Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна

Д. О. Протектор
І. В. Гарячевська

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Бібліотека «MATRIX» для роботи з матрицями на C++

Навчально-методичний посібник

Харків – 2025

Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна

Д. О. Протектор
І. В. Гарячевська

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ
Бібліотека «MATRIX» для роботи з матрицями на C++

Навчально-методичний посібник
для здобувачів вищої освіти за спеціальністю
Е6 «Прикладна фізика та наноматеріали»

Електронний ресурс

Харків – 2025

УДК 004.43
П 78

Рецензенти:

О. О. Стрельнікова – д.т.н., проф., провідний науковий співробітник відділу гідроаеромеханіки енергетичних машин Інституту енергетичних машин і систем ім. А. М. Підгорного НАН України;

Р. В. Сухов – к.ф.-м.н., доц., завідувач кафедри інформаційних технологій в фізико-енергетичних системах навчально-наукового інституту комп'ютерної фізики та енергетики Харківського національного університету імені В. Н. Каразіна.

Затверджено до розміщення в мережі Інтернет рішенням Науково-методичної ради Харківського національного університету імені В. Н. Каразіна (протокол № 10 від 21 травня 2025 року)

Протектор Д. О.

П 78 Об'єктно-орієнтоване програмування: бібліотека «MATRIX» для роботи з матрицями на C++ : навчально-методичний посібник для здобувачів вищої освіти за спеціальністю Е6 «Прикладна фізика та наноматеріали» [Електронний ресурс] / Д. О. Протектор, І. В. Гарячевська. – Харків : ХНУ імені В. Н. Каразіна, 2025. – (PDF 100 с.)

Навчально-методичний посібник розроблено відповідно до програми курсу «Об'єктно-орієнтоване програмування», що є складовою підготовки бакалаврів за спеціальністю Е6 «Прикладна фізика та наноматеріали». Метою цього навчально-методичного посібника є формування у студентів теоретичних знань та практичних навичок роботи з бібліотекою «MATRIX» у мові програмування C++ та застосуванні її функціоналу для розв'язання практичних задач.

Навчально-методичний посібник розрахований на студентів денної форми навчання навчально-наукового інституту комп'ютерної фізики та енергетики. Він містить загальні відомості про бібліотеку «MATRIX», її призначення та основні можливості, огляд основних компонентів класу Matrix, а також приклади використання бібліотеки для роботи з матрицями. Наведені приклади демонструють використання бібліотеки для розв'язання систем лінійних алгебраїчних рівнянь, обчислення градієнта, оператора Лапласа, чисельного диференціювання, а також виконання інших математичних операцій, що є важливими для аналізу та моделювання фізичних процесів в енергетиці. Навчально-методичний посібник також містить перелік рекомендованої літератури та інформаційні ресурси, що можуть бути корисними для глибшого засвоєння матеріалу.

УДК 004.43

ЗМІСТ

1. ВСТУП

Чисельні методи та математичне моделювання є невід'ємною частиною наукових досліджень і технічних розрахунків у фізиці, енергетиці та інших природничих науках. Робота з великими обсягами числових даних вимагає ефективних програмних засобів, які дозволяють виконувати складні обчислення з високою точністю та продуктивністю. Одним із таких засобів є бібліотека «MATRIX» для мови програмування C++, яка містить широкий набір функцій для роботи з двовимірними матрицями та забезпечує високу ефективність при виконанні складних математичних обчислень.

Бібліотека «MATRIX» розроблена авторами цього навчально-методичного посібника і не входить до складу стандартних бібліотек мови C++ (Додаток А). Вона створена з урахуванням специфічних вимог до обчислювальних задач, що постають у фізиці, енергетиці та суміжних наукових галузях. Основна мета її розробки – забезпечити зручний і ефективний інструмент для виконання математичних операцій із матрицями, що дозволяє суттєво спростити процес програмування чисельних методів та моделювання складних систем.

Цей навчально-методичний посібник присвячений вивченню можливостей бібліотеки «MATRIX» та її використанню в задачах обчислювальної фізики, прикладної математики й енергетики. Бібліотека надає інструменти для створення, обробки та аналізу матриць, що дозволяє ефективно розв'язувати задачі лінійної алгебри, чисельного диференціювання, інтерполяції, спектрального аналізу та інших напрямів, де необхідні операції з двовимірними масивами числових даних. Посібник орієнтований на студентів, які вивчають курс «Об'єктно-орієнтоване програмування» у межах підготовки за спеціальністю Е6 «Прикладна фізика та наноматеріали».

Мета цього посібника – надати студентам теоретичні знання та практичні навички використання бібліотеки «MATRIX» на мові програмування C++. Посібник містить опис основних можливостей бібліотеки, розгляд її структури та функціоналу, а також численні приклади коду, що наочно демонструють

ефективне застосування бібліотеки для розв'язання різноманітних практичних задач.

Структура посібника побудована таким чином, щоб поступово знайомити читача з можливостями бібліотеки:

- У **Розділі 1** наведено загальні відомості про бібліотеку «MATRIX», її призначення та основні можливості;
- **Розділ 2** присвячений практичним аспектам використання бібліотеки, зокрема роботі з конструкторами, перевантаженими операторами та методами класу `Matrix`;
- **Розділ 3** та **Розділ 4** містять перелік рекомендованої літератури та інформаційні ресурси, які можуть бути корисними для подальшого вивчення предметної області.

Запропоновані матеріали сприятимуть глибшому розумінню принципів об'єктно-орієнтованого програмування в контексті числових розрахунків та дозволять студентам ефективно використовувати бібліотеку «MATRIX» для обробки даних та розв'язання практичних задач, що є важливими у сфері прикладної фізики, енергетики та комп'ютерного моделювання.

1.1. Загальні положення

Дисципліна «Об'єктно-орієнтоване програмування» викладається студентам першого (бакалаврського) рівня вищої освіти, які навчаються за спеціальністю Е6 «Прикладна фізика та наноматеріали» в навчально-науковому інституті комп'ютерної фізики та енергетики.

Курс включає 15 лекційних тем, цикл лабораторних робіт та контрольні роботи, що дозволяють студентам набути як теоретичних знань, так і практичних навичок програмування в парадигмі об'єктно-орієнтованого програмування.

Метою курсу є вивчення сучасних принципів об'єктно-орієнтованого програмування з використанням мови C++ та актуальних інтегрованих середовищ розробки, а також формування навичок розробки програмного забезпечення для

автоматизованої обробки великих обсягів даних та комп'ютерного моделювання фізичних процесів в енергетиці.

Основні завдання курсу:

- освоїти концепції об'єктно-орієнтованого програмування, включаючи інкапсуляцію, успадкування, поліморфізм, перевантаження операторів та використання шаблонів, для створення структурованого, ефективного та масштабованого програмного забезпечення;
- навчитися працювати з вказівниками, динамічною пам'яттю, операторами `new` та `delete` в мові програмування C++;
- вивчити методи роботи з потоками вводу/виводу та файловими системами;
- навчитися ефективному використанню контейнерних класів (`vector`, `list`, `stack`, `queue`, `map` тощо) для зберігання та обробки даних;
- розглянути алгоритми обробки даних, їх оптимізацію та ефективність;
- набути практичних навичок розробки програмного забезпечення на мові програмування C++ із використанням парадигми об'єктно-орієнтованого програмування.

У результаті вивчення даного курсу студент повинен **досягти таких результатів навчання**:

- знати основні принципи об'єктно-орієнтованого програмування та їх реалізацію в мові програмування C++;
- знати методи управління пам'яттю в мові програмування C++;
- знати механізми роботи з потоками вводу/виводу та файлами;
- знати основи роботи з контейнерами та алгоритмами стандартної бібліотеки шаблонів (Standard Template Library) в мові програмування C++;
- вміти розробляти програмне забезпечення в парадигмі об'єктно-орієнтованого програмування із використанням мови C++;

- ❖ вміти використовувати шаблони та контейнери для створення ефективного та масштабованого коду;
- ❖ вміти реалізовувати динамічне управління пам’яттю, оптимізуючи використання ресурсів.

1.2. Тематичний план навчальної дисципліни

Дисципліна «Об’єктно-орієнтоване програмування» складається з наступних лекційних тем:

Тема 1. Основи об’єктно-орієнтованого програмування: концепції та принципи.

Тема 2. Структурний та об’єктно-орієнтований підходи у програмуванні. Поняття класу. Властивості класу. Методи класу.

Тема 3. Специфікатори доступу: `public`, `private` та `protected`. Інкапсуляція.

Тема 4. Конструктори та деструктори: типи, правила використання, механізм копіювання об’єктів.

Тема 5. Динамічне управління пам’яттю: оператори `new` та `delete`, вказівник `this`, ключове слово `const`.

Тема 6. Механізм успадкування в мові програмування C++.

Тема 7. Перевизначення методів у похідних класах. Поняття поліморфізму. Раннє та пізнє зв’язування.

