r_0 + r_1 + \dots + r_{k-1} \quad (2.16)$$

Помноживши обидві частини (2.15) зліва на  $(-A)$  та додавши до них  $b$  отримаємо

$$b - Ax_{k+1} = b - Ax_k - Ar_k = r_k - Ar_k$$

що дозволяє знайти вираз для нев'язки на  $k$ -ої ітерації через нев'язку початкового наближення:

$$r_k = (I - A)r_{k-1} = (I - A)^k r_0 \quad (2.17)$$

Після підстановки (2.17) в (2.16) отримуємо

$$x_k = x_0 + \left[ \sum_{j=0}^{k-1} (I - A)^j \right] r_0$$

тобто  $\sigma_x \in \text{span}\{r_0, Ar_0, \dots, A^{k-1}r_0\} = k_k(r_0, A)$

Безпосередньо з визначення впливають такі властивості підпросторів Крилова. Нехай  $q$  поліном, такий що  $q(A)v = 0$ , причому має ступінь  $\deg q = \mu$  мінімальний серед усіх таких поліномів. Тоді

- $\forall m(m \geq \mu): K_m(v, A) = K_\mu(v, A)$ . Більше того,  $K$  інваріантно щодо  $A$ .
- $\forall m: \dim(K_m) = m \Leftrightarrow m \leq \mu$

Крім того, з (2.17) випливає, що в методах, що використовують підпростір Крилова, нев'язка  $k$ -ої ітерації виражається через початкову нев'язку деяким матричним поліномом [13].

До ітераційних методів підпростору Крилова відносяться: метод квазімінімізації нев'язки, метод Річардсона, метод спряжених градієнтів та його варіації, такі як метод біспряжених градієнтів, стійкий двонаправлений метод спряжених градієнтів, квадратичний метод спряжених градієнтів. Наведемо виклад деяких з перерахованих вище методів алгоритми, яких будуть описані в бібліотеках класів.

### 2.3.1. Метод спряжених градієнтів

Метод спряжених градієнтів є одним із найефективніших методів вирішення розріджених симетричних позитивно визначених систем [29]. Нижче наведено виклад методу.

Розглядаючи систему лінійних рівнянь  $A\bar{x} = \bar{b}$  із симетричною позитивно визначеною матрицею  $A$ . Метод ґрунтується на ідеї мінімізації функції.

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

Функція  $f(x)$  досягає свого мінімального значення тоді і лише тоді, коли її градієнт  $\nabla f = Ax - b$  перетворюється у нуль, що еквівалентно рівності (2.1).

Нехай  $x_1$  – початкове наближення до рішення (точки мінімуму). На  $k$ -й ітерації метод вибирає напрям  $p_k$  і по наближенню  $x_k$  будується наближення  $x_{k+1} = x_k + \alpha_k p_k$ , де  $\alpha_k$  мінімізує величину  $f(x_k + \alpha_k p_k)$ .

Напрямок вибирається так, що мінімізує функцію не тільки на прямій, а й на лінійній оболонці вектора. Звідси випливає, якщо помилок округлення немає, то метод досягається мінімум за  $n$  кроків [18].

Нижче наведені розрахункові формули:

$$\begin{aligned} r_1 &= b - Ax_1, & p_1 &= r_1, \\ \alpha_k &= \frac{r_k^T r_k}{p_k^T Ap_k}, & r_{k+1} &= r_k - \alpha_k Ap_k, \\ \beta_k &= \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}, & p_{k+1} &= r_{k+1} + \beta_k p_k, \\ x_{k+1} &= x_k + \alpha_k p_k. \end{aligned}$$

На кожній ітерації добуток  $Ap_k$  достатньо вирахувати лише один раз.

Вектор  $r_k$  є нев'язкою для наближення  $x_k$ , тобто  $r_k = b - Ax_k$ , а вектори в послідовності  $\{r_k\}$  ортогональні попарно:

$$r_i^T r_j = 0 \quad (i \neq j),$$

В той же час вектори в послідовності  $\{p_k\}$  попарно пов'язані:

$$p_i^T Ap_j = 0 \quad (i \neq j)$$

Справедлива так само рівність

$$r_i Ap_j = r_i^T p_j = 0 \quad (i \neq j).$$

Ітерації тривають до того часу, поки відносна форма нев'язки  $\|r_k\| / \|b\|$  не стане менше заданого значення.

Зважаючи на помилки округлення, описаний процес необхідно розглядати як ітераційний. Цей процес може бути не стійким або навіть завершатися аварійно (наприклад, при діленні на 0) [33].

### 2.3.2. Метод біспряжених градієнтів

Одним із узагальнень методу спряжених градієнтів у разі лінійної системи  $A\bar{x} = \bar{b}$  з довільною квадратичною невідродженою матрицею  $A$  є метод біспряжених градієнтів. Відомо, що матриця  $A^T A$  – симетрична та позитивно визначена, а система  $A^T A x = A^T b$  еквівалентна вихідній. Для вирішення цієї системи можливе застосування методу спряжених градієнтів, проте таке застосування є не оптимальним рішенням задачі, оскільки обумовленість матриці  $A^T A$  набагато гірша за обумовленість матриці  $A$  [30].

З іншого боку, доведено, що за допомогою рекурентного співвідношення невеликого порядку домогтися попарної ортогональності нев'язок  $\{r_k\}$  у разі довільної матриці  $A$  неможливо. У методі біспряжених градієнтів послідовність ортогональних нев'язок  $\{r_k\}$  замінена на дві біортогональні послідовності  $\{r_k\}$ ,  $\{\tilde{r}\}$ . Послідовність спряжених напрямків  $\{p_k\}$  замінена на біспряжені послідовності  $\{p_k\}$ ,  $\{\tilde{p}_k\}$ . Розрахунки виконуються за формулами:

$$\begin{aligned}
 r_1 &= b - Ax_1 & p_1 &= r_1 & \tilde{r}_1 &= r_1 & \tilde{p}_1 &= p_1 \\
 a_k &= \frac{\tilde{r}_k^T r_k}{\tilde{p}_k^T A p_k} & r_{k+1} &= r_k - \alpha_k A p_k & \tilde{r}_{k+1} &= \tilde{r}_k - \alpha_k A^T \tilde{p}_k \\
 \beta_k &= \frac{\tilde{r}_{k+1}^T r_{k+1}}{\tilde{r}_k^T r_k} & p_{k+1} &= r_{k+1} + \beta_k p_k & \tilde{p}_{k+1} &= \tilde{r}_{k+1} + \beta_k \tilde{p}_k
 \end{aligned}$$



$$x_{k+1} = x_k + a_k p_k$$

На кожній ітерації необхідно обчислити один добуток  $Ap_k$  та один добуток  $A^T \tilde{p}_k$ . Покажемо, що вектори в послідовностях  $\{r_k\}$  і  $\{\tilde{r}\}$  біоортогональні:

$$\tilde{r}_i^T r_j = 0 \quad (i \neq j),$$

а вектори в послідовностях  $\{p_k\}$  та  $\{\tilde{p}_k\}$  біспряжені:

$$\tilde{p}_i^T Ap_j = 0 \quad (i \neq j).$$

Справедлива також рівність

$$\tilde{r}_i^T Ap_j = r_i^T \tilde{p}_j = 0 \quad (i \neq j).$$

#### 2.4. Передумовлення

Нехай  $M$  деяка невиражена матриця розмірності  $n$ . Домноживши (2.1) на  $M^{-1}$ , отримаємо систему

$$M^{-1}Ax = M^{-1}b \quad (2.18)$$

яка через невиродженість  $M$  має те саме точне рішення  $x_*$ . Ввівши позначення  $\hat{A} = M^{-1}A$  і  $\hat{b} = M^{-1}b$ , запишемо (2.9) у вигляді

$$\hat{A}x = \hat{b} \quad (2.19)$$

Запис (2.19) алгебраїчно еквівалентна (2.1), а спектральні характеристики матриці  $\hat{A}$  відрізняються від характеристик матриці  $A$ , що призводить до зміни швидкості збіжності чисельних методів (2.19) по відношенню до (2.1) в кінцевій арифметиці.

Процес переходу від (2.1) до (2.19) з метою покращення характеристик матриці для прискорення збіжності до рішення називається передумовою, а матриця  $M$  - матрицею передумови [5].

З (2.19) випливає вимога: матриця  $M$  має бути близька до матриці  $A$ . (Вибір  $M = A$  наводить (2.1) до виду  $Ix = A^{-1}b$ , проте немає практичного сенсу, оскільки вимагає знаходження  $A^{-1}$ , що з суті і зводиться до рішення (2.1)). Другою природною вимогою є вимога легкої обчислюваності матриці  $M$ .

Знаходження добутку  $M^{-1}A$  для обчислення  $\hat{A}$  у загальному випадку веде до зміни портрета ( $P_A \neq P_{\hat{A}}$ ). Тому на практиці використовується інший підхід.

Нев'язка  $\hat{r}$  системи (2.19) пов'язана з нев'язкою  $r$  системи (2.1) нижче наведеним співвідношенням

$$M\hat{r} = r \quad (2.20)$$

яке справедливе і для матрично-векторних добутків  $z = Aq$  і  $\hat{z} = \hat{A}q$

$$\hat{z} = \hat{A}q = M^{-1}Aq \Rightarrow M\hat{z} = Aq = z \quad (2.21)$$

Це дозволяє замість явного переходу від (2.1) до (2.19) вводити схеми методів коригучі кроки для обліку впливу передумовної матриці.

З (2.21) випливає ще одна умова: структура матриці передумови повинна допускати легке і швидке рішення «зворотних до передумов» систем виду  $M\hat{r} = r$ .

Таким чином:

- $M$  має бути близька до  $A$ ;
- $M$  повинна бути легко обчислювана;
- $M$  повинна бути легко оборотна.

Вибір в якості  $M$  як деякої діагональної матриці задовольняє двом останнім вимогам з трьох перерахованих і зводить передумову до масштабування СЛАР. Як відомо, у ряді випадків масштабування здатне значно прискорити процес розв'язання.

Описана вище передумова іноді називається лівою, так як домноження матриці СЛАР на матрицю передумови проводиться зліва. Інший підхід ґрунтується на переході від (2.1) до системи

$$AM^{-1}y = b \quad (2.22)$$

у якої точне рішення  $y_*$  пов'язане стічним рішенням  $x_*$  вихідної СЛАР [48].

$$x_* = M^{-1}y_* \quad (2.23)$$

Передумовлення (2.22) реалізується шляхом виконання матрично-векторних множень  $z = Aq$  подвійних множень  $z = A(M^{-1}q)$  крім того. При досягненні необхідної точності здійснюється перерахунок рішення відповідно до (2.23). Така схема передумови називається правою.

Для прискорення збіжності в проекційному методі підпростору Крилова часто використовуються передумови. Опишемо застосування передумови до вищевикладених методів сполучених та біспряжених градієнтів.

Нехай матриця  $M$  є матрицею передумови. Система  $M^{-1}A = M^{-1}b$  еквівалентна вихідній системі, проте застосування методу спряжених градієнтів до нової системи неможливе, оскільки матриця  $M^{-1/2}A$  у загальному випадку не симетрична. Ідея методу спряжених градієнтів із передумовою полягає у розгляді системи

$$M^{-1/2}AM^{-1/2}y = M^{-1/2}b \quad (2.24)$$

де  $M^{-1/2}$  - обумовлена єдиним чином симетрична позитивно визначена матриця, така, що  $(M^{-1/2})^2 = M$ . Нові невідомі  $y$  пов'язані з невідомим  $x$  системами (2.1) за формулами  $x = M^{-1/2}y$ . Застосуємо до системи (2.24) метод спряжених градієнтів.

Для ефективної реалізації цієї ідеї вводиться на розгляд нова послідовність векторів  $\{z_k\}$ . Розглянуті формули набувають вигляду:

$$r_1 = b - Ax_1, \quad p_1 = r_1,$$

$$z_k = M^{-1}r_k$$

$$\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}, \quad r_{k+1} = r_k - \alpha_k A p_k,$$

$$\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}, \quad p_{k+1} = z_{k+1} + \beta_k p_k,$$

$$x_{k+1} = x_k + a_k p_k.$$

На кожній ітерації необхідно один раз обчислити добуток  $A p_k$  і один раз обчислити вираз  $M^{-1}r_k$ . З цього видно, що явного  $M^{-1/2}$  не відбувається.

Розглянемо деякі способи вибору передумови. Якщо діагональні елементи матриці  $A$  сильно відрізняються один від одного, то одним з можливих рішень є взяття в якості  $M$  як діагональну частину матриці  $A$ . У цьому випадку добуток  $M^{-1}A$  можна розцінювати як балансування матриці  $A$ , яке покращує її зумовленість. У той же час обчислення виразу  $M^{-1}r$  досить просте і використовує  $n + O(1)$  операцій [4].

Інший спосіб вибору передумови заснований на неповному розкладанні Холецкого. Нехай  $A = LL^T$  - розкладання Холецкого розрідженої матриці  $A$ .  $L$  може мати більш заповнену структуру, ніж нижньотрикутна частина матриці  $A$ . Неповним розкладанням Холецкого називається наближена рівність  $A = \tilde{L}\tilde{L}^T$ , де  $\tilde{L}$  нижньотрикутна невиводжена розріджена матриця

з невеликим заповненням. Нехай  $M = \tilde{L}\tilde{L}^T$ . Оскільки  $A \approx M$ , то матриця  $M^{-1}A$  близька до одиничної і, отже, добре обумовлена. З іншого боку, якщо  $\tilde{L}$  вже знайдено, то обчислення матриці  $M^{-1}r$  зводиться до розв'язання двох трикутних систем  $\tilde{L}y = r$  і  $\tilde{L}x = y$  що вимагає час  $2n^2 + O(n)$ .

Оцінку швидкості збіжності методу спряжених градієнтів з обумовленням можна дати в термінах числа обумовленості  $k = \text{cond}_2(M^{-1}A)$  матриці  $M^{-1}A$ . Нехай  $x$  - точне рішення системи  $A\bar{x} = \bar{b}$  з симетричною позитивно визначеною матрицею  $A$ . Тоді для методу спряжених градієнтів із симетричною позитивно визначеною передумовою  $M$  справедлива оцінка

$$\|x_k - x\|_A \leq 2a^k \|x_1 - x\|_A \quad (2.25)$$

де  $a = (\sqrt{k} - 1)/(\sqrt{k} + 1)$ .