Тема 8. Віртуальні функції. Поняття абстрактного класу та чисті віртуальні функції. Віртуальні деструктори. Множинне спадкування, його види та правила застосування.

Тема 9. Перевантаження операторів в мові програмування C++.

Тема 10. Перевантаження операторів потокового введення/виведення. Форматування виводу. Обробка виключень.

Тема 11. Робота з потоками: ієрархія потокових класів, управління станом потоку, класи `istream` та `ostream`.

Тема 12. Стандартні консольні потоки: особливості використання `cin` і `cout`, методи обробки рядкових даних.

Тема 13. Файловий ввід/вивід: робота з текстовими та бінарними файлами за допомогою класів `ofstream` та `ifstream`.

Тема 14. Шаблони в мові програмування C++. Шаблонні функції. Шаблонні класи. Стандартна бібліотека шаблонів (STL). Контейнерні класи. Ітератори. Послідовні контейнери: `vector`, `list`, `stack`, `queue`. Асоціативні контейнери: `map`, `multimap`, `set`, `multiset`.

Тема 15. Алгоритми стандартної бібліотеки шаблонів. Алгоритми, що не змінюють послідовність. Алгоритми, що змінюють послідовність.

Таким чином, програма дисципліни «Об'єктно-орієнтоване програмування» охоплює повний цикл теоретичних і практичних аспектів сучасної парадигми програмування – від базових концепцій об'єктно-орієнтованого програмування до просунутих технік роботи з шаблонами та стандартною бібліотекою. Послідовне вивчення цих тем дозволяє студентам сформувати цілісне розуміння принципів проєктування та реалізації ефективних програмних рішень, опанувати потужний інструментарій мови C++ та набути практичних навичок розробки, які є затребуваними в сучасній IT-індустрії. Засвоєння представленого матеріалу створює міцну основу для подальшого професійного розвитку в галузі програмної інженерії та розв'язання складних прикладних задач в фізиці та енергетиці.

1.3. Призначення бібліотеки «MATRIX»

Бібліотека «MATRIX» призначена для роботи з двовимірними матрицями на мові програмування C++ та забезпечує інструменти для виконання математичних операцій, необхідних у чисельному моделюванні та аналізі даних.

Функціонал бібліотеки охоплює основні операції лінійної алгебри, зокрема додавання, віднімання, множення та ділення матриць, транспонування, піднесення до степеня, обчислення визначника та оберненої матриці. Також

реалізовано розклад матриць, чисельне диференціювання, обчислення градієнта, оператора Лапласа, пошук елементів за заданими критеріями та широкий набір математичних функцій, включаючи тригонометричні, експоненційні, логарифмічні та спеціальні функції (зокрема, функції Бесселя).

Бібліотека підтримує перевантаження операторів, що спрощує роботу з матрицями та дозволяє формулювати математичні вирази у природному для користувача вигляді. Крім того, реалізовано засоби введення та виведення даних, у тому числі роботу з файлами, що є важливим для збереження та обробки великих обсягів числових даних.

Бібліотека «MATRIX» може бути використана в навчальному процесі для розв'язання задач лінійної алгебри, чисельного аналізу та комп'ютерного моделювання. Вона є корисним інструментом для реалізації обчислювальних методів у задачах фізики та енергетики, зокрема при моделюванні теплових процесів, аналізі електромагнітних полів, обробці експериментальних даних та інших інженерних застосуваннях.

1.4. Основні можливості бібліотеки «MATRIX»

Бібліотека «MATRIX» забезпечує широкий набір функціональних можливостей для роботи з двовимірними матрицями на мові програмування C++. Її основні можливості охоплюють:

- **Арифметичні операції над матрицями** – додавання, віднімання, множення, ділення, транспонування, піднесення до степеня.
- **Обчислення характеристик матриці** – визначник, обернена матриця, слід матриці, діагональні елементи.
- **Робота з розрідженими та спеціальними матрицями** – створення одиничної, нульової, випадкової, діагональної матриць, а також їхніх модифікацій.

- **Чисельні методи** – LU-розклад, QR-розклад, розклад Холецького, SVD-розклад, розв'язання систем лінійних алгебраїчних рівнянь, чисельне диференціювання, обчислення градієнта, оператора Лапласа.
- **Обробка елементів матриці** – пошук мінімальних та максимальних значень, відбір елементів за заданими критеріями, виконання логічних операцій над елементами матриць.
- **Математичні функції** – підтримка тригонометричних, гіперболічних, експоненційних, логарифмічних функцій, а також спеціальних функцій, таких як функції Бесселя, Ханкеля тощо.
- **Операції з файлами** – збереження та зчитування матриць у текстовому та бінарному форматах.
- **Перевантаження операторів** – уніфікований синтаксис для зручної взаємодії з об'єктами класу `Matrix`.
- **Інтеграція з потоками введення/виведення** – зчитування та запис матриць через стандартні потоки вводу/виводу.

Бібліотека «MATRIX» може бути використана для проведення математичних обчислень у різних галузях науки та техніки. Вона може бути ефективно застосована для чисельного моделювання фізичних процесів, аналізу даних у задачах фізики та енергетики, а також інших напрямів, що потребують роботи з матрицями, зокрема в обробці сигналів, комп'ютерній графіці та машинному навчанні.

1.5. Доступ до бібліотеки «MATRIX»

Бібліотека «MATRIX» доступна для завантаження в дистанційному курсі «Об'єктно-орієнтоване програмування», який розміщено на навчальній платформі Moodle за посиланням: https://moodle.karazin.ua/course/view.php?id=2533. Студенти можуть безкоштовно завантажити бібліотеку разом із супровідними матеріалами, зокрема документацією та прикладами коду.

2. ОСНОВИ РОБОТИ З БІБЛІОТЕКОЮ «MATRIX»

Бібліотека «MATRIX» реалізована у вигляді класу `Matrix`, який забезпечує базові та розширені можливості для роботи з двовимірними матрицями у мові програмування C++. Робота з цим класом передбачає використання його конструкторів, перевантажених операторів та методів, що дозволяють виконувати різноманітні математичні операції, зчитувати та зберігати матриці, а також аналізувати їхні властивості.

Даний розділ містить опис основних компонентів класу `Matrix`, їх призначення та особливості використання.

2.1. Конструктори класу Matrix

Для створення об’єктів класу `Matrix` використовуються наступні конструктори:

- `Matrix()` – конструктор, який створює порожню матрицю;
- `Matrix(size_t rows, size_t columns, const double *const *const el)` – конструктор, який створює матрицю із заданою кількістю рядків `rows` та стовпців `columns`, елементи якої заповнюються із двовимірного масиву `el`;
- `Matrix(size_t rows, size_t columns, double fill = 0.0)` – конструктор, який створює матрицю із заданою кількістю рядків `rows` та стовпців `columns`, кожен елемент якої дорівнює значенню змінної `fill`;
- `Matrix(const std::pair<size_t, size_t> &size, double fill = 0.0)` – конструктор, який створює матрицю із заданою кількістю рядків та стовпців за допомогою пари `size`, кожен елемент якої дорівнює значенню змінної `fill`;
- `Matrix(const std::initializer_list<const std::initializer_list<double>> &lists)` – конструктор, який створює

матрицю із заданою кількістю рядків та стовпців на основі вкладених списків ініціалізаторів `lists`;

- `Matrix(const std::initializer_list<const std::initiali zer_list<Matrix>>& lists)` – конструктор, який створює матрицю шляхом об'єднання переданих у вкладені списки ініціалізаторів `lists` матриць в єдину велику матрицю. Кожен елемент `lists` містить список матриць, які інтерпретуються як блоки, що розташовуються у відповідних позиціях результуючої матриці. Всі передані матриці повинні мати узгоджені розміри відповідно до їх розташування, щоб забезпечити коректне злиття в єдину результуючу матрицю;
- `Matrix(const Matrix &m)` – конструктор, який створює матрицю на основі існуючої матриці `m`;
- `Matrix(Matrix&& m) noexcept` – конструктор, який створює матрицю шляхом переміщення ресурсів із матриці `m`, звільняючи її від керування даними та забезпечуючи ефективне створення нового об'єкта без зайвого копіювання.