Нерівність (2.25) є лише верхньою оцінкою. Якщо власні значення матриці  $M^{-1}A$  добре відокремлені. То часто спостерігається збіжність зі швидкістю геометричної прогресії зі зменшуваним знаменником.

У разі методу біспряжених градієнтів як передумову можна взяти діагональну частину матриці  $A$  або неповне LU-розкладання, тобто наближення виду  $A \approx \tilde{L}\tilde{U}$ , де  $\tilde{L}$  нижньотрикутна, а  $\tilde{U}$  верхньотрикутна невинроджені матриці з невеликим заповненням.

Розрахункові формули методу біспряжених градієнтів з обумовленням мають вигляд:

$$\begin{aligned} r_1 &= b - Ax_1 & p_1 &= r_1 & \tilde{r}_1 &= r_1 & \tilde{p}_1 &= p_1 \\ z_k &= M^{-1}r_k & & & \tilde{z}_k &= M^{T^{-1}}r_k & & \end{aligned}$$

$$\begin{aligned}
a_k &= \frac{\tilde{r}_k^T z_k}{\tilde{p}_k^T A p_k} & r_{k+1} &= r_k - \alpha_k A p_k & \tilde{r}_{k+1} &= \tilde{r}_k - \alpha_k A^T \tilde{p}_k \\
\beta_k &= \frac{\tilde{r}_{k+1}^T z_{k+1}}{\tilde{r}_k^T r_k} & p_{k+1} &= z_{k+1} + \beta_k p_k & \tilde{p}_{k+1} &= \tilde{z}_{k+1} + \beta_k \tilde{p}_k \\
x_{k+1} &= x_k + a_k p_k
\end{aligned}$$

На кожній ітерації необхідно обчислити добуток  $A p_k$ ,  $A^T \tilde{p}_k$ ,  $M^{-1} r_k$ ,  $M^{T^{-1}} r_k$ .

## 2.5. LU-факторизація

Одним із широко відомих способів розкладання матриць на множники є LU-факторизація [5], що дозволяє подати матрицю  $A$  у вигляді

$$A = L_A U_A \quad (2.26)$$

де  $L_A$  – нижньо і  $U_A$  – верхньотрикутні матриці відповідно.

Дане уявлення дозволяє легко вирішувати СЛАР виду  $Ax = b$  шляхом виконання прямого ходу для нижньотрикутної системи  $L_A y = b$  та зворотного ходу для верхньотрикутної системи  $U_A x = y$ . Проте алгоритм факторизації непридатний для СЛАР з розрідженими матрицями, оскільки веде до заповнення портрета, тобто появи в матрицях  $L_A$  і  $U_A$  ненульових елементів у тих позиціях, для яких  $a_{ij} = 0$  і, як наслідок, різкого збільшення обсягу пам'яті, необхідної для зберігання матриць [31].

Замість задачі знаходження факторизації (2.26) сформулюємо задачу для заданої матриці  $A$  представивши її у вигляді

$$A = LU + R \quad (2.27)$$

де матриці у правій частині задовольняють наступним властивостям:

– матриці  $L$  та  $U$  є нижньотрикутною та верхньотрикутною відповідно;

- $P_L \subset P_A$  і  $P_U \subset P_A$ ;
- $\forall (i, j) \in P_A : [LU]_{ij} = [A]_{ij}$ ;
- $P_A \cap P_R = \emptyset$

Тоді наближене представлення  $A \approx LU$  називається неповною LU-факторизацією матриці  $A$  або коротко її ILU-розкладом [5].

Останні дві вимоги означають, що на множині індексів  $P_A$  матричний добуток  $LU$  повинен точно відтворювати елементи  $A$ .

Для знаходження матриць  $L$  і  $U$  генеруватимемо їх рядково. Припустимо, що перші  $k - 1$  рядків вже знайдені і потрібно знайти  $k$ -й. Запишемо у блочному вигляді перші  $k$  рядків розкладу (2.27):

$$\begin{pmatrix} A_{11} & A_{12} \\ a_{21}^T & a_{22}^T \end{pmatrix} \begin{pmatrix} L_{11} & 0 \\ l_{21}^T & 1 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & u_{22}^T \end{pmatrix} + \begin{pmatrix} R_{11} & R_{12} \\ r_{21}^T & r_{22}^T \end{pmatrix} \quad (2.28)$$

де  $l_{21}^T$ ,  $u_{22}^T$ ,  $r_{21}^T$  и  $r_{22}^T$  — деякі вектори. Виконав дії над матрицями в правій частині (2.28), отримуємо

$$\begin{pmatrix} A_{11} & A_{12} \\ a_{21}^T & a_{22}^T \end{pmatrix} = \begin{pmatrix} L_{11}U_{11} + R_{11} & L_{11}U_{12} + R_{12} \\ l_{21}^T + U_{11} + r_{21}^T & l_{21}^T + U_{12} + u_{22}^T + r_{22}^T \end{pmatrix}$$

З рівності матриць випливає, що шукані вектори  $l_{21}^T$  і  $u_{22}^T$  повинні задовольняти умови:

$$l_{21}^T U_{11} + r_{21}^T = a_{21}^T \quad (2.29)$$

$$u_{22}^T + r_{22}^T = a_{22}^T - l_{21}^T U_{12} \quad (2.30)$$

Вирішивши ці системи, можна визначити коефіцієнти  $k$ -х рядків матриць розкладання  $l_{k1}, \dots, l_{k,k-1}, u_{kk}, \dots, u_{kn}$  [37].

Визначимо  $l_{kj}$  з (2.29) у припущенні, що  $l_{k1} \dots l_{kj-1}$  вже знайдено. Згідно з раніше сформульованими умовами, якщо  $a_{kj} = 0$ , то  $l_{kj} = 0$ . В іншому випадку  $r_{kj} = 0$  і (2.29) можна записати у вигляді

$$\sum_{i=1}^j l_{ki} u_{ij} = \sum_{i=1}^{j-1} l_{ki} u_{ij} + l_{kj} u_{jj} = a_{kj} \quad (2.31)$$

Це дозволяє обчислити  $l_{kj}$  наступним чином:

$$l_{kj} = \frac{1}{u_{jj}} \left( a_{kj} - \sum_{i=1}^{j-1} l_{ki} u_{ij} \right) \quad (2.32)$$

Аналогічними міркуваннями  $l_{jj} \equiv 1$  з урахуванням (2.30) можна отримати вираз для  $u_{kj}$ :

$$u_{kj} = a_{kj} - \sum_{i=1}^{k-1} l_{ki} u_{ij} \quad (2.33)$$

для тих випадків, коли  $a_{kj} \neq 0$ , інакше  $u_{kj} = 0$ .

Елементи матриць  $L$  і  $U$  отриманих в результаті ILU-розкладання, не збігаються з відповідними елементами матриць  $L_A$  і  $U_A$ , отриманих при повній LU-факторизації [31].

Матриця, отримана на вході ILU-розкладання, як правило, задовольняє всім трьом вимогам до матриці передумовника, сформульованим у § 2.4. Вона наближає матрицю  $A$ , оскільки на множині індексів  $P_A$  точно відтворює її; вона легко обчислюється на підставі формул (2.32) та (2.33) на рис. 2.3 представлено блок-схему алгоритму ILU-розкладання; вона так само легко оборотна, оскільки є добутком двох трикутних матриць [5].

Таким чином, використання  $M = LU$  є досить ефективним способом передумовлення.



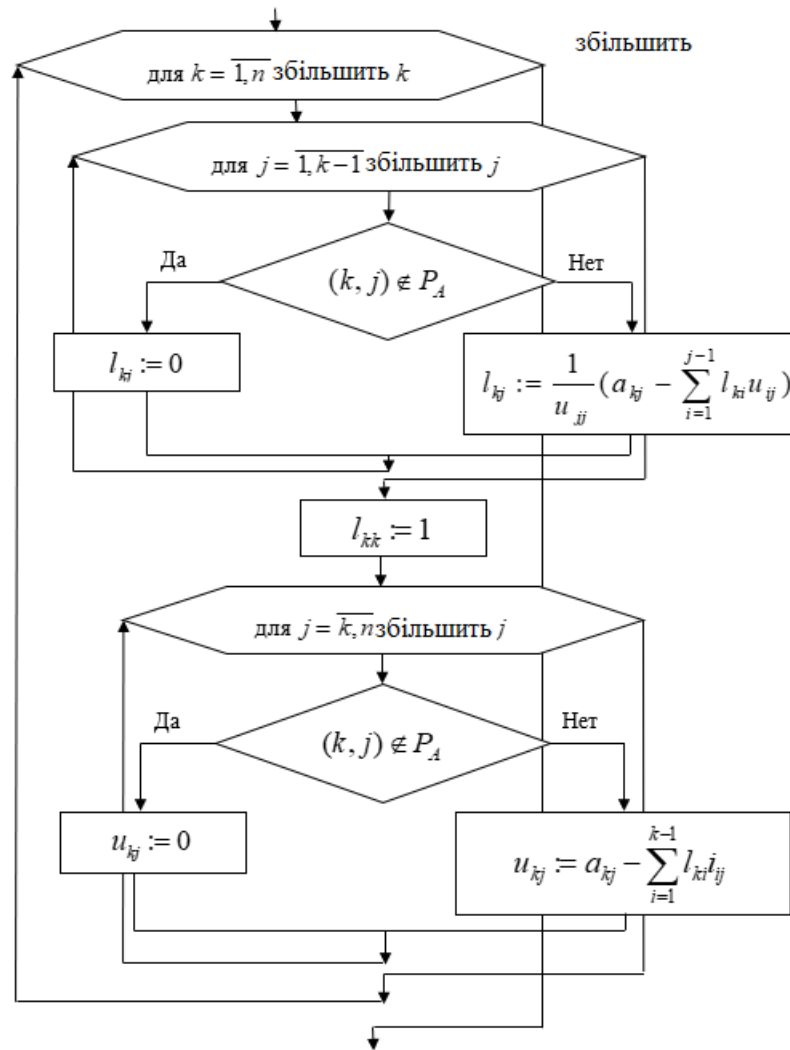


Рис. 2.3. Блок-схема алгоритму ILU-розкладу

У випадку, коли матриця  $A$  симетрична, з рівностей  $A = LU$  та  $A = A^T$  для трикутних матриць  $L$  і  $U$  випливає, що  $L = U^T$ . Факторизація (2.27) у цьому випадку набуває вигляду

$$A = LL^T + R = U^T U + R \quad (2.34)$$

В цьому випадку для знаходження діагональних елементів такої матриці  $L$  потрібні операції вилучення квадратного кореня. Щоб уникнути цього, замість факторизації (2.34) використовується дещо різний варіант

$$A = LDL^T + R$$

де головна діагональ  $L$  складається з одиничних елементів, а  $D$  діагональна матриця. Нижче на рис 2.4 наведена блок-схема алгоритму для

обчислення коефіцієнтів  $L$  і  $D$  із застосуванням логіки ILU-факторизації у разі, коли матриця симетрична.

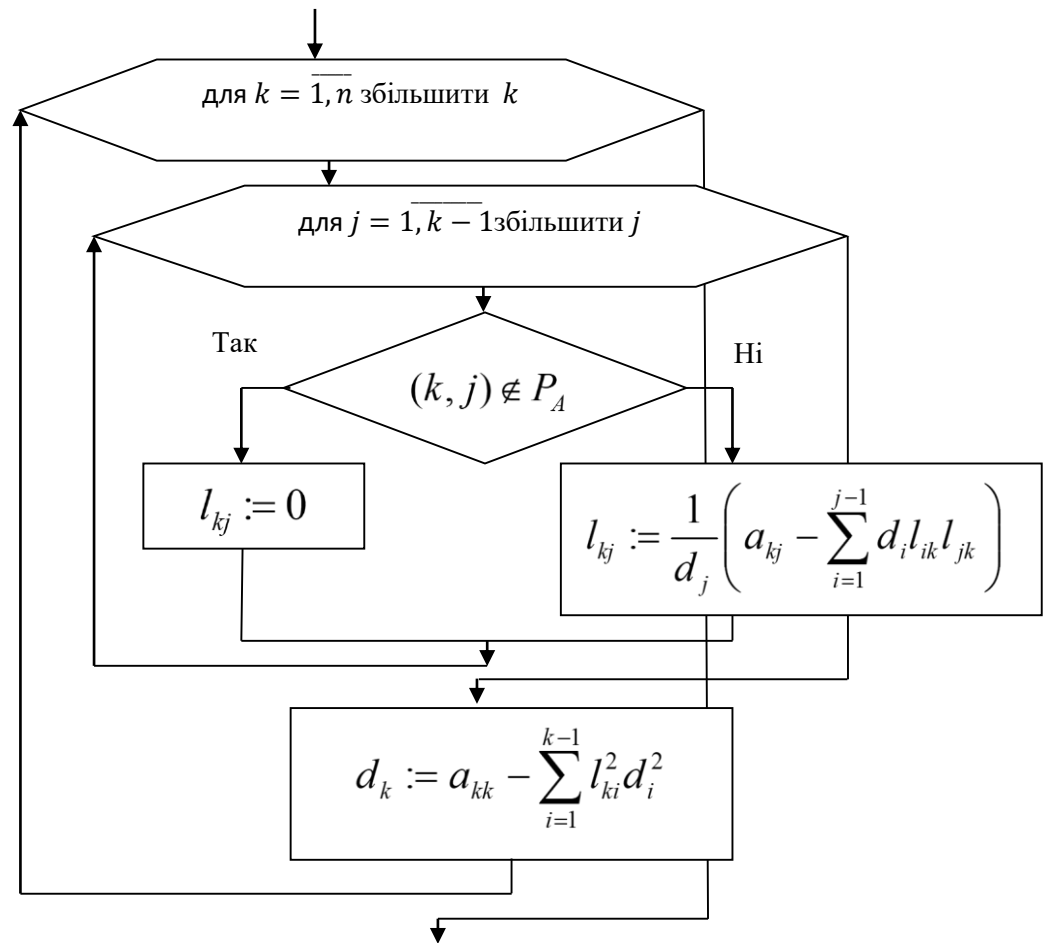


Рис. 2.4. Блок-схема ILU-розкладу для симетричної матриці

### Висновки до другого розділу

1. Наведено принципи побудови ітераційних процесів. Здійснено поділ ітераційних процесів на стаціонарні циклічні та нестаціонарні. Розглянуто принцип релаксації як із основних принципів побудови ітераційних процесів.
2. Розглянуто побудову проекційних методів. Наведено блок-схему в загальному вигляді, що описує алгоритм будь-якого методу проекційного класу.
3. Розглянуто проекційні методи підпростору Крилова. Наведено опис методів розв'язання спряжених градієнтів та біспряжених градієнтів, що належать до даного підпростору.

4. Визначено призначення передумови. Зформульовані вимоги необхідні для формування матриці, що зумовлює. Описано методи спряжених та біспряжених градієнтів з урахуванням застосування передумовлення.
5. Проаналізовано неповну LU-факторизацію матриці. Визначено, що отримана матриця вході ILU-розкладання задовольняє всім вимогам до матриці передумовника, будучи легко обчислюваною і оборотною, так як є добутком двох трикутних матриць. Використання неповної LU-факторизації є ефективним способом передумовлення.

## **РОЗДІЛ 3. РОЗРОБКА БІБЛІОТЕКИ КЛАСІВ ДЛЯ ПРЕДСТАВЛЕННЯ СЛАР**

### **3.1. Використання ООП підходу до подання СЛАР**

Дослідження різних методів розв'язання систем лінійних алгебраїчних рівнянь (СЛАР) полягає в порівняльному аналізі продуктивності методів при вирішенні СЛАР на ЕОМ. У цьому випадку оцінку ефективності даних методів доцільно проводити в рамках єдиного програмного комплексу – бібліотеки класів. Найлегше реалізувати подібний комплекс вдається за допомогою технології об'єктно-орієнтованого програмування (ООП), яка є методологією програмування, заснованою на представленні програми як сукупності взаємодіючих об'єктів, кожен із яких є екземпляром певного класу, а класи є членами певної ієрархії успадкування [8]. Технологія ООП дозволяє в рамках єдиної програми будувати функціонально однорідні елементи (в даному випадку алгоритми рішення СЛАР), які використовують загальні структури та механізми програмної реалізації. До цих структур слід віднести, перш за все, представлення в пам'яті ЕОМ матриць та векторів, які можуть бути реалізовані, проаналізувавши схеми зберігання матриць великої розмірності. До механізмів відносяться процедури визначення та введення вихідних даних, матрично-векторні операції лінійної алгебри, а також процедури контролю процесом виконання розрахунку. Всі загальні структури та механізми інкапсулюються і можуть бути об'єднані в базовий абстрактний клас рішення СЛАР, наслідуючи який далі здійснюється реалізація конкретного методу (алгоритму) рішення СЛАР. Використовуючи цей підхід, можна надати програмний комплекс рішення СЛАР у вигляді бібліотеки класів. Розроблена бібліотека дозволить виконати рішення заданої СЛАР будь-яким із наявних методів. Розрахунок може проводитися при заданні деяких початкових умов, таких як установки точності обчислення та кількості використовуваних ітерацій, вибору передумови та схеми зберігання матриці в пам'яті. Переважна більшість алгоритмів рішення СЛАР, у рамках представленої бібліотеки, будуть орієнтовані рішення несиметричних

матриць довільної структури. Такі матриці виходять під час реалізації різних чисельних алгоритмів. Використовуючи такий програмний комплекс, для заданого типу СЛАР можна підбирати найоптимальніший метод рішення, і навіть найефективніший режим розрахунку, як із погляду продуктивності, і економії пам'яті ЕОМ. Однією з основних переваг при використанні ООП підходу для представлення СЛАР є те, що досить просто в рамках бібліотеки, що розробляється, проводити модифікування наявних і розробку нових методів рішення СЛАР успадковуючи вже існуючі властивості та методи обробки матриці. Також зручним є використання поліморфних властивостей об'єктів, які є екземплярами класів методів рішення. У цьому випадку залежно від матриці об'єкт може приймати такий стан, у якому метод вирішення найефективніший.

### **3.2. Реалізація моделі бібліотеки класів**

Для того щоб розробити ефективний інструмент вирішення та дослідження систем лінійних алгебраїчних рівнянь у вигляді об'єктно-орієнтованої бібліотеки класів незалежної від інших компонентів і бібліотек крім як стандартних засобів мови програмування в даній магістерській роботі були виділені наступні завдання:

1. Реалізація структур даних для широкого класу матриць, які дозволяють ефективно зберігати та обробляти матрицю у пам'яті ЕОМ.
2. Реалізація проєкційних методів вирішення СЛАР підпростору Крилова на основі розроблених структур даних.

Для цього у вигляді аналізу предметної області було проаналізовано існуючі технології зберігання матриць великої розмірності та математичне представлення матричних алгоритмів. На наступному етапі розробки необхідно проведення об'єктно-орієнтованого аналізу бібліотеки класів, що розробляється.

Об'єктно-орієнтований аналіз – є метод аналізу, що досліджує вимоги до системи з точки зору класів та об'єктів [8]. У цьому випадку на основі

отриманої інформації в ході аналізу предметної області можна виділити такі сутності, що представлені на рис 3.1 як результат об'єктно-орієнтованого аналізу, що визначають вимоги до системи.

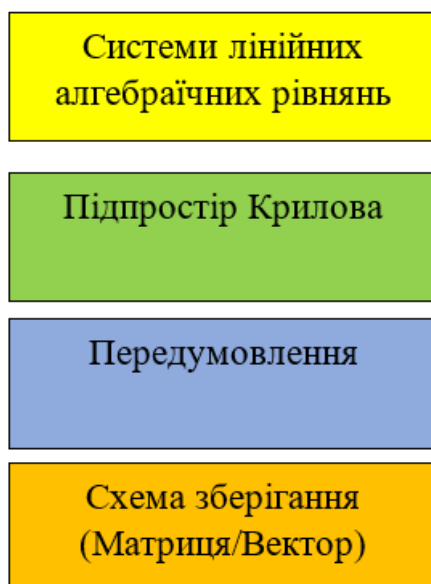


Рис. 3.1. Результат об'єктно-орієнтованого аналізу предметної області

На зображеному вище малюнку представлені основні визначення предметної області, які необхідно реалізувати у вигляді класів та екземпляри об'єктів, що дадуть необхідну функціональність бібліотеці, що розробляється.

Наступним кроком розробки об'єктно-орієнтованого програмного комплексу є реалізація його моделі, використовуючи метод об'єктно-орієнтованого проектування. Об'єктно-орієнтоване проектування є метод проектування, що поєднує процес об'єктно-орієнтованої декомпозиції та систему позначень для представлення логічної та фізичної моделей проектованої системи [8]. Декомпозиція здійснюється під час проектування складних систем програмного забезпечення. В цьому випадку проводять поділ системи на більш дрібні її частини, кожен з яких можна уточнювати незалежно один від одного.

Скориставшись результатами об'єктно-орієнтованого аналізу можна спроектувати наступну логічну модель бібліотеки, що розробляється, представлену на рис 3.2.

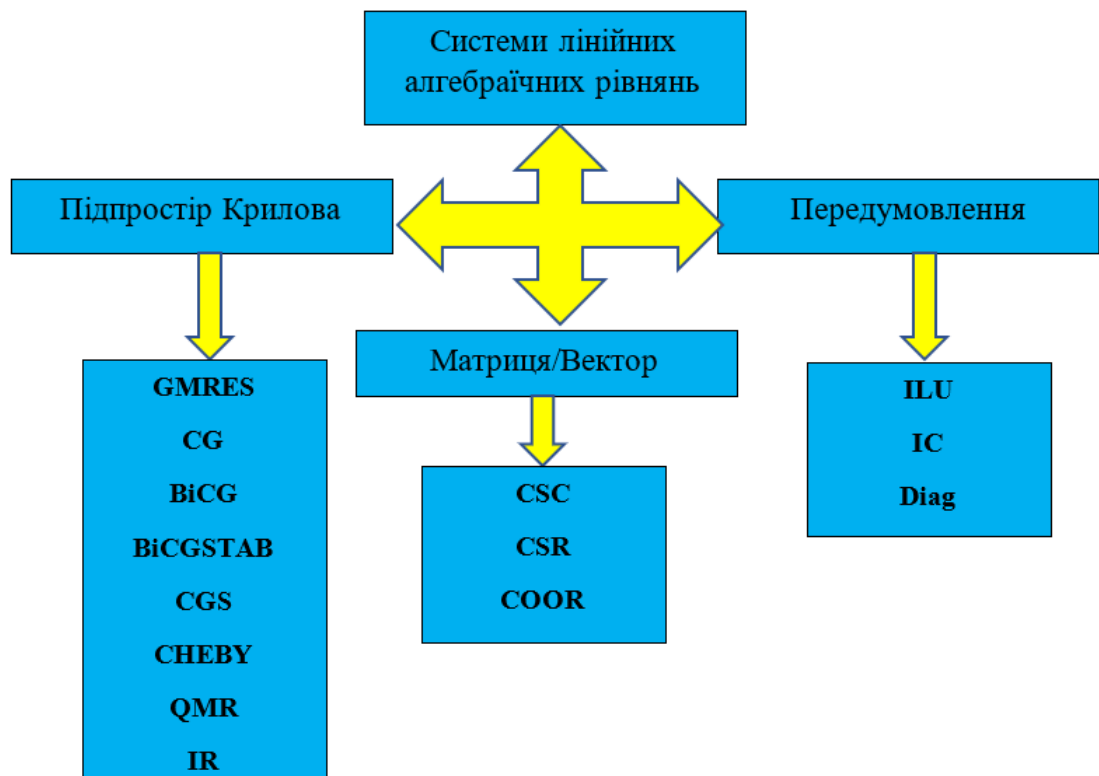


Рис. 3.2. Логічна модель бібліотеки, що розробляється

У представленій моделі було зроблено уточнення деяких визначень, описані методи вирішення СЛАР, що стосуються безлічі проекційних методів підпростору Крилова. Уточнено схеми зберігання матриць та алгоритми передумовлення, що використовуються.

На основі представленої логічної моделі реалізовано діаграму класів UML, яка зображена на рис. 3.3. Дана діаграма є структурною і описує архітектурну організацію та фізичну модель бібліотеки, що розробляється.

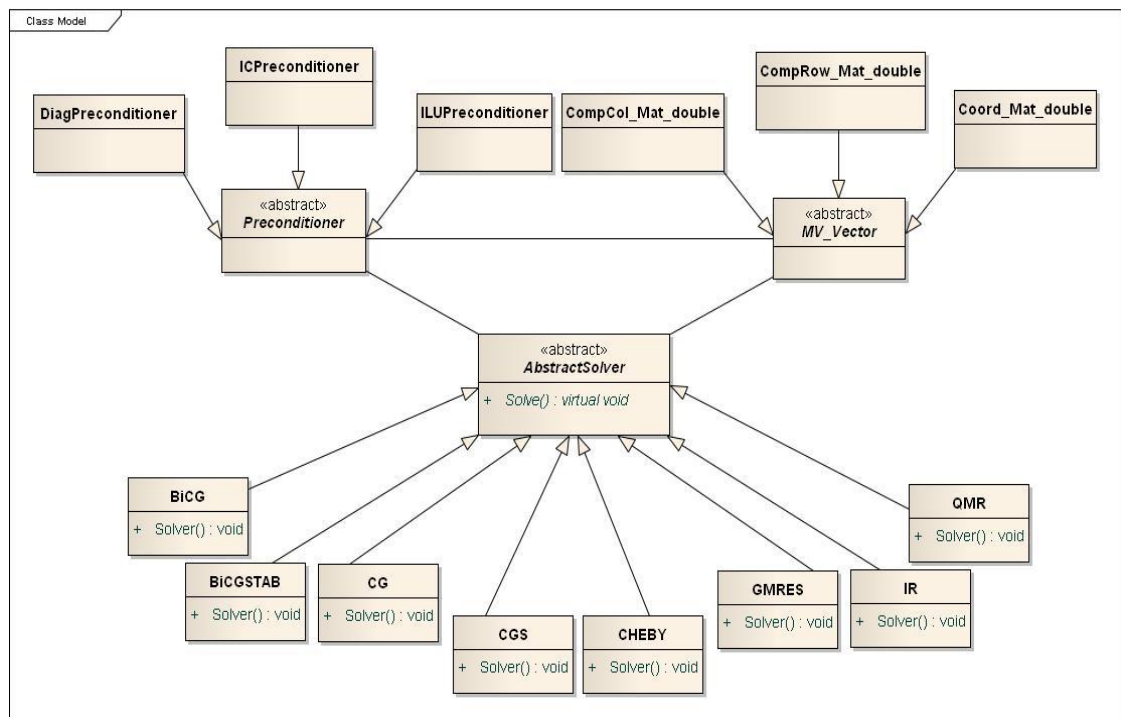


Рис. 3.3. Діаграма класів

У представленій діаграмі можна виділити три абстрактні базові класи, в яких описуються основні властивості визначень, виділених з об'єктно-орієнтованого аналізу предметної області. Наслідуючи ці базові класи, проводиться уточнення схем зберігання та методів обробки матриць для покращення спектральних характеристик, а також алгоритмів розв'язання СЛАР. Так, наприклад, клас `MV_Vector` є базовим класом, в якому визначається можливість зберігання вектора і деякі операції лінійної алгебри, що дозволяють робити дії над векторами. У свою чергу класи спадкоємці розширюють його можливості та дозволяють ефективно зберігати та обробляти як щільні, так і розріджені матриці відповідно до реалізованої в них схеми зберігання.

### 3.3. Програмна реалізація класів

На основі, наведених, на рис. 3.3 діаграми класів, що описує архітектурну організацію та фізичну модель розроблюваної бібліотеки, була розроблена бібліотека класів мовою програмування C++. Ядром бібліотеки є 14 класів, які використовуються для реалізації алгоритмів рішення або є методами вирішення СЛАР за визначенням, надаючи зручний інтерфейс



використання. Також бібліотека містить класи введення-виведення, що інкапсулюють роботу зі стандартними функціями мови програмування, що дозволяє спростити операції введення-виведення. Для коректної обробки виняткових ситуацій в ході рішення в бібліотеці реалізований клас, що дозволяє обробляти помилку, що робить розроблювані додатки з використанням даної бібліотеки більш стабільними.