Приклад 1:

```
#include "Matrix.h"

int main()
{
    // створюємо порожню матрицю
    Matrix a = Matrix();
    // створюємо матрицю розміром 2 x 3,
    // кожен елемент якої дорівнює 5.6
    Matrix b(2, 3, 5.6);
    // створюємо матрицю розміром 3 x 2
    // на основі вкладених списків ініціалізаторів
    Matrix c{ { 1, 2 }, { 3, 4 }, { 5, 6 } };
    // створюємо матрицю на основі існуючої матриці
```

```
    Matrix d(b);
    // створюємо матрицю розміром 3 x 2,
    // кожен елемент якої дорівнює 1.0
    Matrix m1(3, 2, 1.0);
    // створюємо матрицю розміром 3 x 3,
    // кожен елемент якої дорівнює 2.0
    Matrix m2(3, 3, 2.0);
    // створюємо матрицю розміром 4 x 3,
    // кожен елемент якої дорівнює 3.0
    Matrix m3(4, 3, 3.0);
    // створюємо матрицю розміром 4 x 2,
    // кожен елемент якої дорівнює 4.0
    Matrix m4(4, 2, 4.0);
    // створюємо матрицю розміром 7 x 5
    // шляхом об'єднання матриць m1, m2, m3, m4
    Matrix result{ { m1, m2 },
                   { m3, m4 } };
    // виводимо значення елементів
    // матриці result в консоль
    std::cout << "Matrix result:\n" << result;
    return 0;
}
```

Результат роботи програми:

```
Matrix result:
1 1 2 2 2
1 1 2 2 2
1 1 2 2 2
3 3 3 4 4
3 3 3 4 4
3 3 3 4 4
3 3 3 4 4
```

2.2. Оператори класу Matrix

Для об’єктів класу `Matrix` перевантажені наступні оператори:

- `Matrix& operator=(const Matrix &m)` – оператор виконує копіювання вмісту матриці `m` у поточний об’єкт, замінюючи його дані;

- `Matrix& operator=(Matrix&& m) noexcept` – оператор виконує переміщення вмісту матриці `m` у поточний об'єкт, звільняючи `m` від керування ресурсами та забезпечуючи ефективне присвоєння без зайвого копіювання;
- `Matrix& operator()(size_t rows, size_t columns, const double *const *const el)` – оператор встановлює задану кількість рядків `rows` та стовпців `columns` існуючій матриці з подальшим заповненням її елементів значеннями із двовимірного масиву `el`;
- `Matrix& operator()(size_t rows, size_t columns, double fill = 0.0)` – оператор встановлює задану кількість рядків `rows` та стовпців `columns` існуючій матриці з подальшим заповненням її елементів значенням змінної `fill`;
- `Matrix& operator()(const std::pair<size_t, size_t> &size, double fill = 0.0)` – оператор встановлює задану кількість рядків та стовпців існуючій матриці за допомогою пари `size` з подальшим заповненням її елементів значенням змінної `fill`;
- `Matrix& operator()(const std::initializer_list<const std::initializer_list<double>> &lists)` – оператор встановлює задану кількість рядків та стовпців існуючій матриці з подальшим заповненням її елементів на основі вкладених списків ініціалізаторів `lists`;
- `Matrix& operator()(const Matrix &m)` – оператор виконує копіювання вмісту матриці `m` у поточний об'єкт, замінюючи його дані;
- `double& operator[](size_t ind)` – оператор повертає посилання на елемент матриці за вказаним індексом `ind`;
- `friend Matrix operator+(const Matrix &m1, const Matrix &m2)` – оператор виконує поелементне додавання двох матриць `m1` та `m2`;

- `friend Matrix operator+(const Matrix &m, double d)` – оператор виконує поелементне додавання числа `d` до матриці `m`;
- `friend Matrix operator+(double d, const Matrix &m)` – оператор виконує поелементне додавання числа `d` до матриці `m`;
- `Matrix& operator+=(const Matrix &m)` – оператор виконує поелементне додавання матриці `m` до існуючої матриці;
- `Matrix& operator+=(double d)` – оператор виконує поелементне додавання числа `d` до існуючої матриці;
- `friend Matrix operator-(const Matrix &m1, const Matrix &m2)` – оператор виконує поелементне віднімання матриці `m2` від матриці `m1`;
- `friend Matrix operator-(const Matrix &m, double d)` – оператор виконує поелементне віднімання числа `d` від матриці `m`;
- `friend Matrix operator-(double d, const Matrix &m)` – оператор виконує поелементне віднімання матриці `m` від числа `d`;
- `Matrix& operator-=(const Matrix &m)` – оператор виконує поелементне віднімання матриці `m` від існуючої матриці;
- `Matrix& operator-=(double d)` – оператор виконує поелементне віднімання числа `d` від існуючої матриці;
- `Matrix operator-()` – оператор повертає матрицю, елементи якої є однаковими за абсолютним значенням з елементами існуючої матриці, але мають протилежний знак;
- `friend Matrix operator*(const Matrix &m1, const Matrix &m2)` – оператор виконує поелементне множення двох матриць `m1` та `m2`;
- `friend Matrix operator*(const Matrix &m, double d)` – оператор виконує поелементне множення матриці `m` на число `d`;
- `friend Matrix operator*(double d, const Matrix &m)` – оператор виконує поелементне множення числа `d` на матрицю `m`;

- `Matrix& operator*=(const Matrix &m)` – оператор виконує поелементне множення існуючої матриці на матрицю `m`;
- `Matrix& operator*=(double d)` – оператор виконує поелементне множення існуючої матриці на число `d`;
- `friend Matrix operator/(const Matrix &m1, const Matrix &m2)` – оператор виконує поелементне ділення матриці `m1` на матрицю `m2`;
- `friend Matrix operator/(const Matrix &m, double d)` – оператор виконує поелементне ділення матриці `m` на число `d`;
- `friend Matrix operator/(double d, const Matrix &m)` – оператор виконує поелементне ділення числа `d` на матрицю `m`;
- `Matrix& operator/=(const Matrix &m)` – оператор виконує поелементне ділення існуючої матриці на матрицю `m`;
- `Matrix& operator/=(double d)` – оператор виконує поелементне ділення існуючої матриці на число `d`;
- `friend bool operator==(const Matrix &m1, const Matrix &m2)` – оператор виконує перевірку на рівність матриць `m1` та `m2`;
- `friend bool operator!=(const Matrix &m1, const Matrix &m2)` – оператор виконує перевірку на нерівність матриць `m1` та `m2`;
- `friend Matrix operator&(const Matrix &m1, const Matrix &m2)` – оператор виконує поелементне логічне І між матрицями `m1` та `m2`;
- `friend Matrix operator&(const Matrix &m, double d)` – оператор виконує поелементне логічне І між матрицею `m` та числом `d`;
- `friend Matrix operator&(double d, const Matrix &m)` – оператор виконує поелементне логічне І між числом `d` та матрицею `m`;
- `Matrix& operator&=(const Matrix &m)` – оператор виконує поелементне логічне І між існуючою матрицею та матрицею `m`;

- `Matrix& operator&=(double d)` – оператор виконує поелементне логічне І між існуючою матрицею та числом `d`;
- `friend Matrix operator|(const Matrix &m1, const Matrix &m2)` – оператор виконує поелементне логічне АБО між матрицями `m1` та `m2`;
- `friend Matrix operator|(const Matrix &m, double d)` – оператор виконує поелементне логічне АБО між матрицею `m` та числом `d`;
- `friend Matrix operator|(double d, const Matrix &m)` – оператор виконує поелементне логічне АБО між числом `d` та матрицею `m`;
- `Matrix& operator|=(const Matrix &m)` – оператор виконує поелементне логічне АБО між існуючою матрицею та матрицею `m`;
- `Matrix& operator|=(double d)` – оператор виконує поелементне логічне АБО між існуючою матрицею та числом `d`;
- `Matrix operator~()` – оператор повертає матрицю, елементи якої утворені в результаті застосування логічного НЕ до елементів існуючої матриці;
- `friend std::ostream& operator<<(std::ostream &out, const Matrix &m)` – оператор виконує виведення значень елементів матриці `m` в консоль;
- `friend std::istream& operator>>(std::istream &in, Matrix &m)` – оператор виконує введення значень елементів матриці `m` з консолі;
- `friend std::ofstream& operator<<(std::ofstream &out, const Matrix &m)` – оператор виконує запис значень елементів матриці `m` в файл;

- `friend std::ifstream& operator>>(std::ifstream &in, Matrix &m)` – оператор виконує зчитування значень елементів матриці `m` з файлу.