Так для організації зберігання матриць у пам'яті комп'ютера була розроблена ієрархія у вигляді базового класу `MV_Vector`, про який говорилося раніше, та його спадкоємців:

1. клас `CompRow_Mat_double` – з реалізацією малої схеми зберігання матриці (CSR), у разі всі ненульові елементи матриці перераховуються по рядкам;
2. клас `CompCol_Mat_double` – з реалізацією стовпцевої схеми зберігання матриці (CSC), де ненульові елементи матриці перераховуються стовпцями;
3. клас `Coord_Mat_double` – з координатною схемою зберігання матриці (COOR), де всі ненульові елементи матриці перераховуються по рядкам і стовпцям з допомогою координат.

Базовий клас цієї ієрархії є шаблонним і складається з трьох полів шести конструкторів та двадцяти чотирьох членів-функцій. Нижче в таблиці 3.1 представлені поля класу та описано їх призначення.

Таблиця 3.1

**Поля класу `MV_Vector`**

Назва поля	Тип даних	Специфікатор доступу в класі	Опис та призначення
<code>p_</code>	TYPE *	protected	Вказівник на одновимірний масив, що містить елементи вектора

Назва поля	Тип даних	Специфікатор доступу в класі	Опис та призначення
dim_	int	protected	Розмір вектора
ref_	int	protected	Лічильник кількості посилань

Наслідуючи клас `MV_Vector` можна використовувати його конструктори, які дозволяють виділити необхідну кількість пам'яті та проініціалізувати її. Нижче у таблиці 3.2 представлені прототипи деяких конструкторів.

Таблиця 3.2

### Конструктори класу `MV_Vector`

Прототип	Опис
<code>MV_Vector()</code>	Конструктор за замовчуванням
<code>MV_Vector( int)</code>	Конструктор із встановленням розмірності вектора
<code>MV_Vector( int, const TYPE&amp;)</code>	Конструктор із встановленням розмірності та ініціалізації значенням параметра <code>TYPE</code>
<code>MV_Vector(TYPE*, int)</code>	Конструктор з ініціалізації значенням параметра <code>TYPE*</code> та встановленням розмірності
<code>MV_Vector(TYPE*,int, MV_Vector::ref_type i)</code>	Конструктор з ініціалізації значенням параметра <code>TYPE*</code> , встановлення розмірності та кількості посилань
<code>MV_Vector(const MV_Vector&lt;TYPE&gt;&amp;)</code>	Конструктор копіювання

У класі `MV_Vector` представлена реалізація двадцяти чотирьох членів функції серед них сім є перевантаженням операторів. Перевантаження операторів дозволило реалізувати основні операції лінійної алгебри. Ця можливість мови програмування C++ одна із її переваг. Так як

перевизначення математичних операцій «+», «-», «=» для розроблених програмістом класів дає можливість, змусити ці класи поводитися подібно до вбудованих типів і дає більше ступенів свободи в управлінні поведінкою алгоритму, що реалізується. У таблиці 3.3 подано основні функції-члени класу MV\_Vector.

Таблиця 3.3

### Основні методи класу MV\_Vector

Тип значення, що повертається	Прототип	Опис
TYPE&	operator()(int)	Перевантаження оператора виклику функції прямого доступу до елемента вектора
int	size() const	Повертає розмір вектора
int	ref() const	Повертає кількість посилань
MV_Vector<TYPE>&	newsize(int)	Встановлює новий розмір вектора
MV_Vector<TYPE>&	newsize(int, int)	Встановлює новий розмір вектора та ініціалізує його
MV_Vector<TYPE>&	operator=(const MV_Vector<TYPE>&);	Перевантаження оператора присвоєння. Надає значення одного вектора іншому.
MV_Vector<TYPE>	operator*(const TYPE &, const MV_Vector<TYPE> &)	Перевантаження оператора множення. Множить вектори

Тип значення, що повертається	Прототип	Опис
MV_Vector<TYPE>	operator+(const MV_Vector<TYPE> &, const MV_Vector<TYPE> &)	Перевантаження оператора додавання. Додавання векторів
MV_Vector<TYPE>	operator-(const MV_Vector<TYPE> &, const MV_Vector<TYPE> &)	Перевантаження оператора віднімання. Різниця векторів
TYPE	dot(const MV_Vector<TYPE> &, const MV_Vector<TYPE> &)	Скалярний добуток векторів
TYPE	norm(const MV_Vector<TYPE> &)	Евклідова норма

Класи спадкоємці – CompRow\_Mat\_double, CompCol\_Mat\_double, Coord\_Mat\_double складаються з п'яти полів, п'яти варіантів конструкторів та шістнадцяти членів-функцій. Функції-члени мають ідентичні імена і призначення, різниця полягає лише у схемі зберігання елементів матриці та як наслідок алгоритмів її обробки. Це спрощує освоєння бібліотеки користувачем та її використання. У таблиці 3.4 представлені поля спадкоємців класу MV\_Vector та описано їхнє призначення.

Таблиця 3.4

#### Поля спадкоємців класу MV\_Vector

Назва поля	Тип даних	Специфікатор доступу в класі	Опис та призначення
------------	-----------	------------------------------	---------------------

Назва поля	Тип даних	Специфікатор доступу в класі	Опис та призначення
val_;	MV_Vector <double>	private	Усі ненульові елементи матриці. Залежно від схеми зберігання вони перераховуються або в рядковому або в стовпцевому порядку
rowind_	MV_Vector <int>	private	Залежно від схеми зберігання вказує, в якому рядку знаходиться елемент або з якої позиції в масивах val_, colind_ починається і рядок матриці
colptr_	MV_Vector <int>	private	Залежно від схеми зберігання вказує, в якому стовпці знаходиться елемент або з якої позиції в масивах val_, rowind_ починається і рядок матриці
nz_	int	private	Кількість ненульових елементів
dim_[2]	int	private	Масив, що містить розмірність матриці

Конструктори класів – CompRow\_Mat\_double, CompCol\_Mat\_double, Coord\_Mat\_double також ідентичні. У таблиці 3.5 наведені прототипи деяких конструкторів класу CompRow\_Mat\_double.

Таблиця 3.5

#### Конструктори класу CompRow\_Mat\_double

Прототип	Опис
CompRow_Mat_double(const CompRow_Mat_double &)	Конструктор копіювання

Прототип	Опис
CompRow_Mat_double(const CompCol_Mat_double &)	Конструктор приймає як параметр посилання на об'єкт класу CompCol_Mat_double
CompRow_Mat_double(const Coord_Mat_double &)	Конструктор приймає як параметр посилання на об'єкт класу Coord_Mat_double

Функції-члени спадкоємців класів від MV\_Vector надають зручний інтерфейс доступу до елементів матриці її ініціалізації, а також дозволяють легко отримувати кількість не нульових елементів і розмірність матриці. Нижче в таблиці 3.6 представлені основні функції-члени спадкоємців класу MV\_Vector та описано їхнє призначення.

Таблиця 3.6

#### Прототипи методів спадкоємців класу MV\_Vector

Тип значення, що повертається	Прототип	Опис
double&	val(int)	Повертає елемент матриці, що зберігається в одновимірному масиві у вигляді вектора
int&	row_ind(int)	Повертає посилання на ціле значення, яке вказує з якої позиції в масивах val_, rowind_ починається i-й рядок матриці
int&	col_ptr(int)	Повертає посилання, яке вказує в якому стовпці знаходиться елемент

Тип значення, що повертається	Прототип	Опис
int	NumNonzeros() const	Повертає кількість ненульових елементів
CompCol_Mat_double&	operator=(const CompCol_Mat_double &)	Перевантаження оператора присвоєння. Привласнює одну матрицю іншою
CompCol_Mat_double&	newsize(int , int , int)	Встановлює нову розмірність матриці та ініціалізує її
MV_Vector<double>	trans_mult(const VECTOR_double & const	Транспонує матрицю та множить її на вектор

Оскільки швидкість збіжності ітераційних методів розв'язання СЛАР залежить від спектральних характеристик матриці, необхідне використання алгоритмів передумовлення. Для цього вихідна матриця множиться на матрицю передумови, яка покращує спектральні властивості, що збільшує швидкість збіжності. Тому, крім найітераційнішого методу, важливою частиною обчислювальної методики є передумова матриці. Використання передумови вносить додаткові обчислення на етапі побудови та застосування на кожній ітерації. Однак виграш у швидкості збіжності може бути значним. Тому в розробленій бібліотеці реалізовано ієрархію класів, що надає інтерфейс до алгоритмів передумовлення. Бібліотека містить такі передумови:

- 1) діагональна передумова Якобі – клас DiagPreconditioner є факторизацією головної діагоналі матриці;
- 2) передумова Холецького – клас ICPreconditioner у ньому реалізовано розкладанням з нульовим заповненням;

3) ILU-факторизація – клас ILUPreconditioner. Його призначення не повна LU факторизація матриці.

Реалізована ієрархія складається з базового абстрактного класу Preconditioner та трьох класів спадкоємців DiagPreconditioner, ICPreconditioner, ILUPreconditioner. На діаграмі класів UML ці елементи бібліотеки розташовані у верхній лівій частині (рис. 3.3).

Клас DiagPreconditioner містить алгоритм діагональної передумови Якобі і складається з одного поля двох конструкторів та двох членів функції. Нижче в таблиці 3.7 представлені конструктори класу DiagPreconditioner.

Таблиця 3.7

#### Конструктори класу DiagPreconditioner

Прототип	Опис
DiagPreconditioner (const CompCol_Mat_double &)	Конструктор, що приймає параметр у вигляді константного посилання на об'єкт класу CompCol_Mat_double, що містить вихідну матрицю
DiagPreconditioner (const CompRow_Mat_double &);	Конструктор, який приймає параметр у вигляді константного посилання на об'єкт класу CompRow_Mat_double, що містить вихідну матрицю

При виклику вище описаних конструкторів здійснюється пошук існування нульових елементів в діагоналі матриці, у разі якщо такий елемент був знайдений в конструкторі генерується виняткова ситуація і об'єкт передумовленого класу не створюється.

Базовий клас Preconditioner містить дві суто віртуальні функції solve() і trans\_solve() успадковуючи його класи, визначають реалізацію цих функцій для кожного конкретного методу передумовлення. Деякі з функцій, визначених у класі DiagPreconditioner, представлені в таблиці 3.8.

Таблиця 3.8

#### Прототипи методів класу DiagPreconditioner

Тип значення, що повертається	Прототип	Опис
MV_Vector<double>	solve(const VECTOR_double &)	Здійснює множення вектора на діагональ



	const;	матриці
double&	diag(int)	Повертає посилання діагональний елемент матриці

Клас `ICPreconditioner` містить алгоритм передумови Холецького складається з чотирьох полів двох конструкторів та двох членів функції. Нижче в таблиці 3.9 представлені поля класу `ICPreconditioner`.

Таблиця 3.9

#### Поля класу `ICPreconditioner`

Назва поля	Тип даних	Специфікатор доступу в класі	Опис та призначення
val_	MV_Vector <double>	private	Містить ненульові елементи матриці
indx_	MV_Vector <int>	private	Містить індекси трикутної матриці, що вказують, в якому стовпці знаходиться елемент
nz_	int	private	Кількість не нульових елементів
dim_[2]	int	private	Масив, що містить розмірність матриці

Для того щоб створити об'єкт передумовника в класі `ICPreconditioner` реалізовано два конструктори з параметрами, які дозволяють ініціалізувати об'єкт, що створюється, запустивши алгоритм передумовлення. Нижче в таблиці 3.10 представлені конструктори класу `ICPreconditioner`.

Таблиця 3.10

#### Конструктори класу `ICPreconditioner`

Прототип	Опис
<code>ICPreconditioner (const CompCol_Mat_double &amp;);</code>	Конструктор, що приймає параметр у вигляді константного посилання на об'єкт класу <code>CompCol_Mat_double</code> , що містить вихідну матрицю
<code>ICPreconditioner (const CompRow_Mat_double &amp;);</code>	Конструктор, який приймає параметр у вигляді константного посилання на об'єкт класу <code>CompRow_Mat_double</code> , що містить вихідну матрицю

Клас `ILUPreconditioner` що містить алгоритм не повної LU факторизації матриці складається з дев'яти полів двох конструкторів та двох членів

функції. У таблиці 3.11 представлені поля класу ILUPreconditioner і описано їх призначення.

Таблиця 3.11

**Поля класу ILUPreconditioner**

<b>Назва поля</b>	<b>Тип даних</b>	<b>Специфікатор доступу в класі</b>	<b>Опис та призначення</b>
<code>l_val_</code>	<code>MV_Vector&lt;double&gt;</code>	<code>private</code>	Містить елементи нижнього трикутника матриці
<code>u_val_</code>	<code>MV_Vector&lt;double&gt;</code>	<code>private</code>	Містить елементи верхнього трикутника матриці
<code>l_colptr_</code>	<code>MV_Vector&lt;int&gt;</code>	<code>private</code>	Вказує, з якої позиції в масивах <code>l_val_</code> , <code>l_rowind_</code> починається <i>i</i> -й рядок матриці
<code>u_colptr_</code>	<code>MV_Vector&lt;int&gt;</code>	<code>private</code>	Вказує, з якої позиції в масивах <code>u_val_</code> , <code>u_rowind_</code> починається <i>i</i> -й рядок матриці.
<code>l_rowind_</code>	<code>MV_Vector&lt;int&gt;</code>	<code>private</code>	Вказує, у якому стовпці нижньотрикутної матриць знаходиться елемент
<code>u_rowind_</code>	<code>MV_Vector&lt;int&gt;</code>	<code>private</code>	Вказує, у якому стовпці верхньотрикутної матриць знаходиться елемент
<code>l_nz_</code>	<code>int</code>	<code>private</code>	Кількість ненульових елементів у нижньотрикутній матриці
<code>u_nz_</code>	<code>int</code>	<code>private</code>	Кількість ненульових елементів у верхньотрикутній матриці
<code>dim_[2]</code>	<code>int</code>	<code>private</code>	Загальна розмірність матриці

Конструктори класу ILUPreconditioner подібні конструкторам класів ICPreconditioner і DiagPreconditioner вони приймають у вигляді параметрів константне посилання на об'єкт класу `CompCol_Mat_double` чи `CompRow_Mat_double` що містить вихідну матрицю. Результатом є об'єкт передумови класу, що містить матрицю передумов. Дотримання такої послідовності в розробці компонентів бібліотеки визначає її інтуїтивне використання, освоївши роботу з одним класом, користувач досить легко зможе використовувати й інші.

Основне призначення бібліотеки класів є рішення систем лінійних рівнянь алгебри, у зв'язку з цим у бібліотеці реалізована ієрархія класів, які містять алгоритми ітераційних методів рішення СЛАР і надають зручний інтерфейс використання. Основна перевага ітераційних методів перед прямими методами рішення полягає у мінімальних вимогах до пам'яті для зберігання матриць, а також у тому, що ітераційні методи замість матрично-матричних операцій множення використовують матрично-векторні та працюють з результуючими векторами [16]. Найефективнішою групою ітераційних методів, застосовуваної нині у лінійній алгебрі для вирішення СЛАР, є проєкційні методи підпростору Крилова. У розробленій бібліотеці представлено вісім класів з реалізацією алгоритмів рішення, що належать до цієї групи, нижче представлений список цих класів:

- 1) GMRES - узагальнений метод мінімальних нев'язок;
- 2) CG – спосіб сполучених градієнтів;
- 3) BiCG – метод біспряжених градієнтів;
- 4) CGS – квадратичний спосіб сполучених градієнтів;
- 5) BiCGSTAB – стабілізований метод біспряжених градієнтів;
- 6) CHEBY – метод Чебишева;
- 7) QMR - метод квазімінімальних нев'язок;
- 8) IR – метод Річардсона.

Клас `AbstractSolve` є абстрактним базовим класом для всіх вище перерахованих класів, його особливістю є наявність суто віртуальної функції `Solve()`, реалізація якої визначається у всіх класах спадкоємців. Виклик цієї функції дозволяє отримувати рішення СЛАР відповідним методом рішення в залежності від класу екземпляром, якого є об'єкт, що викликає функцію. Нижче в таблиці 3.12 представлені основні функції-члени класу `AbstractSolve` та описано їх призначення.

Таблиця 3.12

### Прототипи методів класу `AbstractSolve`

Тип значення, що повертається	Прототип	Опис
void	initMatrix( CompCol_Mat_double&) / initMatrix( CompRow_Mat_double&)	Ініціалізація матриці. Функція має перевантажений варіант для двох класів CompCol_Mat_double та CompRow_Mat_double.
int	GetSteps();	Повертає кількість ітерацій витрачених отримання рішення.
bool	SetPrecision(const double& val);	Встановлення точності обчислення.

Для організації введення-виводу до розробленої бібліотеки були додані класи IOHarwellBoeing та IOMatrixMarket з підтримкою читання та запису у специфікації файлів Matrix Market та Harwell-Boeing.

Формат Harwell-Boeing є поширеною специфікацією файлу спеціально спроектованого для зберігання розріджених матриць в текстовому файлі. Дані подаються у 80-колонках, фіксованої довжини. Кожна матриця починається з декількох блоків рядка заголовка, за яким слідує два, три, або чотири блоки даних. Заголовок блоку містить зведену інформацію про формат зберігання та раціональне використання оперативної пам'яті. З заголовка блоку користувач може визначити скільки пам'яті потрібно для зберігання матриці.

Специфікація Matrix Market дозволяє забезпечити простий механізм зберігання та обміну елементами матриці. Формат поділяється на колекцію дочірніх форматів, що відрізняються методом представлення даних. У початковій специфікації визначено два матричні формати:

1) Формат координат, що підходить для представлення загальних розріджених матриць. Файл містить лише ненульові елементи та координати кожного ненульового елемента.

2) Формат масиву, який підходить для представлення загальних щільних матриць. Усі записи представлені у колонко-орієнтованому порядку.

### 3.4. Дослідження ефективності методів вирішення представлених у розробленій бібліотеці

Як було сказано ранні, одним із способів дослідження різних методів вирішення систем лінійних алгебраїчних рівнянь полягає в порівняльному аналізі продуктивності алгоритмів рішення. Таким чином, основна мета дослідження полягає в порівнянні часу виконання проєкційних методів рішення з різними передумовами і матрицею, що зберігається за допомогою рядкової (CSR) та стовпцевої (CSC) схем зберігання на основі розробленої та вище представленої бібліотеки класів.

Для експерименту було обрано дві матриці зі збірки Sparse Matrix Collection [65]. Вибрані матриці є несиметричними та містять лише раціональні числа. У таблиці 3.13 подано деякі характеристики вибраних матриць.

Таблиця 3.13

#### Характеристики матриць

Найменування	Розмірність матриці	Загальна кількість елементів	Елементів на діагоналі	Елементів під діагоналлю	Елементів над діагоналлю
Матриця А	17758 x 17758	99147	17758	41534	39855
Матриця Б	5005 x 5005	20033	5005	7514	7514

Графічне уявлення матриць що у цьому дослідженні представлено на рис. 3.4. Це подання забезпечує візуальне відображення розрідженості структури матриці.

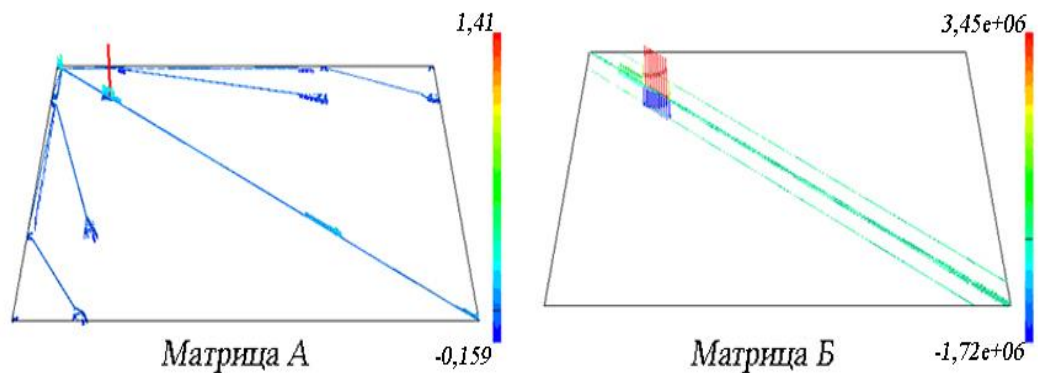


Рис. 3.4. Структура вибраних матриць [65]

У додатку Г представлені дані отримані в ході дослідження, які є часовими витратами і використаною кількістю ітерацій для отримання рішення СЛАР. Відомості було отримано під час рішення з точністю  $1e-06$ . У разі відсутності збіжності, виконувалась максимальна кількість ітерацій, що дорівнює 3000. На основі отриманих даних були побудовані гістограми, представлені нижче (рис 3.5, 3.6), що описують найбільш ефективні випадки рішення для матриць, що беруть участь у дослідженні.



Рис. 3.5 Найбільш ефективні випадки розв'язання матриці А

У першому випадку при вирішенні матриці розмірністю  $17758 \times 17758$  найшвидше рішення надав метод узагальнених мінімальних нев'язок спільно з діагональною передумовою Якобі. Рядкова і стовпцева схеми зберігання в більшості випадків були однаково ефективні за винятком квадратичного методу пов'язаних градієнтів і методу квазімінімальних нев'язок, які спільно з передумовою Холецького використовуючи малу схему зберігання матриці, досягли рішення за меншу кількість ітерацій, ніж при використанні.

У другому випадку при вирішенні матриці розмірністю  $5005 \times 5005$  найбільш швидке рішення було отримано методом спряжених градієнтів з діагональною передумовою Якобі. Аналізуючи вплив схем зберігання, були отримані аналогічні результати що дозволяють зробити висновок про однакову ефективність малої та стовпцевої схем зберігання.



Рис. 3.6. Найбільш ефективні випадки вирішення матриці Б

Це пояснюється тим, що матрично-векторне множення найбільше просто реалізується саме для CSR, коли матриця зберігається по рядках - проводиться послідовний перебір елементів матриці, які множаться на коефіцієнт вектора.

Аналізуючи ефективність передумов, що використовуються, можна виділити діагональну передумову Якобі, яка в більшості випадків дозволила виконати рішення до заданого ступеня точності, витративши найменшу кількість ітерацій. Розглядаючи методи рішення, у першому випадку найбільш ефективним виявився узагальнений метод мінімальних нев'язок у другому метод спряжених градієнтів. Слід виділити методи біспряжених градієнтів і квадратичний метод спряжених градієнтів, які в загальному випадку для матриці А і Б показали найкращий результат у поєднанні з усіма передумовами, що використовуються. Також слід додати, що задану точність обчислення ( $1e-06$ ) було досягнуто в 55 випадках з 96 можливих при заданій максимальній кількості ітерацій.

### **Висновки до третього розділу**

1. Визначено основні механізми об'єктно-орієнтованого програмування та їх застосованість у поданні СЛАР.
2. Виділені основні визначення в ході аналізу предметної області представлені у вигляді результатів об'єктно-орієнтованого аналізу, що визначає вимоги до системи у вигляді класів та об'єктів.
3. Використовуючи методи об'єктно-орієнтованого проектування та результати об'єктно-орієнтованого аналізу, було побудовано структурну UML діаграму класів, яка описує архітектурну організацію та фізичну модель бібліотеки класів, що розробляється.
4. На основі спроектованої UML діаграми розроблено об'єктно-орієнтовану бібліотеку класів мовою програмування C++, що дозволяє отримувати рішення СЛАР з матрицею загального вигляду з нерегулярною структурою, використовуючи методи підпростору Крилова з передумовою.
5. Проведено дослідження ефективності методів розв'язання СЛАР реалізованих у бібліотеці класів, дані дослідження представлені у додатку Г.



## ВИСНОВКИ

Основною метою даної магістерської роботи було дослідження та розробка структур даних, у яких реалізовано ефективні методи зберігання та обробки матриць у пам'яті комп'ютера, а також реалізація алгоритмів рішення СЛАР на їх основі. Формулювання мети обумовлено розвитком обчислювальної техніки та процесом переходу до більш складних тривимірних, у довільних геометричних областях моделей у вигляді систем диференціальних рівнянь у приватних похідних та їх дискретних аналогів на неструктурованих сітках. Як наслідок цей процес призвів до необхідності розв'язання великих розріджених систем лінійних алгебраїчних рівнянь з матрицями нерегулярної структури. У цьому основним обмеженням щодо обчислень є оперативна пам'ять ЕОМ. У зв'язку з цим було визначено завдання рішенням, якого було оптимізоване подання матриці пам'яті комп'ютера.

Аналіз предметної області дозволив визначити основні критерії, які дають змогу ефективно зберігати та обробляти матриці великої розмірності. Такими критеріями є:

- зберігання тільки ненульових елементів матриці;
- оперування тільки з ненульовими елементами;
- збереження розрідженості матриці.

У зв'язку з цим ефективне подання матриці в пам'яті спирається на взаємозв'язок трьох основних складових: даної матриці, даного алгоритму та обчислювальної машини. Таке визначення має евристичний характер: матриця розріджена, якщо має сенс отримувати вигоду з наявності в ній багатьох нулів. Будь-яку розріджену матрицю можна обробляти так, ніби вона була щільною: навпаки, будь-яку щільну, або заповнену, матрицю можна обробляти алгоритмами для розріджених матриць. В обох випадках буде отримано правильні чисельні результати, але обчислювальні витрати зростуть.

Аналіз класичних структур даних, таких як масивів, списків, стеків та черг дозволив виділити різні схеми зберігання матриць на їх основі. Було визначено, що ефективність застосування різних схем зберігання залежить від операцій, які передбачаються в алгоритмі обробки. Також виявлено, що проблеми реалізації і накладна пам'ять зростають паралельно з ускладненням схеми зберігання. Високо складні схеми вимагають професійної програмної реалізації, інакше їх потенційні переваги будуть втрачені.

Основою алгоритмічної частини проекту послужили принципи побудови ітераційних процесів. Було здійснено поділ ітераційних процесів на стаціонарні циклічні та нестаціонарні. Як наслідок розглянуто принцип релаксації як із основних принципів побудови ітераційних процесів. Також було проаналізовано побудову проекційних методів і як підмножина цього класу проекційні методи підпростору Крилова, оскільки методи цієї групи є найефективнішими ітераційними методами, які нині застосовують у лінійній алгебрі для вирішення СЛАР. У зв'язку з тим, що швидкість збіжності ітераційних методів залежить від спектральних характеристик матриці збільшення швидкості збіжності необхідно використання передумовлення [5]. Для поліпшення спектральних властивостей вихідна матриця множиться на матрицю передумови, що збільшує швидкість збіжності. Тому були сформульовані вимоги, необхідні для формування передумовливої матриці. Розглянуто неповну LU-факторизацію матриці. Визначено, що отримана матриця на вході ILU-розкладання задовольняє всім вимогам до матриці передумовника, будучи легко обчислюваною і оборотною, так як являє собою добуток двох трикутних матриць.

Перед тим як розпочати розробку структури та програмних модулів бібліотеки класів мовою C++ для вирішення СЛАР було визначено основні механізми об'єктно-орієнтованого програмування та їх застосовність у поданні СЛАР. Дана парадигма дозволяє в рамках єдиного застосування будувати функціонально однорідні елементи, що подаються у вигляді

алгоритмів рішення СЛАР, які використовують загальні структури та механізми програмної реалізації.

Для того щоб спроектувати модель бібліотеки класів, що розробляється, виділені основні визначення в ході аналізу предметної області були представлені у вигляді результатів об'єктно-орієнтованого аналізу, що дозволило визначити вимоги до системи у вигляді класів та об'єктів. Використовуючи методи об'єктно-орієнтованого проектування та результати об'єктно-орієнтованого аналізу, було побудовано структурну UML діаграму класів, яка описує архітектурну організацію та фізичну модель бібліотеки, що розробляється. На наступному етапі розробки було здійснено програмну реалізацію бібліотеки у вигляді ієрархії класів представленої в UML діаграмі (рис. 3.3). Таким чином було розроблено бібліотеку класів ітераційних методів підпросторів Крилова з передумовою для вирішення СЛАР. Орієнтована на матриці загального виду з нерегулярною структурою як симетричних так і не симетричних структур даних для широкого класу матриць, які дозволяють ефективно зберігати і обробляти матрицю в пам'яті.

Одним із завдань, поставленим при реалізації магістерської роботи було дослідження ефективності проєкційних методів рішення у вигляді порівняння часу виконання з різними передумовами використовуючи реалізовані структури даних для зберігання широкого класу матриць. Дане дослідження було проведено, використовуючи розроблену бібліотеку класів. Результати дослідження виявили найефективніші методи рішення для досліджуваних матриць, і навіть дозволив визначити найбільш застосовні методи які не залежать від конкретної структури матриці. У дослідженні були використані рядкова та стовпцева схема зберігання матриць. Аналізуючи їх вплив на рішення СЛАР з допомогою отриманих даних можна дійти висновку, що дані схеми зберігання рівно ефективні, оскільки отримані рішення не відрізняються витраченою кількістю ітерацій з незначним переважанням малої схеми зберігання за часом виконання. Це пояснюється тим, що матрично-векторне множення найбільше просто

реалізується для малої схеми зберігання, коли матриця зберігається по рядках – проводиться послідовний перебір елементів матриці, які множаться на коефіцієнт вектора.