Приклад 2:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3, заповнену 0
    Matrix A(3, 3);
    // вводимо значення елементів матриці A з консолі
    std::cout << "Enter values for matrix A:\n";
    std::cin >> A;
    // копіюємо значення змінної A в змінну B
    Matrix B = A;
    // змінюємо розмір матриці B на 3 x 4
    // та заповнюємо її значенням 5
    B(3, 4, 5);
    // змінюємо значення елемента матриці B
    // з індексами [1][2]
    B[1][2] = 9;
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Enter values for matrix A:
5 7 6
2 3 1
4 8 2
Matrix B:
5 5 5 5
5 5 9 5
5 5 5 5
```

Приклад 3:

```
#include <iostream>
#include <iomanip>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 4, заповнену 0
    Matrix A(3, 4);
    // створюємо матрицю B розміром 3 x 4, заповнену 0
    Matrix B(3, 4);
    // вводимо значення елементів матриці A з консолі
    std::cout << "Enter values for matrix A:\n";
    std::cin >> A;
    // вводимо значення елементів матриці B з консолі
    std::cout << "Enter values for matrix B:\n";
    std::cin >> B;
    // обчислюємо значення матриці C
    Matrix C = 5 * (A + B) / 2 - B / A;
    // віднімаємо 4 від значення кожного елемента
    // матриці C
    C -= 4;
    // виводимо значення елементів матриці C в консоль
    std::cout << std::fixed << std::setprecision(4) <<
    "Matrix C:\n" << C << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Enter values for matrix A:
7 6 2 9
3 8 4 1
9 6 7 2
Enter values for matrix B:
3 5 6 1
2 7 8 2
6 9 1 4
Matrix C:
20.5714 22.6667 13.0000 20.8889
 7.8333 32.6250 24.0000  1.5000
32.8333 32.0000 15.8571  9.0000
```

Приклад 4:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 0, 3 },
              { 4, 5, 6 } };
    // копіюємо значення змінної A в змінну B
    Matrix B = A;
    // перевіряємо на рівність матриці A та B
    if (A == B) std::cout << "equal" << std::endl;
    // збільшуємо значення всіх елементів матриці A на 2
    A += 2;
    // виконуємо поелементне логічне І між матрицями
    // A та НЕ B, після чого присвоюємо результат
    // в змінну C
    Matrix C = A & ~B;
    // виводимо значення елементів матриці C в консоль
    std::cout << "Matrix C:\n" << C << std::endl;
    return 0;
}
```

Результат роботи програми:

```
equal
Matrix C:
0 1 0
0 0 0
```

2.3. Методи класу Matrix

В класі `Matrix` реалізовані наступні методи для роботи з матрицями:

- `static std::pair<size_t, size_t> size(const Matrix &m)` – метод повертає розміри матриці `m` у вигляді пари [кількість рядків, кількість стовпців];

Приклад 5:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 0, 3 },
              { 4, 5, 6 } };
    // визначаємо розмір матриці A
    std::pair<size_t, size_t> s = Matrix::size(A);
    // виводимо кількість рядків матриці A в консоль
    std::cout << "rows:    " << s.first << std::endl;
    // виводимо кількість стовпців матриці A в консоль
    std::cout << "columns: " << s.second << std::endl;
    return 0;
}
```

Результат роботи програми:

```
rows:    2
columns: 3
```

- `static size_t size(const Matrix &m, unsigned int dim)` – метод повертає розмір матриці `m` за вказаною розмірністю `dim` (`dim = 1` – кількість рядків, `dim = 2` – кількість стовпців);

Приклад 6:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 0, 3 },
              { 4, 5, 6 } };
    // виводимо кількість рядків матриці A в консоль
    std::cout << "rows:    " << Matrix::size(A, 1) <<
```

```
    std::endl;
    // виводимо кількість стовпців матриці A в консоль
    std::cout << "columns: " << Matrix::size(A, 2) <<
    std::endl;
    return 0;
}
```

Результат роботи програми:

```
rows:    2
columns: 3
```

- `static Matrix find(const Matrix &m, double d, std::string cond = "==")` – метод виконує пошук елементів у матриці `m`, які відповідають заданій умові `cond` (`"=="`, `"!="`, `"<="`, `">="`, `"<"`, `">"`); значення кожного елемента матриці `m` порівнюється зі значенням параметра `d` відповідно до заданої умови `cond`; метод повертає матрицю такого ж розміру, як і початкова матриця `m`, що складається з 0 та 1 (0 – елемент не відповідає умові, 1 – елемент відповідає умові);

Приклад 7:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 7, 3 },
              { 4, 5, 6 } };
    // знаходимо елементи в матриці A,
    // значення яких менше або дорівнює 5
    // та виводимо результат пошуку в консоль
    std::cout << Matrix::find(A, 5, "<=") << std::endl;
    return 0;
}
```

Результат роботи програми:

```
1 0 1
1 1 0
```

- `static std::vector<std::pair<size_t, size_t>> find_index(const Matrix &m, double d, std::string cond = "==")` – метод виконує пошук елементів у матриці `m`, які відповідають заданій умові `cond` (`"=="`, `"!="`, `"<="`, `">="`, `"<"`, `">"`); значення кожного елемента матриці `m` порівнюється зі значенням параметра `d` відповідно до заданої умови `cond`; метод повертає вектор, що містить пари індексів елементів матриці `m`, які відповідають заданій умові `cond`;

Приклад 8:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 7, 3 },
              { 4, 5, 6 } };
    // знаходимо елементи в матриці A,
    // значення яких більше 3 та зберігаємо їх індекси
    auto index = Matrix::find_index(A, 3, ">");
    // виводимо результат пошуку в консоль
    for (const auto &i : index)
    {
        std::cout << '[' << i.first << ',' <<
        i.second << ']' << std::endl;
    }
    return 0;
}
```

Результат роботи програми:

```
[0,1]
[1,0]
[1,1]
[1,2]
```

- `static Matrix iszeros(const Matrix &m)` – метод виконує пошук елементів у матриці `m`, значення яких дорівнює 0; метод повертає матрицю такого ж розміру, як і початкова матриця `m`, що складається з 0 та 1 (0 – елемент не відповідає умові, 1 – елемент відповідає умові);

Приклад 9:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 0, -3 },
              { 4, 5,  0 } };
    // знаходимо елементи в матриці A,
    // значення яких дорівнює 0 та
    // виводимо результат пошуку в консоль
    std::cout << Matrix::iszeros(A) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
0 1 0
0 0 1
```

- `static Matrix transpose(const Matrix &m)` – метод повертає транспоновану матрицю до початкової матриці `m`;

Приклад 10:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 0, -3 },
              { 4, 5,  0 } };
    // створюємо матрицю B, транспоновану до матриці A
    Matrix B = Matrix::transpose(A);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
 1 4
 0 5
-3 0
```

- `static double sum(const Matrix &m)` – метод виконує підсумовування значень всіх елементів матриці `m`;

Приклад 11:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // обчислюємо суму значень всіх елементів
    // матриці A та виводимо суму в консоль
    std::cout << "Sum of all elements of matrix A: ";
    std::cout << Matrix::sum(A) << std::endl;
```

```
    return 0;
}
```

Результат роботи програми:

```
Sum of all elements of matrix A: 15
```

- `static Matrix sum(const Matrix &m, unsigned int dim)` – метод виконує підсумовування значень елементів матриці `m` за вказаною розмірністю `dim` (`dim = 1` – сумування за рядками, `dim = 2` – сумування за стовпцями);

Приклад 12:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // обчислюємо суму значень елементів в кожному
    // стовпці матриці A та виводимо суму в консоль
    std::cout << "Sum of elements in each column of
    matrix A:" << std::endl;
    std::cout << Matrix::sum(A, 1) << std::endl;
    // обчислюємо суму значень елементів в кожному
    // рядку матриці A та виводимо суму в консоль
    std::cout << "Sum of elements in each row of
    matrix A:" << std::endl;
    std::cout << Matrix::sum(A, 2) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Sum of elements in each column of matrix A:
5 11 -1

Sum of elements in each row of matrix A:
 4
11
```

- `static double prod(const Matrix &m)` – метод виконує множення значень всіх елементів матриці `m`;

Приклад 13:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // обчислюємо добуток значень всіх елементів
    // матриці A та виводимо добуток в консоль
    std::cout << "Product of all elements of matrix A: ";
    std::cout << Matrix::prod(A) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Product of all elements of matrix A: -720
```

- `static Matrix prod(const Matrix &m, unsigned int dim)` – метод виконує множення значень елементів матриці `m` за вказаною розмірністю `dim` (`dim = 1` – множення за рядками, `dim = 2` – множення за стовпцями);

Приклад 14:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // обчислюємо добуток значень елементів в кожному
    // стовпці матриці A та виводимо добуток в консоль
    std::cout << "Product of elements in each column of
    matrix A:" << std::endl;
    std::cout << Matrix::prod(A, 1) << std::endl;
    // обчислюємо добуток значень елементів в кожному
    // рядку матриці A та виводимо добуток в консоль
    std::cout << "Product of elements in each row of
    matrix A:" << std::endl;
    std::cout << Matrix::prod(A, 2) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Product of elements in each column of matrix A:
4 30 -6

Product of elements in each row of matrix A:
-18
 40
```

- `static Matrix eye(size_t size)` – метод повертає одиничну матрицю заданого розміру `size`;

Приклад 15:

```
#include <iostream>
#include "Matrix.h"

int main()
```

```
{
    // створюємо одиничну матрицю A розміром 5 x 5
    Matrix A = Matrix::eye(5);
    // виводимо значення елементів матриці A в консоль
    std::cout << "Matrix A:\n" << A << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix A:
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

- `static Matrix fliplr(const Matrix &m)` – метод формує матрицю, переставляючи стовпці початкової матриці `m` відносно вертикальної осі;

Приклад 16:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // формуємо матрицю B, переставляючи стовпці
    // матриці A відносно вертикальної осі
    Matrix B = Matrix::fliplr(A);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
-3 6 1
 2 5 4
```

- `static Matrix flipud(const Matrix &m)` – метод формує матрицю, переставляючи рядки початкової матриці `m` відносно горизонтальної осі;