Таким чином, у даній магістерській роботі реалізовано всі поставлені цілі та завдання, підтверджено гіпотезу, яка передбачала ефективне використання об'єктно-орієнтованих бібліотек класів для вирішення систем лінійних алгебраїчних рівнянь.

При подальшому розвитку даного проекту необхідна реалізація паралельних алгоритмів рішення з ефективними способами розподілу процесорів оброблюваної матриці.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Абаффі Й. Математичні методи для лінійних та нелінійних рівнянь: Проекційні ABS-алгоритми / Й. Абаффі, Е. Спедикато; пров. з англ. А. Я. Белянкова, О. П. Бурдакова. - М.: Світ, 1996. - 268 с.
2. Амосов А. А. Обчислювальні методи / А. А. Амосов, Ю. А. Дубинський, Н. В. Копченова. - 3-тє вид., Перероб. та дод. - М.: Видавничий дім MEI, 2008. - 672 с.
3. Антонов А. С. Паралельне програмування з використанням технології MPI/А. С. Антонов. - М.: Вид-во МДУ, 2004. - 71 с.
4. Баканов В. М., Введення в практику розробки паралельних програм у стандарті MPI / В. М. Баканов, Д. В. Осипов. - М.: МДАПД, 2005. - 63 с.
5. Баландін М. Ю. Методи вирішення СЛАР великої розмірності. / М. Ю. Баландін, Е. П. Шуріна. - Новосибірськ: Вид-во НДТУ, 2000. - 70 с.
6. Бартеньєв О. В. Фортран для студентів / О. В. Бартеньєв. - М.: Діалог-МІФІ, 2006. - 397 с.
7. Бахвалов Н. С. Чисельні методи / Н. С. Бахвалов. - М.: Наука, 1975. - 468 с.
8. Буч Граді Об'єктно-орієнтований аналіз та проектування з прикладами додатків / Граді Буч. - 3 вид. ; пров з англ. - М.: Вільямс, 2008. - 720 с.
9. Вандервуд Д. Шаблони C++. Довідник розробника / Д. Вандервуд, Ніколаї М. Джосаттіс; пров. з англ. - М.: Вільямс, 2008. - 544 с.
10. Вержбицький В. М. Обчислювальна лінійна алгебра / В. М. Вержбицький - М.: Вища школа, 2009. - 351 с.
11. Вержбицький В. М. Чисельні методи (лінійна алгебра та нелінійні рівняння) / В. М. Вержбицький. - 2-ге вид., Випр. - М.: ОНІКС 21 століття, 2005. - 432 с.

- 12.Віленкін Н. Я. Методи послідовних наближень / Н. Я. Віленкін - М.: Наука, 1963. - 108 с.
- 13.Воєводін В. В. Матриці та обчислення / В. В. Воєводін, Ю. А. Кузнєцов. - М.: Наука, 1984. - 320 с.
- 14.Воєводін В. В. Обчислювальна математика та структура алгоритмів / В. В. Воєводін. - М.: Вид-во МДУ, 2006. - 112 с.
- 15.Воєводін В. В. Математичні моделі та методи в паралельних процесах / В. В. Воєводін. - М.: Наука, 1986. - 296 с.
- 16.Воєводін В. В. Паралельні обчислення / В. В. Воєводін, Вл. В. Воєводін. - СПб. : БХВ-Петербург, 2002. - 608 с.
- 17.Воєводін Вл. В. Чисельні методи, паралельні обчислення та інформаційні технології / Вл. В. Воєводін. - М.: Видавництво московського Університету, 2008. - 320 с.
- 18.Воробйов Г. Н. Практикум з чисельних методів. / Г. Н. Воробйов, А. Н. Данилова. - М.: Вищ. шк., 2007. - 184 с.
- 19.Гамма Еге. Прийоми об'єктно-орієнтованого проектування. Патерни проектування / Е. Гамма, Р. Хелм, Р. Джонсон, Д. Вільямс; пров. з англ. - СПб. : Пітер, 2011. - 368 с.
- 20.Годунов С. К. Рішення систем лінійних рівнянь / С. К. Годунов. - Новосибірськ: Наука, 1980. - 250 с.
- 21.Голуб Дж. Матричні обчислення / Дж. Голуб, Лоун Ч. Ван; пров. з англ. Ю. М. Нечепуренко, А. Ю. Романова. - М.: Світ, 1999. - 548 с.
- 22.Грибан В. Г. Охорона праці: навч. посібник. (для студ. вищ. навч. закл.) / В. Г. Грибан, О. В. Негодченко – К. : Центр учбової літератури, 2009. – 280 с.
- 23.Давидов В. Розробка Windows-додатків за допомогою MFC та API функцій / В. Давидов. - СПб. : Пітер, 2008. - 587 с.
- 24.Демидович Б. П. Чисельні методи аналізу. Наближення функцій, диференціальні та інтегральні рівняння: навч. посібник / Б. П. Демидович, І. А. Марон, Е. З. Шувалова. - М.: Наука, 2008. - 400 с.

- 25.Деммель Дж. Обчислювальна лінійна алгебра. Теорія та додатки / Дж. Деммель. - М.: Світ, 2001. - 430 с.
- 26.Джордж А. Чисельне рішення великих розріджених систем рівнянь / А. Джордж, Дж. Лю; пров. з англ. Х. Д. Ікрамов. - М.: Світ, 1984. - 333 с.
- 27.Жигульська В.Ю. Чисельні методи/В.Ю. Жигульська. - Луганськ: Альма-матер, 2005. - 137 с.
- 28.Жидацький В. Ц. Практикум із охорони праці / В. Ц. Жидацький, В. С. Джигірей, В. С. Сторожук та ін. – Львів: Афіша, 2000. – 352 с.
- 29.Залізняк В. Є. Основи наукових обчислень. Введення в чисельні методи для фізиків та інженерів/В. Є. Залізняк. - М.: Інститут комбінованих досліджень, 2006. - 263 с.
- 30.Ікрамов Х. Д. Чисельне рішення матричних рівнянь / Х. Д. Ікрамов. - М.: Наука, 1984. - 192 с.
- 31.Ільїн В.П. Методи неповної факторизації на вирішення алгебраїчних систем / В.П. Ільїн. - М.: Фізматліт, 1995 - 450 с.
- 32.Корнєєв В. Д. Паралельне програмування в МРІ / В. Д. Корнєєв. - 2-ге вид., Випр. - Новосибірськ: Вид-во ІВМіМГ З РАН, 2002. - 215 с.
- 33.Костомаров Д. П. Програмування та чисельні методи / Д. П. Костомаров, Л. С. Корухова, С. Г. Манжелей. - М.: Видавництво МДУ, 2001. - 578 с.
- 34.Ланцош К. Практичні методи прикладного аналізу/К. Ланцош. ; пров. англ. М. З. Кайнер. - М.: Державне видавництво фізико-математичної літератури, 1961 - 524 с.
- 35.Лафор Р. Об'єктно-орієнтоване програмування в С++ / Р. Лафор. - 4-те вид. ; пров. з англ. - СПб. : Пітер, 2004. - 1040 с.
- 36.Мак-Кракен Д. Чисельні методи та програмування на фортані / Мак-Кракен Д., У. Дорн. - М.: Світ, 1977. - 579 с.
- 37.Маргуліс Б. Є. Системи лінійних рівнянь / Б. Є. Маргуліс. - М.: Державне видавництво фізико-математичної літератури, 1960 - 97 с.

- 38.Марчук Г. І. Методи обчислювальної математики/Г. І. Марчук. - М.: Наука, 1977. - 456 с.
- 39.Мюсер Девід. Р. С++ та STL довідкове керівництво / Девід Р. Мюссер, Жілмер Дж. Дердж, Атул Сейні. - 2-ге вид. ; пров з англ. - М.: Вільявс, 2010. - 412 с.
- 40.Ніколас. А. С++ для професіоналів / А. Ніколас, Скот Дж. Клепер. ; пров. з англ. - М.: Вільявс, 2006. - 912 с.
- 41.Ортега Дж. Введення в паралельні та векторні методи вирішення лінійних систем / Дж. Ортега; пров. з англ. Х. Д. Ікрамов, І. Є. Капоріна. - М.: Світ, 1991 - 367 с.
- 42.Ортега Дж. Ітераційні методи вирішення нелінійних систем рівнянь з багатьма невідомими / Дж. Ортега, В. Рейнболдт; пров. англ. Е. В. Вершкова, Н. П. Вершкова. - М.: Світ, 1975. - 560 с.
- 43.Островський А. М. Рішення рівнянь та систем рівнянь / А. М. Островський. - М.: Видавництво іноземної літератури, 1963. - 214 с.
- 44.Павлова С.П. Охорона праці радіо- та електронної промисловості / З. П. Павлова. - М.: Енергія, 1979. - 321 с.
- 45.Пісанецькі С. Технологія розріджених матриць / С. Пісанецькі; пров. з англ. Х. Д. Ікрамов, І. Є. Капоріна. - М.: Світ, 1988. - 410 с.
- 46.Подбельський В. В. Мова С++: навч. посібник/В. В. Подбельський. - М.: Фінанси та статистика, 1995. - 560 с.
- 47.Самарський А. А. Введення в чисельні методи: навч. посібник для вузів / А. А. Самарський. - 3-тє вид., Стер. - СПб: Лань, 2005. - 288 с.
- 48.Самарський А. А. Чисельні методи / А. А. Самарський, А. В. Гулін. - М.: Наука, 1989. - 432 с.
- 49.Кушнір Л. А. Системи лінійних рівнянь / Л. А. Кушнір. - М.: Наука, 1986. - 64 с.
- 50.Соболь І. М. Чисельні методи Монте-Карло / І. М. Соболь. - М.: Наука, 1973. - 308 с.



51. Страуструп Б. Програмування: принципи та практика використання C++ / Б. Страуструп; пров. з англ. - М.: Вільявс, 2006. - 1248 с.
52. Страуструп Б. Мова програмування C++ / Б. Страуструп; пров. з англ. - М.: Біном, 2011. - 1136 с.
53. Стрег Г. Лінійна алгебра та її застосування / Г. Стрег. - М.: Світ, 1980. - 446 с.
54. Тьюарсон Р. Розріджені матриці / Р. Тьюарсон; пров. з англ. Е. М. Пейсахович. - М.: Світ, 1977. - 171 с.
55. Вільям Т. Структури даних у C++ / Т. Вільям, Ф. Вільям; пров. з англ. - М.: Біном-Прес, 2006. - 816 с.
56. Уоткінс Д. С. Основи матричних обчислень / Д. С. Уоткінс. - М.: БІНОМ, 2006. - 664 с.
57. Фаддєєв Д. К. Обчислювальні методи лінійної алгебри / Д. К. Фаддєєв, В. Н. Фаддєєва - М.: Наука, 1996. - 645 с.
58. Форсайт Дж. Чисельне рішення систем лінійних рівнянь алгебри / Дж. Форсайт, К. Молер ; пров. англ. В. П. Ільїна, Ю. І. Кузнецова - М.: Світ, 1969. - 163 с.
59. Хеммінг Р. В. Чисельні методи для науковців та інженерів / Р. В. Хеммінг; пров. англ. В. Л. Арлазарова - М.: Наука, 1972. - 399 с.
60. Холзнер С. Visual C++ 6: навчальний курс / С. Холзнер. - СПб: Пітер, 2007. - 789 с.
61. Хортон А. Visual C++ 2005: базовий курс. / А. Хортон; пров. з англ. - М.: Вільявс, 2007. - 1152 с.
62. Шілд Г. Повний довідник по C++ / Г. Шілд. ; пров. з англ. - М.: Вільявс, 2010. - 800 с.
63. Елліс М. Довідковий посібник з мови C++ з коментарями / М. Елліс, Б. Страуструп. - М.: Світ, 1992. - 445с.
64. Естербю О. Прямі методи для розряджених матриць / О. Естербю, З. Златев; пров. з англ. Х. Д. Ікрамов. - М.: Світ, 1987. - 120 с.

65.MatrixMarket [Електронний ресурс]. – Режим доступу:  
<http://math.nist.gov/MatrixMarket> – Заголовок з екрана.

## ДОДАТКИ

### Додаток А. Реалізація класу MV\_Vector

```
#ifndef _MV_VECTOR_TPL_H_
#define _MV_VECTOR_TPL_H_
#include <sstream>
#include <string>
#include <iostream>
#include <cstdlib>
// Обробка виняткових ситуацій
#include "../Exceptions/qtexception.h"
#ifdef MV_VECTOR_BOUNDS_CHECK
#include <cassert>
#endif
#include "mvvind.h"
#include "mvvrf.h"
template <class TYPE> class MV_Vector
{
protected:
    TYPE *p_;
    int dim_;
    int ref_;
public:
    MV_Vector();
    MV_Vector( int);
    MV_Vector( int, const TYPE&);

    MV_Vector(TYPE*, int);
    MV_Vector(const TYPE*, int);

    MV_Vector(TYPE*, int, MV_Vector::ref_type i);
    MV_Vector(const MV_Vector<TYPE>&);
    ~MV_Vector();

    inline TYPE& operator()( int i)
    {
        #ifdef MV_VECTOR_BOUNDS_CHECK
        assert(i < dim_);
        #endif
        return p_[i];
    }
    inline const TYPE& operator()( int i) const
    {
        #ifdef MV_VECTOR_BOUNDS_CHECK
        assert(i < dim_);
        #endif
        return p_[i];
    }
    inline TYPE& operator[]( int i)
    {
        #ifdef MV_VECTOR_BOUNDS_CHECK
        assert(i < dim_);
        #endif
        return p_[i];
    }
    inline const TYPE& operator[]( int i) const
    {
        #ifdef MV_VECTOR_BOUNDS_CHECK
        assert(i < dim_);
        #endif
        return p_[i];
    }
};
```

```

    }
    inline MV_Vector<TYPE> operator() (const MV_VecIndex &I) ;
    inline MV_Vector<TYPE> operator() (void);
    inline const MV_Vector<TYPE> operator() (void) const;
    inline const MV_Vector<TYPE> operator() (const MV_VecIndex &I) const;
    inline int size() const { return dim_;}
    inline int ref() const { return ref_;}
    inline int null() const {return dim_== 0;}
    MV_Vector<TYPE> & newsize( int );
    MV_Vector<TYPE> & newsize( int, int );

    MV_Vector<TYPE> & operator=(const MV_Vector<TYPE>&);
    MV_Vector<TYPE> & operator=(const TYPE&);
    friend std::ostream& operator<<(std::ostream &s, const MV_Vector<TYPE>
&V)
    {
        int N = V.size();
        for (int i=0; i< N; i++)
            s << V(i) << "\n";
        return s;
    }
};
template <class TYPE>
MV_Vector<TYPE>::MV_Vector() : p_(0), dim_(0) , ref_(0)
{
}
template <class TYPE>
MV_Vector<TYPE>::MV_Vector( int n) : p_(new TYPE[n]), dim_(n), ref_(0)
{
    if (p_ == NULL)
    {
        throw QtException("Error: NULL pointer in MV_Vector(int) constructor
Most likely out of memory.");
    }
}
template <class TYPE>
MV_Vector<TYPE>::MV_Vector( int n, const TYPE& v) : p_(new TYPE[n]), dim_(n),
ref_(0)
{
    if (p_ == NULL)
    {
        throw QtException("Error: NULL pointer in MV_Vector(int) constructor
Most likely out of memory.");
    }
    for (int i=0; i<n; i++)
        p_[i] = v;
}
template <class TYPE>
MV_Vector<TYPE>& MV_Vector<TYPE>::operator=(const TYPE & m)
{
    int N = size();
    int Nminus4 = N-4;
    int i;
    for (i=0; i<Nminus4; )
    {
        p_[i++] = m;
        p_[i++] = m;
        p_[i++] = m;
        p_[i++] = m;
    }
    for (; i<N; p_[i++] = m);
    return *this;
}
template <class TYPE>

```

```

MV_Vector<TYPE>& MV_Vector<TYPE>::newsize( int n)
{
    if (ref_ )
    {
        throw QtException("MV_Vector::newsize can't operator on
references.");
    }
    else
    if (dim_ != n )
    {
        if (p_) delete [] p_;
        p_ = new TYPE[n];
        if (p_ == NULL)
        {
            throw QtException("Error : NULL pointer in operator= ");
        }
        dim_ = n;
    }
    return *this;
}
template <class TYPE>
MV_Vector<TYPE>& MV_Vector<TYPE>::newsize( int n, int v)
{
    if (ref_ )
    {
        throw QtException("MV_Vector::newsize can't operator on
references.");
    }
    else
    if (dim_ != n )
    {
        if (p_) delete [] p_;
        p_ = new TYPE[n];
        if (p_ == NULL)
        {
            throw QtException("Error : NULL pointer in operator= ");
        }
        dim_ = n;
        for (int i=0; i<n; i++)
            p_[i] = v;
    }
    return *this;
}
template <class TYPE>
MV_Vector<TYPE>& MV_Vector<TYPE>::operator=(const MV_Vector<TYPE> & m)
{
    int N = m.dim_;
    int i;
    if (ref_ )
    {
        if (dim_ != m.dim_)
        {
            throw QtException("MV_VectorRef::operator= non-conformant
assignment.");
        }
        if ((m.p_ + m.dim_) >= p_)
        {
            for (i= N-1; i>=0; i--)
                p_[i] = m.p_[i];
        }
        else
        {
            for (i=0; i<N; i++)

```

```

        p_[i] = m.p_[i];
    } }
else
{
    newsize(N);
    for (i =0; i< N; i++)
        p_[i] = m.p_[i];
}
return *this;
}
template <class TYPE>
MV_Vector<TYPE>::MV_Vector(const MV_Vector<TYPE> & m) : p_(new TYPE[m.dim_]),
    dim_(m.dim_) , ref_(0)
{
    if (p_ == NULL)
    {
        throw QtException("Error: Null pointer in MV_Vector(const
MV_Vector&).");
    }
    int N = m.dim_;
    for (int i=0; i<N; i++)
        p_[i] = m.p_[i];
}
template <class TYPE>
MV_Vector<TYPE>::MV_Vector(TYPE* d,   int n, MV_Vector_::ref_type i) :
    p_(d), dim_(n) , ref_(i) {}
template <class TYPE>
MV_Vector<TYPE>::MV_Vector(TYPE* d,   int n) : p_(new TYPE[n]),
    dim_(n) , ref_(0)
{
    if (p_ == NULL)
    {
        throw QtException("Error: Null pointer in MV_Vector(TYPE*, int)");
    }
    for (int i=0; i<n; i++)
        p_[i] = d[i];
}
template <class TYPE>
MV_Vector<TYPE>::MV_Vector(const TYPE* d,   int n) : p_(new TYPE[n]),
    dim_(n) , ref_(0)
{
    if (p_ == NULL)
    {
        throw QtException("Error: Null pointer in MV_Vector(TYPE*, int)");
    }
    for (int i=0; i<n; i++)
        p_[i] = d[i];
}
template <class TYPE>
MV_Vector<TYPE> MV_Vector<TYPE>::operator() (void)
{
    return MV_Vector<TYPE>(p_, dim_, MV_Vector_::ref);
}
template <class TYPE>
const MV_Vector<TYPE> MV_Vector<TYPE>::operator() (void) const
{
    return MV_Vector<TYPE>(p_, dim_, MV_Vector_::ref);
}
template <class TYPE>
MV_Vector<TYPE> MV_Vector<TYPE>::operator() (const MV_VecIndex &I)
{
    if (I.all())
        return MV_Vector<TYPE>(p_, dim_, MV_Vector_::ref);

```

```

else
{
    if ( I.end() >= dim_ )
    {
        std::stringstream ss;
        ss << "MV_VecIndex: (" << I.start() << ":" << I.end() <<
            ") too big for matrix (0:" << dim_ - 1 << ") " << std::endl;
        throw QtException(ss.str());
    }
    return MV_Vector<TYPE>(p_ + I.start(), I.end() - I.start() + 1,
        MV_Vector_::ref);
}
}

template <class TYPE>
const MV_Vector<TYPE> MV_Vector<TYPE>::operator() (const MV_VecIndex &I) const
{
    if ( I.end() >= dim_ )
    {
        std::stringstream ss;
        ss << "MV_VecIndex: (" << I.start() << ":" << I.end() <<
            ") too big for matrix (0:" << dim_ - 1 << ") " << std::endl;
        throw QtException(ss.str());
    }
    return MV_Vector<TYPE>(p_ + I.start(), I.end() - I.start() + 1,
        MV_Vector_::ref);
}

template <class TYPE>
MV_Vector<TYPE>::~~MV_Vector()
{
    if (p_ && !ref_ ) delete [] p_;
}

#include "mvblas.h"
#endif

```

## Додаток Б. Реалізація класу CompCol\_Mat\_double

```
#ifndef CompCol_Mat_double_H
#define CompCol_Mat_double_H
#include <sstream>
#include <string>
#include <iostream>
#include <cstdlib>
#include "../Exceptions/qtexception.h"
#include "../MV/vecdefs.h"
class CompRow_Mat_double;
class Coord_Mat_double;
class CompCol_Mat_double
{
private:
    VECTOR_double    val_;
    VECTOR_int       rowind_;
    VECTOR_int       colptr_;
    int base_;
    int nz_;
    int dim_[2];

public:
    CompCol_Mat_double(void);
    CompCol_Mat_double(const CompCol_Mat_double &S);
    CompCol_Mat_double(const CompRow_Mat_double &R);
    CompCol_Mat_double(const Coord_Mat_double &CO);
    CompCol_Mat_double(int M, int N, int nz, double *val, int *r, int *c,
int base = 0);
    CompCol_Mat_double(int M, int N, int nz, const VECTOR_double &val,
const VECTOR_int &r, const VECTOR_int &c, int
base = 0);
    ~CompCol_Mat_double() {}
    double&    val(int i) { return val_(i); }
    int&       row_ind(int i) { return rowind_(i); }
    int&       col_ptr(int i) { return colptr_(i); }
    const double&    val(int i) const { return val_(i); }
    const int&       row_ind(int i) const { return rowind_(i); }
    const int&       col_ptr(int i) const { return colptr_(i); }
    int           dim(int i) const {return dim_[i];}
    int           size(int i) const {return dim_[i];}
    int           NumNonzeros() const {return nz_;}
    int           base() const {return base_;}
    CompCol_Mat_double& operator=(const CompCol_Mat_double &C);
    CompCol_Mat_double& newsize(int M, int N, int nz);
    double        operator() (int i, int j) const;
    double&       set(int i, int j);
    VECTOR_double operator*(const VECTOR_double &x) const;
    VECTOR_double trans_mult(const VECTOR_double &x) const;
};

    std::ostream& operator << (std::ostream & os, const CompCol_Mat_double &
mat);
#endif

#include "compcol_double.h"
#include "comprow_double.h"
#include "coord_double.h"
#include "spblas.h"
CompCol_Mat_double::CompCol_Mat_double(void)
    : val_(0), rowind_(0), colptr_(0), base_(0), nz_(0)
{
    dim_[0] = 0;
    dim_[1] = 0;
}
```



```

}
CompCol_Mat_double::CompCol_Mat_double(const CompCol_Mat_double &S) :
    val_(S.val_), rowind_(S.rowind_), colptr_(S.colptr_),
    base_(S.base_), nz_(S.nz_)
{
    dim_[0] = S.dim_[0];
    dim_[1] = S.dim_[1];
}
CompCol_Mat_double::CompCol_Mat_double(int M, int N, int nz, double *val,
                                         int *r, int *c, int base) :
    val_(val, nz), rowind_(r, nz), colptr_(c, N+1), base_(base), nz_(nz)
{
    dim_[0] = M;
    dim_[1] = N;
}
CompCol_Mat_double::CompCol_Mat_double(int M, int N, int nz,
                                         const VECTOR_double &val, const VECTOR_int &r,
                                         const VECTOR_int &c, int base) :
    val_(val), rowind_(r), colptr_(c), base_(base), nz_(nz)
{
    dim_[0] = M;
    dim_[1] = N;
}
CompCol_Mat_double::CompCol_Mat_double(const CompRow_Mat_double &R) :
    val_(R.NumNonzeros()), rowind_(R.NumNonzeros()), colptr_(R.dim(1)
+1),
    base_(R.base()), nz_(R.NumNonzeros())
{
    dim_[0] = R.dim(0);
    dim_[1] = R.dim(1);
    int i,j;
    VECTOR_int tally(R.dim(1)+1, 0);
    for (i=0;i<nz_;i++) tally(R.col_ind(i))++;
    colptr_(0) = 0;
    for (j=0;j<dim_[1];j++) colptr_(j+1) = colptr_(j)+tally(j);
    tally = colptr_;
    int count = 0;
    for (i=1;i<=dim_[0];i++)
    {
        for (j=count;j<R.row_ptr(i);j++)
        {
            val_(tally(R.col_ind(j))) = R.val(j);
            rowind_(tally(R.col_ind(j))) = i-1;
            tally(R.col_ind(count))++;
            count++;
        }
    }
}
CompCol_Mat_double::CompCol_Mat_double(const Coord_Mat_double &CO) :
    val_(CO.NumNonzeros()), rowind_(CO.NumNonzeros()),
    colptr_(CO.dim(1) +1), base_(CO.base()), nz_(CO.NumNonzeros())
{
    dim_[0] = CO.dim(0);
    dim_[1] = CO.dim(1);
    int i,j;
    VECTOR_int tally(CO.dim(1)+1, 0);
    for (i=0;i<nz_;i++) tally(CO.col_ind(i))++;
    colptr_(0) = 0;
    for (j=0;j<dim_[1];j++) colptr_(j+1) = colptr_(j)+tally(j);
    tally = colptr_;
    for (i = 0; i < nz_; i++)
    {
        val_(tally(CO.col_ind(i))) = CO.val(i);
    }
}

```

```

        rowind_(tally(CO.col_ind(i))) = CO.row_ind(i);
        tally(CO.col_ind(i))++;
    }
}
CompCol_Mat_double& CompCol_Mat_double::operator=(const CompCol_Mat_double
&C)
{
    dim_[0] = C.dim_[0];
    dim_[1] = C.dim_[1];
    base_   = C.base_;
    nz_     = C.nz_;
    val_    = C.val_;
    rowind_ = C.rowind_;
    colptr_ = C.colptr_;
    return *this;
}
CompCol_Mat_double& CompCol_Mat_double::newsize(int M, int N, int nz)
{
    dim_[0] = M;
    dim_[1] = N;
    nz_ = nz;
    val_.newsize(nz);
    rowind_.newsize(nz);
    colptr_.newsize(N+1);
    return *this;
}
double CompCol_Mat_double::operator()(int i, int j) const
{
    for (int t=colptr_(j); t<colptr_(j+1); t++)
        if (rowind_(t) == i) return val_(t);
    if (i < dim_[0] && j < dim_[1]) return 0.0;
    else
    {
        throw QtException("Array accessing exception - out of bounds.");
        return (0); // Для того, щоб заспокоїти компілятор
    }
}
double& CompCol_Mat_double::set(int i, int j)
{
    for (int t=colptr_(j); t<colptr_(j+1); t++)
        if (rowind_(t) == i) return val_(t);
    std::stringstream ss;
    ss<<"Array element ("<<i<<","<<j<<") not in sparse structure - cannot
assign.";
    throw QtException(ss.str());
    return val_(0); // Для того, щоб заспокоїти компілятор
}
VECTOR_double CompCol_Mat_double::operator*(const VECTOR_double &x)
const
{
    int M = dim_[0];
    int N = dim_[1];
    if (x.size() != N)
    {
        throw QtException("Error in CompCol Matvec - incompatible
dimensions.");
        return x;
    }
    VECTOR_double result(M, 0.0);
    VECTOR_double work(M);

    int descra[9];
    descra[0] = 0;