Приклад 17:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // формуємо матрицю B, переставляючи рядки
    // матриці A відносно горизонтальної осі
    Matrix B = Matrix::flipud(A);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
4  5  2
1  6 -3
```

- `static Matrix rot90(const Matrix &m)` – метод формує матрицю шляхом «повороту» початкової матриці `m` на 90 градусів проти годинникової стрілки;

Приклад 18:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // формуємо матрицю B шляхом «повороту»
    // матриці A на 90 градусів проти
    // годинникової стрілки
    Matrix B = Matrix::rot90(A);
    // виводимо значення елементів
    // матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
-3 2
 6 5
 1 4
```

- `static Matrix reshape(const Matrix &m, size_t rows, size_t columns)` – метод формує матрицю із заданою кількістю рядків `rows` та стовпців `columns`, що складається з елементів початкової матриці `m`; кількість елементів нової матриці повинна співпадати з кількістю елементів матриці `m`;

Приклад 19:

```
#include <iostream>
#include "Matrix.h"

int main()
{
```

```
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // створюємо матрицю B розміром 3 x 2,
    // що складається з елементів матриці A
    Matrix B = Matrix::reshape(A, 3, 2);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
1  5
4 -3
6  2
```

- `static Matrix mult(const Matrix &m1, const Matrix &m2)` – метод знаходить добуток матриць `m1` та `m2`; якщо матриця `m1` є матрицею розміром $s_1 \times s_2$, а `m2` є матрицею розміром $s_2 \times s_3$, то результуюча матриця матиме розмір $s_1 \times s_3$;

Приклад 20:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 6, -3 },
              { 4, 5,  2 } };
    // створюємо матрицю B розміром 3 x 4,
    Matrix B{ { 4, 2, 8, 9 },
              { 6, 1, 3, 1 },
              { 8, 3, 4, 2 } };
    // знаходимо добуток матриць A та B,
    // та присвоюємо його в змінну C
```

```
    Matrix C = Matrix::mult(A, B);
    // виводимо значення елементів матриці C в консоль
    std::cout << "Matrix C:\n" << C << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix C:
16 -1 14  9
62 19 55 45
```

- `static Matrix diag(const Matrix &m)` – метод формує діагональну матрицю, яка утворена діагональними елементами початкової матриці `m`;

Приклад 21:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 },
              { 8, 3, 5 } };
    // формуємо діагональну матрицю B із
    // діагональних елементів матриці A
    Matrix B = Matrix::diag(A);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Diagonal matrix B:" << std::endl;
    std::cout << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Diagonal matrix B:
4 0 0
0 1 0
0 0 5
```

- `static double det(const Matrix &m)` – метод знаходить визначник матриці `m`;

Приклад 22:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 },
              { 8, 3, 5 } };
    // знаходимо визначник матриці A
    // та виводимо його значення в консоль
    std::cout << "Determinant of matrix A: ";
    std::cout << Matrix::det(A) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Determinant of matrix A: 52
```

- `static Matrix inv(const Matrix &m)` – метод знаходить зворотну матрицю до квадратної матриці `m`;

Приклад 23:

```
#include <iostream>
#include "Matrix.h"
```

```
int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 },
              { 8, 3, 5 } };
    // знаходимо зворотну матрицю до матриці A
    Matrix B = Matrix::inv(A);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Inverse matrix B:" << std::endl;
    std::cout << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Inverse matrix B:
-0.076923  0.269231 -0.038462
-0.115385 -0.846154  0.692308
 0.192308  0.076923 -0.153846
```

- `static Matrix solve(const Matrix &A, const Matrix &B)` – метод розв'язує систему лінійних алгебраїчних рівнянь $Ax = b$ відносно x, у вигляді $x = A^{-1}b$; `A` – квадратна матриця розміром $n \times n$, `B` – вектор стовпець розміром $n \times 1$;

Приклад 24:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1,  2, -1 },
              { 2, -3,  2 },
              { 3,  1,  1 } };
    // створюємо вектор-стовпець B розміром 3 x 1
```

```
    Matrix b = { { 2 },
                 { 2 },
                 { 8 } };
    // розв'язуємо систему лінійних алгебраїчних рівнянь
    Matrix x = Matrix::solve(A, b);
    // виводимо значення елементів вектора x в консоль
    std::cout << "Solution of the system of linear
    algebraic equations:" << std::endl;
    std::cout << x << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Solution of the system of linear algebraic equations:
1
2
3
```

- `static Matrix tril(const Matrix &m, int k = 0)` – `L = tril(m)` метод формує матрицю, яка складається з елементів нижньої трикутної частини початкової матриці `m`; `L = tril(m, k)` метод формує матрицю, яка складається з елементів початкової матриці `m` на *k*-й діагоналі та нижче *k*-ої діагоналі;

Приклад 25:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1,  2, -1 },
              { 2, -3,  2 },
              { 3,  1,  1 } };
    // формуємо матрицю L, яка складається з елементів
    // нижньої трикутної частини матриці A
    Matrix L = Matrix::tril(A);
```

```
    // виводимо значення елементів матриці L в консоль
    std::cout << "Lower triangular matrix L:\n";
    std::cout << L << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Lower triangular matrix L:
1  0  0
2 -3  0
3  1  1
```

- `static Matrix triu(const Matrix &m, int k = 0)` – `U = triu(m)` метод формує матрицю, яка складається з елементів верхньої трикутної частини початкової матриці `m`; `U = triu(m, k)` метод формує матрицю, яка складається з елементів початкової матриці `m` на *k*-й діагоналі та вище *k*-ої діагоналі;

Приклад 26:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1,  2, -1 },
              { 2, -3,  2 },
              { 3,  1,  1 } };
    // формуємо матрицю U, яка складається з елементів
    // верхньої трикутної частини матриці A
    Matrix U = Matrix::triu(A);
    // виводимо значення елементів матриці U в консоль
    std::cout << "Upper triangular matrix U:\n";
    std::cout << U << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Upper triangular matrix U:
1  2 -1
0 -3  2
0  0  1
```

- `static bool istril(const Matrix &m)` – метод визначає, чи є початкова матриця `m` нижньою трикутною матрицею;

Приклад 27:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1,  2, -1 },
              { 2, -3,  2 },
              { 3,  1,  1 } };
    // формуємо матрицю L, яка складається з елементів
    // нижньої трикутної частини матриці A
    Matrix L = Matrix::tril(A);
    // визначаємо, чи є матриця L нижньою трикутною
    // матрицею та виводимо результат в консоль
    if (Matrix::istril(L))
        std::cout << "L is lower triangular matrix" <<
        std::endl;
    else
        std::cout << "L is not lower triangular matrix"
        << std::endl;
    return 0;
}
```

Результат роботи програми:

```
L is lower triangular matrix
```

- `static bool istriu(const Matrix &m)` – метод визначає, чи є початкова матриця `m` верхньою трикутною матрицею;

Приклад 28:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1,  2, -1 },
              { 2, -3,  2 },
              { 3,  1,  1 } };
    // формуємо матрицю U, яка складається з елементів
    // верхньої трикутної частини матриці A
    Matrix U = Matrix::triu(A);
    // визначаємо, чи є матриця U верхньою трикутною
    // матрицею та виводимо результат в консоль
    if (Matrix::istriu(U))
        std::cout << "U is upper triangular matrix" <<
        std::endl;
    else
        std::cout << "U is not upper triangular matrix"
        << std::endl;
    return 0;
}
```

Результат роботи програми:

```
U is upper triangular matrix
```

- `static bool isdiag(const Matrix &m)` – метод визначає, чи є матриця `m` діагональною;

Приклад 29:

```
#include <iostream>
#include "Matrix.h"
```

```
int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 },
              { 8, 3, 5 } };
    // формуємо діагональну матрицю B із
    // діагональних елементів матриці A
    Matrix B = Matrix::diag(A);
    // визначаємо, чи є матриця B діагональною
    // та виводимо результат в консоль
    if (Matrix::isdiag(B))
        std::cout << "B is diagonal matrix" << std::endl;
    else
        std::cout << "B is not diagonal matrix" <<
        std::endl;
    return 0;
}
```

Результат роботи програми:

```
B is diagonal matrix
```

- `static Matrix cat(const Matrix &m1, const Matrix &m2, unsigned int dim = 1)` – метод здійснює об'єднання матриць `m1` та `m2` вздовж заданої розмірності `dim`; `dim = 1` – якщо матриця `m1` є матрицею розміром $s_1 \times s_2$, а `m2` є матрицею розміром $s_3 \times s_2$, то результуюча матриця матиме розмір $(s_1 + s_3) \times s_2$; `dim = 2` – якщо матриця `m1` є матрицею розміром $s_1 \times s_2$, а `m2` є матрицею розміром $s_1 \times s_3$, то результуюча матриця матиме розмір $s_1 \times (s_2 + s_3)$;