```

```

        descra[1] = 0;
        descra[2] = 0;
        F77NAME(dscmm) (0, M, 1, N, 1.0,
                        descra, &val_(0), &rowind_(0), &colptr_(0),
                        &x(0), N, 1.0, &result(1), M,
                        &work(1), M);
        return result;
    }
    std::ostream& operator << (std::ostream & os, const CompCol_Mat_double & mat)
    {
        int M = mat.dim(0);
        int N = mat.dim(1);
        int rowp1, colp1;
        int flag = 0;
        std::ios::fmtflags olda = os.setf(std::ios::right,
std::ios::adjustfield);
        std::ios::fmtflags oldf = os.setf(std::ios::scientific,
std::ios::floatfield);
        int oldp = os.precision(12);
        // Цыкл по столбцам
        for (int j = 0; j < N ; j++)
            for (int i=mat.col_ptr(j);i<mat.col_ptr(j+1);i++)
            {
                rowp1 = mat.row_ind(i)+1;
                colp1 = j + 1;
                if ( rowp1 == M && colp1 == N ) flag = 1;
                os.width(14);
                os << rowp1 ; os << "      " ;
                os.width(14);
                os << colp1 ; os << "      " ;
                os.width(20);
                os << mat.val(i) << "\n";
            }
        if (flag == 0)
        {
            os.width(14);
            os << M ; os << "      " ;
            os.width(14);
            os << N ; os << "      " ;
            os.width(20);
            os << mat(M-1,N-1) << "\n";
        }
        os.setf(olda, std::ios::adjustfield);
        os.setf(oldf, std::ios::floatfield);
        os.precision(oldp);
        return os;
    }
}

```

## Додаток В. Реалізація класу CompRow\_Mat\_double

```
#ifndef CompRow_Mat_double_H
#define CompRow_Mat_double_H
#include <sstream>
#include <string>
#include <iostream>
#include <cstdlib>
#include "../Exceptions/qtexception.h"
#include "../MV/vecdefs.h"
class CompCol_Mat_double;
class Coord_Mat_double;
class CompRow_Mat_double
{
private:
    VECTOR_double val_;
    VECTOR_int rowptr_;
    VECTOR_int colind_;
    int base_;
    int nz_;
    int dim_[2];
public:
    CompRow_Mat_double(void);
    CompRow_Mat_double(const CompRow_Mat_double &S);
    CompRow_Mat_double(const CompCol_Mat_double &C);
    CompRow_Mat_double(const Coord_Mat_double &CO);
    CompRow_Mat_double(int M, int N, int nz, double *val, int *r, int *c,
int base = 0);
    CompRow_Mat_double(int M, int N, int nz, const VECTOR_double &val,
const VECTOR_int &r, const VECTOR_int &c,int
base = 0);
    ~CompRow_Mat_double() {}
    double& val(int i) { return val_(i); }
    int& row_ptr(int i) { return rowptr_(i); }
    int& col_ind(int i) { return colind_(i); }
    const double& val(int i) const { return val_(i); }
    const int& row_ptr(int i) const { return rowptr_(i); }
    const int& col_ind(int i) const { return colind_(i); }
    int dim(int i) const {return dim_[i];}
    int size(int i) const {return dim_[i];}
    int NumNonzeros() const {return nz_;}
    int base() const {return base_;}
    CompRow_Mat_double& operator=(const CompRow_Mat_double &R);
    CompRow_Mat_double& newsize(int M, int N, int nz);
    double operator() (int i, int j) const;
    double& set(int i, int j);
    VECTOR_double operator*(const VECTOR_double &x) const;
    VECTOR_double trans_mult(const VECTOR_double &x) const;
};

std::ostream& operator << (std::ostream & os, const CompRow_Mat_double &
mat);
#endif
#include "compcol_double.h"
#include "comprow_double.h"
#include "coord_double.h"
#include "spblas.h"
CompRow_Mat_double::CompRow_Mat_double(void)
: val_(0), rowptr_(0), colind_(0), base_(0), nz_(0)
{
    dim_[0] = 0;
    dim_[1] = 0;
}
```

```

}
CompRow_Mat_double::CompRow_Mat_double(const CompRow_Mat_double &S) :
    val_(S.val_), rowptr_(S.rowptr_), colind_(S.colind_), base_(S.base_),
    nz_(S.nz_)
{
    dim_[0] = S.dim_[0];
    dim_[1] = S.dim_[1];
}
CompRow_Mat_double::CompRow_Mat_double(int M, int N, int nz, double *val,
                                         int *r, int *c, int base) :
    val_(val, nz), rowptr_(r, M+1), colind_(c, nz), base_(base), nz_(nz)
{
    dim_[0] = M;
    dim_[1] = N;
}
CompRow_Mat_double::CompRow_Mat_double(int M, int N, int nz,
                                         const VECTOR_double &val, const VECTOR_int &r,
                                         const VECTOR_int &c, int base) :
    val_(val), rowptr_(r), colind_(c), base_(base), nz_(nz)
{
    dim_[0] = M;
    dim_[1] = N;
}
CompRow_Mat_double::CompRow_Mat_double(const CompCol_Mat_double &C) :
    val_(C.NumNonzeros()), rowptr_(C.dim(0)+1),
    colind_(C.NumNonzeros()), base_(C.base()), nz_(C.NumNonzeros())
{
    dim_[0] = C.dim(0);
    dim_[1] = C.dim(1);

    int i,j;
    VECTOR_int tally(C.dim(0)+1, 0);
    for (i=0;i<nz_;i++) tally(C.row_ind(i))++;
    rowptr_(0) = 0;
    for (j=0;j<dim_[0];j++) rowptr_(j+1) = rowptr_(j)+tally(j);
    tally = rowptr_;

    int count = 0;
    for (i=1;i<=dim_[1];i++)
    {
        for (j=count;j<C.col_ptr(i);j++)
        {
            val_(tally(C.row_ind(j))) = C.val(j);
            colind_(tally(C.row_ind(j))) = i-1;
            tally(C.row_ind(count))++;
            count++;
        }
    }
}
CompRow_Mat_double::CompRow_Mat_double(const Coord_Mat_double &CO) :
    val_(CO.NumNonzeros()), rowptr_(CO.dim(0)+1),
    colind_(CO.NumNonzeros()), base_(CO.base()), nz_(CO.NumNonzeros())
{
    dim_[0] = CO.dim(0);
    dim_[1] = CO.dim(1);
    int i;
    VECTOR_int tally(CO.dim(0)+1, 0);
    for (i=0;i<nz_;i++) tally(CO.row_ind(i))++;
    rowptr_(0) = 0;
    for (i=0;i<dim_[0];i++) rowptr_(i+1) = rowptr_(i)+tally(i);
    tally = rowptr_;
    for (i = 0; i < nz_; i++)
    {

```

```

        val_(tally(CO.row_ind(i))) = CO.val(i);
        colind_(tally(CO.row_ind(i))) = CO.col_ind(i);
        tally(CO.row_ind(i))++;
    }
}
CompRow_Mat_double& CompRow_Mat_double::operator=(const CompRow_Mat_double
&R)
{
    dim_[0] = R.dim_[0];
    dim_[1] = R.dim_[1];
    base_    = R.base_;
    nz_      = R.nz_;
    val_     = R.val_;
    rowptr_  = R.rowptr_;
    colind_  = R.colind_;
    return *this;
}
CompRow_Mat_double& CompRow_Mat_double::newsize(int M, int N, int nz)
{
    dim_[0] = M;
    dim_[1] = N;
    nz_     = nz;
    val_.newsize(nz);
    rowptr_.newsize(M+1);
    colind_.newsize(nz);
    return *this;
}
double CompRow_Mat_double::operator()(int i, int j) const
{
    for (int t=rowptr_(i); t<rowptr_(i+1); t++)
        if (colind_(t) == j) return val_(t);
    if (i < dim_[0] && j < dim_[1]) return 0.0;
    else
    {
        throw QtException("Array accessing exception - out of bounds.");
        return (0);
    }
}
double& CompRow_Mat_double::set(int i, int j)
{
    for (int t=rowptr_(i); t<rowptr_(i+1); t++)
        if (colind_(t) == j) return val_(t);
    std::stringstream ss;
    ss<<"Array element ("<<i<<","<<j<<") not in sparse structure - cannot
assign.";
    throw QtException(ss.str());
    return val_(0);
}
VECTOR_double CompRow_Mat_double::operator*(const VECTOR_double &x)
const
{
    int M = dim_[0];
    int N = dim_[1];
    if (x.size() != N)
    {
        throw QtException("Error in CompCol Matvec - incompatible
dimensions.");
        return x;          // Для того, щоб заспокоїти компілятор
    }
    VECTOR_double result(M, 0.0);
    VECTOR_double work(M);

    int descra[9];

```

```

        descra[0] = 0;
        descra[1] = 0;
        descra[2] = 0;
        F77NAME(dcsrmm) (0, M, 1, N, 1.0,
                        descra, &val_(0), &colind_(0), &rowptr_(0),
                        &x(1), N, 1.0, &result(0), M,
                        &work(0), M);
        return result;
}

std::ostream& operator << (std::ostream & os, const CompRow_Mat_double & mat)
{
    int M = mat.dim(0);
    int N = mat.dim(1);
    int rowp1, colp1;
    int flag = 0;
    std::ios::fmtflags olda = os.setf(std::ios::right,
std::ios::adjustfield);
    std::ios::fmtflags oldf = os.setf(std::ios::scientific,
std::ios::floatfield);
    int oldp = os.precision(12);
    // Цыкл по строкам
    for (int i = 0; i < M ; i++)
        for (int j=mat.row_ptr(i);j<mat.row_ptr(i+1);j++)
        {
            rowp1 = i + 1;
            colp1 = mat.col_ind(j) + 1;
            if ( rowp1 == M && colp1 == N ) flag = 1;
            os.width(14);
            os << rowp1 ; os << "    " ;
            os.width(14);
            os << colp1 ; os << "    " ;
            os.width(20);
            os << mat.val(j) << "\n";
        }
    if (flag == 0)
    {
        os.width(14);
        os << M ; os << "    " ;
        os.width(14);
        os << N ; os << "    " ;
        os.width(20);
        os << mat(M-1,N-1) << "\n";
    }
    os.setf(olda, std::ios::adjustfield);
    os.setf(oldf, std::ios::floatfield);
    os.precision(oldp);
    return os;
}

```

## Результати обчислень матриці А

Метод розв'язання	Схема зберігання	Передумовник	Ітерацій	Час (с.)
Метод спряжених градієнтів	CSC	Якобі	3000	4.61461
		Холецького	3000	7.53186
		ILU	3000	7.26612
	CSR	Якобі	3000	4.52036
		Холецького	3000	7.87631
		ILU	3000	7.03978
Метод біспряжених градієнтів	CSC	Якобі	110	0.331949
		Холецького	125	0.643338
		ILU	129	0.656141
	CSR	Якобі	110	0.329862
		Холецького	125	0.638196
		ILU	129	0.650483
Метод біспряжених градієнтів зі стабілізацією	CSC	Якобі	402	1.35525
		Холецького	93	0.582855
		ILU	3000	0.572193
	CSR	Якобі	402	1.38109
		Холецького	93	0.526853
		ILU	3000	0.574244
Квадратичний метод спряжених градієнтів	CSC	Якобі	62	0.206345
		Холецького	159	0.85413
		ILU	165	0.884821
	CSR	Якобі	62	0.210762
		Холецького	149	0.844134
		ILU	199	1.03282
Узагальнений метод мінімальних нев'язок	CSC	Якобі	67	0.185979
		Холецького	110	0.4318
		ILU	127	0.509911
	CSR	Якобі	67	0.185555
		Холецького	110	0.428708
		ILU	127	0.506716
Метод квазімінімальних нев'язок	CSC	Якобі	917	4.33082
		Холецького	629	5.30062
		ILU	3000	23.7931
	CSR	Якобі	917	4.13803
		Холецького	608	5.19617
		ILU	3000	23.4654



Метод розв'язання	Схема зберігання	Передумовник	Ітерацій	Час (с.)
Метод Річардсона	CSC	Якобі	3000	3.57283
		Холецького	3000	6.50613
		ILU	3000	6.17907
	CSR	Якобі	3000	3.54343
		Холецького	3000	6.93418
		ILU	3000	6.07315
Метод Чебишева	CSC	Якобі	3000	4.43466
		Холецького	3000	7.27296
		ILU	2797	6.57052
	CSR	Якобі	3000	4.54279
		Холецького	3000	9.91164
		ILU	2797	6.35639

Таблиця Г.2

Результати обчислень матриці Б

Метод розв'язання	Схема зберігання	Передумовник	Ітерацій	Час (с.)
Метод спряжених градієнтів	CSC	Якобі	407	0.103967
		Холецького	128	0.0740123
		ILU	76	0.0349377
	CSR	Якобі	407	0.100423
		Холецького	128	0.0608576
		ILU	76	0.0345246
Метод біспряжених градієнтів	CSC	Якобі	311	0.148166
		Холецького	100	0.094184
		ILU	68	0.0585873
	CSR	Якобі	311	0.14623
		Холецького	100	0.0913659
		ILU	68	0.0587729
Метод біспряжених градієнтів зі стабілізацією	CSC	Якобі	278	0.190209
		Холецького	3000	0.0798196
		ILU	3000	0.0639321
	CSR	Якобі	278	0.184286
		Холецького	3000	0.0919525
		ILU	3000	0.0606988
Квадратичний метод спряжених	CSC	Якобі	440	0.23856
		Холецького	85	0.0847072
		ILU	66	0.0639849

Метод розв'язання	Схема зберігання	Передумовник	Ітерацій	Час (с.)
градієнтів	CSR	Якобі	440	0.2883
		Холецького	85	0.0920128
		ILU	67	0.0655669
Узагальнений метод мінімальних нев'язок	CSC	Якобі	3000	1.9259
		Холецького	267	0.234729
		ILU	128	0.113213
	CSR	Якобі	3000	1.95306
		Холецького	288	0.251872
		ILU	128	0.109324
Метод квазімінімальних нев'язок	CSC	Якобі	3000	2.28642
		Холецького	3000	4.88698
		ILU	2505	4.06416
	CSR	Якобі	3000	2.76433
		Холецького	3000	4.86368
		ILU	2500	3.69673
Метод Річардсона	CSC	Якобі	3000	0.579719
		Холецького	3000	1.21133
		ILU	3000	1.09057
	CSR	Якобі	3000	0.562555
		Холецького	3000	1.1883
		ILU	3000	1.05557
Метод Чебишева	CSC	Якобі	3000	0.712338
		Холецького	3000	1.34694
		ILU	2180	0.897073
	CSR	Якобі	3000	0.685391
		Холецького	3000	1.3174
		ILU	2180	0.873463