Приклад 30:

```
#include <iostream>
#include "Matrix.h"

int main()
```

```
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 },
              { 8, 3, 5 } };
    // створюємо матрицю B розміром 3 x 2
    Matrix B{ { 5, 1 },
              { 8, 4 },
              { 9, 2 } };
    // створюємо матрицю C розміром 3 x 5
    Matrix C = Matrix::cat(A, B, 2);
    // виводимо значення елементів матриці C в консоль
    std::cout << "Matrix C:\n" << C << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix C:
4 2 8 5 1
6 1 3 8 4
8 3 5 9 2
```

- `static Matrix repmat(const Matrix &matrix, size_t m, size_t n)` – метод формує матрицю, що містить `m` × `n` копій початкової матриці `matrix` у першому та другому вимірах відповідно;

Приклад 31:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 } };
    // створюємо матрицю B, що містить
    // 4 x 5 копій матриці A у першому
    // та другому вимірах відповідно
```

```
    Matrix B = Matrix::repmat(A, 4, 5);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
4 2 8 4 2 8 4 2 8 4 2 8 4 2 8
6 1 3 6 1 3 6 1 3 6 1 3 6 1 3
4 2 8 4 2 8 4 2 8 4 2 8 4 2 8
6 1 3 6 1 3 6 1 3 6 1 3 6 1 3
4 2 8 4 2 8 4 2 8 4 2 8 4 2 8
6 1 3 6 1 3 6 1 3 6 1 3 6 1 3
4 2 8 4 2 8 4 2 8 4 2 8 4 2 8
6 1 3 6 1 3 6 1 3 6 1 3 6 1 3
```

- `static double maxel(const Matrix &m)` – метод виконує пошук максимального елемента в матриці `m`;

Приклад 32:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 } };
    // виконуємо пошук максимального елемента
    // в матриці A та виводимо його в консоль
    std::cout << "Maximum element of matrix A: ";
    std::cout << Matrix::maxel(A) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Maximum element of matrix A: 8
```

- `static Matrix maxel(const Matrix &m, unsigned int dim)` – метод виконує пошук максимальних елементів в матриці `m` за вказаною розмірністю `dim` (`dim = 1` – пошук максимального елемента в кожному стовпці, `dim = 2` – пошук максимального елемента в кожному рядку);

Приклад 33:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 } };
    // виконуємо пошук максимального елемента
    // в кожному стовпці матриці A та
    // виводимо результат пошуку в консоль
    std::cout << "Maximum element in each column of
    matrix A:" << std::endl;
    std::cout << Matrix::maxel(A, 1) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Maximum element in each column of matrix A:
6 2 8
```

- `static double minel(const Matrix &m)` – метод виконує пошук мінімального елемента в матриці `m`;

Приклад 34:

```
#include <iostream>
#include "Matrix.h"

int main()
```

```
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 } };
    // виконуємо пошук мінімального елемента
    // в матриці A та виводимо його в консоль
    std::cout << "Minimum element of matrix A: ";
    std::cout << Matrix::minel(A) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Minimum element of matrix A: 1
```

- `static Matrix minel(const Matrix &m, unsigned int dim)` – метод виконує пошук мінімальних елементів в матриці `m` за вказаною розмірністю `dim` (`dim = 1` – пошук мінімального елемента в кожному стовпці, `dim = 2` – пошук мінімального елемента в кожному рядку);

Приклад 35:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 } };
    // виконуємо пошук мінімального елемента
    // в кожному рядку матриці A та
    // виводимо результат пошуку в консоль
    std::cout << "Minimum elements in each row of
    matrix A: " << std::endl;
    std::cout << Matrix::minel(A, 2) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Minimum elements in each row of matrix A:
2
1
```

- `static Matrix range(double start, double stop, double step = 1.0)` – метод формує вектор-рядок, що складається з послідовності чисел від `start` до `stop` з кроком `step`;

Приклад 36:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // формуємо вектор-рядок, що складається
    // з послідовності чисел від -2 до 2 з кроком 0.1;
    Matrix x = Matrix::range(-2, 2, 0.1);
    // виводимо значення елементів
    // вектора-рядка x в консоль
    std::cout << "Range of values from -2 to 2 with a
    step of 0.1:" << std::endl;
    std::cout << x << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Range of values from -2 to 2 with a step of 0.1:
-2.0 -1.9 -1.8 -1.7 -1.6 -1.5 -1.4 -1.3 -1.2 -1.1 -1.0
-0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1
 0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.0  1.1  1.2
 1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0
```

- `static Matrix linspace(double start, double stop, size_t number = 100)` – метод формує вектор-рядок, що складається з `number` елементів від `start` до `stop`;

Приклад 37:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // формуємо вектор-рядок, що складається
    // з 10 елементів від 1 до 5;
    Matrix x = Matrix::linspace(1, 5, 10);
    // виводимо значення елементів
    // вектора-рядка x в консоль
    std::cout << "The values of the elements of the
    vector x:" << std::endl;
    std::cout << x << std::endl;
    return 0;
}
```

Результат роботи програми:

```
The values of the elements of the vector x:
1 1.44444 1.88889 2.33333 2.77778 3.22222 3.66667 4.11111
4.55556 5
```

- `static Matrix besselj(int nu, const Matrix &m)` – метод обчислює функцію Бесселя першого роду для кожного елемента матриці `m`; `nu` – ціле число, яке задає порядок функції Бесселя першого роду;
- `static double besselj(int nu, double z)` – метод обчислює функцію Бесселя першого роду від числа `z`; `nu` – ціле число, яке задає порядок функції Бесселя першого роду;
- `static Matrix bessely(int nu, const Matrix &m)` – метод обчислює функцію Бесселя другого роду для кожного елемента матриці `m`; `nu` – ціле число, яке задає порядок функції Бесселя другого роду;
- `static double bessely(int nu, double z)` – метод обчислює функцію Бесселя другого роду від числа `z`; `nu` – ціле число, яке задає порядок функції Бесселя другого роду;

- `static std::pair<Matrix, Matrix> besselh(int nu, unsigned int k, const Matrix &m)` – метод обчислює функцію Бесселя третього роду (функцію Ханкеля) для кожного елемента матриці `m`; `nu` – ціле число, яке задає порядок функції Ханкеля; `k` – ціле число, яке задає тип функції Ханкеля (якщо `k = 1`, то `besselh` обчислює функцію Ханкеля першого роду; якщо `k = 2`, то `besselh` обчислює функцію Ханкеля другого роду); `besselh` повертає результат обчислення функції Ханкеля у вигляді пари $[J_v(m), Y_v(m)]$, де $J_v(m)$ – дійсна частина, $Y_v(m)$ – уявна частина;

Приклад 38:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 } };
    // задаємо порядок функції Ханкеля
    int nu = 0;
    // задаємо тип функції Ханкеля
    unsigned int k = 2;
    // обчислюємо фукцію Ханкеля другого роду порядку 0
    // для кожного елемента матриці A
    std::pair<Matrix, Matrix> H =
    Matrix::besselh(nu, k, A);
    // виводимо значення фукції Ханкеля для
    // кожного елемента матриці A в консоль
    std::cout << "Hankel function values for each element
    of matrix A:\n";
    for (int i = 0; i < Matrix::size(A, 1); ++i)
    {
        for (int j = 0; j < Matrix::size(A, 2); ++j)
        {
            std::cout << '(' << H.first[i][j] << ',' <<
            H.second[i][j] << ')' << '\t';
```

```
        }
        std::cout << '\n';
    }
    return 0;
}
```

Результат роботи програми:

```
Hankel function values for each element of matrix A:
(-0.39715,0.01694) (0.22389,-0.51037)  (0.17165,-0.22352)
 (0.15064,0.28819) (0.76519,-0.08825) (-0.26005,-0.37685)
```

- `static std::pair<double, double> besselh(int nu, unsigned int k, double z)` – метод обчислює функцію Бесселя третього роду (функцію Ханкеля) від числа `z`; `nu` – ціле число, яке задає порядок функції Ханкеля; `k` – ціле число, яке задає тип функції Ханкеля (якщо `k = 1`, то `besselh` обчислює функцію Ханкеля першого роду; якщо `k = 2`, то `besselh` обчислює функцію Ханкеля другого роду); `besselh` повертає результат обчислення функції Ханкеля у вигляді пари $[J_\nu(z), Y_\nu(z)]$, де $J_\nu(z)$ – дійсна частина, $Y_\nu(z)$ – уявна частина;
- `static Matrix abs(const Matrix &m)` – метод обчислює абсолютне значення для кожного елемента матриці `m`;
- `static Matrix exp(const Matrix &m)` – метод обчислює експоненту піднесену до заданого степеня e^n (n – елемент матриці) для кожного елемента матриці `m`;
- `static Matrix exp2(const Matrix &m)` – метод обчислює 2^n (n – елемент матриці) для кожного елемента матриці `m`;
- `static Matrix expm1(const Matrix &m)` – метод обчислює експоненту піднесену до заданого степеня мінус 1.0, а саме $e^n - 1$ (n – елемент матриці) для кожного елемента матриці `m`;
- `static Matrix log(const Matrix &m)` – метод обчислює натуральний (за основою e) логарифм для кожного елемента матриці `m`;

- `static Matrix log10(const Matrix &m)` – метод обчислює десятковий логарифм для кожного елемента матриці `m`;
- `static Matrix log2(const Matrix &m)` – метод обчислює двійковий (за основою 2) логарифм для кожного елемента матриці `m`;
- `static Matrix log1p(const Matrix &m)` – метод обчислює натуральний (за основою e) логарифм числа $1+n$ (n – елемент матриці) для кожного елемента матриці `m`;
- `static Matrix pow(const Matrix &m, double p)` – метод здійснює піднесення елементів матриці `m` до заданого степеня `p`;
- `static Matrix pow(double n, const Matrix &p)` – метод здійснює піднесення числа `n` до степенів, які задаються елементами матриці `p`;
- `static Matrix pow(const Matrix &m, const Matrix &p)` – метод здійснює піднесення елементів матриці `m` до заданих степенів, які задаються елементами матриці `p`;
- `static Matrix sqrt(const Matrix &m)` – метод обчислює квадратний корінь із кожного елемента матриці `m`;
- `static Matrix cbrt(const Matrix &m)` – метод обчислює кубічний корінь із кожного елемента матриці `m`;
- `static Matrix hypot(const Matrix &x, const Matrix &y)` – метод обчислює квадратний корінь із суми квадратів a та b, а саме $\sqrt{a^2+b^2}$ (a – елементи матриці `x`, b – елементи матриці `y`) для кожного елемента матриць `x` та `y`;
- `static Matrix sin(const Matrix &m)` – метод обчислює синус від кожного елемента матриці `m` в радіанах;
- `static Matrix cos(const Matrix &m)` – метод обчислює косинус від кожного елемента матриці `m` в радіанах;

- `static Matrix tan(const Matrix &m)` – метод обчислює тангенс від кожного елемента матриці `m` в радіанах;
- `static Matrix asin(const Matrix &m)` – метод обчислює головне значення арксинусу від кожного елемента матриці `m`;
- `static Matrix acos(const Matrix &m)` – метод обчислює головне значення арккосинусу від кожного елемента матриці `m`;
- `static Matrix atan(const Matrix &m)` – метод обчислює головне значення арктангенсу від кожного елемента матриці `m`;
- `static Matrix atan2(const Matrix &y, const Matrix &x)` – метод поелементно обчислює арктангенс a/b (a – елементи матриці `y`, b – елементи матриці `x`), використовуючи знаки елементів матриць `y` та `x` для визначення правильного квадранта;
- `static Matrix sinh(const Matrix &m)` – метод обчислює гіперболічний синус від кожного елемента матриці `m`;
- `static Matrix cosh(const Matrix &m)` – метод обчислює гіперболічний косинус від кожного елемента матриці `m`;
- `static Matrix tanh(const Matrix &m)` – метод обчислює гіперболічний тангенс від кожного елемента матриці `m`;
- `static Matrix asinh(const Matrix &m)` – метод обчислює зворотний гіперболічний синус від кожного елемента матриці `m`;
- `static Matrix acosh(const Matrix &m)` – метод обчислює зворотний гіперболічний косинус від кожного елемента матриці `m`;
- `static Matrix atanh(const Matrix &m)` – метод обчислює зворотний гіперболічний тангенс від кожного елемента матриці `m`;
- `static Matrix erf(const Matrix &m)` – метод обчислює функцію помилок $\operatorname{erf}(n)$ (n – елемент матриці) від кожного елемента матриці `m`;

- `static Matrix erfc(const Matrix &m)` – метод обчислює додаткову функцію помилок $1 - \text{erf}(n)$ (n – елемент матриці) від кожного елемента матриці `m`;
- `static Matrix tgamma(const Matrix &m)` – метод обчислює гамма-функцію від кожного елемента матриці `m`;
- `static Matrix lgamma(const Matrix &m)` – метод обчислює натуральний логарифм абсолютного значення гамма-функції від кожного елемента матриці `m`;
- `static Matrix sigmoid(const Matrix& m)` – метод обчислює сигмоїдальну функцію від кожного елемента матриці `m`;
- `static Matrix ceil(const Matrix &m)` – метод обчислює найменше ціле значення не менше за аргумент від кожного елемента матриці `m`;
- `static Matrix floor(const Matrix &m)` – метод обчислює найбільше ціле значення, що не перевищує аргумент від кожного елемента матриці `m`;
- `static Matrix trunc(const Matrix &m)` – метод обчислює найближче ціле число, величина якого не перевищує аргумент від кожного елемента матриці `m`;
- `static Matrix round(const Matrix &m)` – метод обчислює ціле число, найближче до аргументу, округляючи значення наполовину від нуля, незалежно від поточного режиму округлення від кожного елемента матриці `m`;
- `static Matrix nearbyint(const Matrix &m)` – метод округляє кожний елемент матриці `m` до цілого значення у форматі з плаваючою комою, використовуючи поточний режим округлення;

- `static Matrix rint(const Matrix &m)` – метод округляє кожний елемент матриці `m` до цілого значення у форматі з плаваючою комою, використовуючи поточний режим округлення;
- `static Matrix isfinite(const Matrix &m)` – метод поелементно визначає, чи має елемент матриці `m` скінченне значення, тобто нормальне, субнормальне чи нульове, але не нескінченне чи NaN;
- `static Matrix isinf(const Matrix &m)` – метод поелементно визначає, чи є елемент матриці `m` позитивною чи негативною нескінченністю;
- `static Matrix isnan(const Matrix &m)` – метод поелементно визначає, чи є елемент матриці `m` NaN;
- `static Matrix isnormal(const Matrix &m)` – метод поелементно визначає, чи є елемент матриці `m` нормальним, тобто чи не є він нулем, субнормальним, нескінченним або NaN;

Приклад 39:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 4, 2, 8 },
              { 6, 1, 3 } };
    // виконуємо піднесення елементів матриці
    // A до степеня 3 та зберігаємо отриману
    // матрицю в змінній B
    Matrix B = Matrix::pow(A, 3);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
 64   8  512
216   1   27
```

- `static std::pair<Matrix, Matrix> meshgrid(const Matrix &x, const Matrix &y)` – метод повертає координати двовимірної сітки в форматі [X, Y] на основі координат, що містяться у векторах `x` та `y`; X – це матриця, в якій кожен рядок є копією `x`, а Y – це матриця, в якій кожен стовпець є копією `y`;
- `static std::pair<Matrix, Matrix> meshgrid(const Matrix &x)` – метод повертає координати двовимірної сітки в форматі [X, Y] на основі координат, що містяться у векторі `x`. X – це матриця, в якій кожен рядок є копією `x`, а Y – це матриця, в якій кожен стовпець є копією `transpose(x)`;

Приклад 40:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // формуємо вектор-рядок, що складається
    // з послідовності чисел від -5 до 5 з кроком 1
    Matrix x = Matrix::range(-5, 5);
    // формуємо вектор-рядок, що складається
    // з послідовності чисел від 0 до 9 з кроком 1
    Matrix y = Matrix::range(0, 9);
    // формуємо координати двовимірної сітки
    // в форматі [X, Y] на основі координат,
    // що містяться у векторах x та y
    std::pair<Matrix, Matrix> coord =
    Matrix::meshgrid(x, y);
    // виводимо значення елементів матриці X в консоль
```

```
    std::cout << "X:\n" << coord.first << std::endl;
    // виводимо значення елементів матриці Y в консоль
    std::cout << "Y:\n" << coord.second << std::endl;
    return 0;
}
```

Результат роботи програми:

```
X:
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5
-5 -4 -3 -2 -1 0 1 2 3 4 5

Y:
0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9 9
```

- `static Matrix diff(const Matrix &m, unsigned int dim = 1)` – метод обчислює різницю між значеннями сусідніх елементів матриці `m` за вказаною розмірністю `dim` (`dim = 1` – різниця по рядках, `dim = 2` – різниця по стовпцях);

Приклад 41:

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <iomanip>
#include "Matrix.h"

int main()
{
    // оголошуємо крок за координатою
    double h = 0.1;
    // формуємо вектор-рядок, що складається
    // з послідовності чисел від -pi до pi з кроком h
    Matrix x = Matrix::range(-M_PI, M_PI, h);
    // обчислюємо синус від кожного елемента вектора x
    Matrix f = Matrix::sin(x);
    // обчислюємо похідну від sin(x)
    Matrix df = Matrix::diff(f, 2) / h;
    // виводимо значення елементів вектора df в консоль
    std::cout << std::fixed << std::setprecision(4) <<
    df << std::endl;
    return 0;
}
```

Результат роботи програми:

```
-0.9983 -0.9884 -0.9685 -0.9390 -0.9001 -0.8522 -0.7958
-0.7314 -0.6597 -0.5814 -0.4974 -0.4083 -0.3152 -0.2189
-0.1205 -0.0208  0.0791  0.1782  0.2755  0.3700  0.4609
 0.5471  0.6279  0.7024  0.7699  0.8297  0.8812  0.9239
 0.9574  0.9813  0.9954  0.9995  0.9937  0.9780  0.9524
 0.9174  0.8732  0.8202  0.7591  0.6904  0.6147  0.5330
 0.4459  0.3544  0.2593  0.1616  0.0623 -0.0376 -0.1371
-0.2353 -0.3311 -0.4236 -0.5119 -0.5950 -0.6722 -0.7427
-0.8058 -0.8608 -0.9073 -0.9446 -0.9725 -0.9907
```

- `static Matrix gradient(const Matrix &m, unsigned int dim = 1, double h = 1.0)` – метод обчислює градієнт матриці `m` за вказаною розмірністю `dim` (`dim = 1` – градієнт по рядках, `dim = 2` – градієнт по стовпцях); `h` – відстань між вузлами;

Приклад 42:

```
#include <iostream>
#include <iomanip>
#include "Matrix.h"

int main()
{
    // формуємо вектор-рядок, що складається
    // з послідовності чисел від -2 до 2 з кроком 1
    Matrix x = Matrix::range(-2, 2);
    // формуємо координати двовимірної сітки
    // в форматі [X, Y] на основі значень вектору x
    std::pair<Matrix, Matrix> coord =
    Matrix::meshgrid(x);
    // обчислюємо відстань між вузлами
    Matrix r = Matrix::hypot(coord.first, coord.second);
    // обчислюємо Гаусову функцію
    Matrix f = Matrix::exp(-Matrix::pow(r, 2));
    // обчислюємо градієнт f по стовпцях
    Matrix fx = Matrix::gradient(f, 2);
    // обчислюємо градієнт f по рядках
    Matrix fy = Matrix::gradient(f, 1);
    // виводимо значення елементів матриці fx в консоль
    std::cout << "fx:\n" << std::fixed <<
    std::setprecision(12) << fx << std::endl;
    // виводимо значення елементів матриці fy в консоль
    std::cout << "fy:\n" << std::fixed <<
    std::setprecision(12) << fy << std::endl;
    return 0;
}
```

Результат роботи програми:

```
fx:
 0.006402484371  0.008990088130  0.000000000000 -0.008990088130 -0.006402484371
 0.128597336238  0.180570747086  0.000000000000 -0.180570747086 -0.128597336238
 0.349563802283  0.490842180556  0.000000000000 -0.490842180556 -0.349563802283
 0.128597336238  0.180570747086  0.000000000000 -0.180570747086 -0.128597336238
 0.006402484371  0.008990088130  0.000000000000 -0.008990088130 -0.006402484371

fy:
 0.006402484371  0.128597336238  0.349563802283  0.128597336238  0.006402484371
 0.008990088130  0.180570747086  0.490842180556  0.180570747086  0.008990088130
 0.000000000000  0.000000000000  0.000000000000  0.000000000000  0.000000000000
-0.008990088130 -0.180570747086 -0.490842180556 -0.180570747086 -0.008990088130
-0.006402484371 -0.128597336238 -0.349563802283 -0.128597336238 -0.006402484371
```

- `static Matrix laplace(const Matrix &m, double hx = 1.0, double hy = 1.0)` – метод обчислює чисельно оператор Лапласа від заданої матриці `m`; `hx` – відстань між вузлами в напрямку *x*, `hy` – відстань між вузлами в напрямку *y*;

Приклад 43:

```
#include <iostream>
#include <iomanip>
#include "Matrix.h"

int main()
{
    // формуємо вектор-рядок, що складається
    // з послідовності чисел від -2 до 2 з кроком 1
    Matrix x = Matrix::range(-2, 2);
    // формуємо координати двовимірної сітки
    // в форматі [X, Y] на основі координат,
    // що містяться у векторі x
    std::pair<Matrix, Matrix> coord =
    Matrix::meshgrid(x);
    // обчислюємо відстань між вузлами
    Matrix r = Matrix::hypot(coord.first, coord.second);
    // обчислюємо Гаусову функцію
    Matrix f = Matrix::exp(-Matrix::pow(r, 2));
    // обчислюємо оператор Лапласа
    Matrix df = Matrix::laplace(f);
    // виводимо значення елементів матриці df в консоль
    std::cout << std::fixed << std::setprecision(5) <<
    df << std::endl;
    return 0;
}
```

Результат роботи програми:

```
0.00518  0.04877  0.13229  0.04877  0.00518
0.04877 -0.12860 -0.35535 -0.12860  0.04877
0.13229 -0.35535 -0.98168 -0.35535  0.13229
0.04877 -0.12860 -0.35535 -0.12860  0.04877
0.00518  0.04877  0.13229  0.04877  0.00518
```

- `static Matrix cumsum(const Matrix& m, unsigned int dim = 1)` – метод обчислює кумулятивну суму елементів матриці `m` за вказаною розмірністю `dim` (`dim = 1` – сума за рядками, `dim = 2` – сума за стовпцями);

Приклад 44:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1, 4, 7 },
              { 2, 5, 8 },
              { 3, 6, 9 } };
    // обчислюємо кумулятивну суму елементів в кожному
    // стовпці матриці A та виводимо суму в консоль
    std::cout << "Cumulative sum of elements in each
    column of matrix A:" << std::endl;
    std::cout << Matrix::cumsum(A, 1) << std::endl;
    // обчислюємо кумулятивну суму елементів в кожному
    // рядку матриці A та виводимо суму в консоль
    std::cout << "Cumulative sum of elements in each
    row of matrix A:" << std::endl;
    std::cout << Matrix::cumsum(A, 2) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Cumulative sum of elements in each column of matrix A:
1  4   7
3  9  15
6 15  24

Cumulative sum of elements in each row of matrix A:
1  5  12
2  7  15
3  9  18
```

- `static Matrix cumprod(const Matrix &m, unsigned int dim = 1)` – метод обчислює кумулятивний добуток елементів матриці `m` за вказаною розмірністю `dim` (`dim = 1` – добуток по рядках, `dim = 2` – добуток по стовпцях);

Приклад 45:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1, 4, 7 },
              { 2, 5, 8 },
              { 3, 6, 9 } };
    // обчислюємо кумулятивний добуток
    // елементів в кожному
    // стовпці матриці A
    // та виводимо добуток в консоль
    std::cout << "Cumulative product
    of elements in each
    column of matrix A" << std::endl;
    std::cout << Matrix::cumprod(A, 1) << std::endl;
    // обчислюємо кумулятивний добуток
    // елементів в кожному
    // рядку матриці A
    // та виводимо добуток в консоль
    std::cout << "Cumulative product
    of elements in each
    row of matrix A" << std::endl;
    std::cout << Matrix::cumprod(A, 2) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Cumulative product of elements in each column of matrix A
1   4   7
2  20  56
6 120 504

Cumulative product of elements in each row of matrix A
1   4  28
2  10  80
3  18 162
```

- `static Matrix randn(double mean, double stddev, size_t rows, size_t columns)` – метод повертає матрицю із заданою кількістю рядків `rows` та стовпців `columns`, елементи якої заповнюються псевдовипадковими числами із нормального розподілу; `mean` – середнє значення, `stddev` – стандартне відхилення;

Приклад 46:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 4, елементи якої
    // заповнюються псевдовипадковими числами
    // із нормального розподілу
    Matrix A = Matrix::randn(5, 2, 3, 4);
    // виводимо значення елементів матриці A в консоль
    std::cout << "Matrix A:\n" << A << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix A:
4.77817 1.97515 6.97724 1.80859
3.53278 4.02004 5.51554 5.34789
7.23201 6.83308 4.81389 4.29364
```

- `static Matrix rand(double minv, double maxv, size_t rows, size_t columns)` – метод повертає матрицю із заданою кількістю рядків `rows` та стовпців `columns`, елементи якої заповнюються псевдовипадковими числами з рівномірного розподілу в діапазоні від `minv` до `maxv`;

Приклад 47:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 5 x 7,
    // елементи якої заповнюються псевдовипадковими
    // числами з рівномірного розподілу
    // в діапазоні від -4 до 8
    Matrix A = Matrix::rand(-4, 8, 5, 7);
    // виводимо значення елементів матриці A в консоль
    std::cout << "Matrix A:\n" << A << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix A:
 2.4587  7.3348  2.1180  0.1648  1.0590 -1.2094  4.7097
-1.6789  6.0206 -2.5567  6.6124  6.5194  1.8203 -1.1307
 4.0875 -3.4391 -2.1145  5.6032  0.8794 -2.5754  3.9388
 0.9535  2.4908  5.8583  3.2997  2.4328  1.8943  0.1769
 2.4505  1.3068  1.4079 -1.3114 -2.7056  5.8405  2.4142
```

- `static Matrix randi(long long minv, long long maxv, size_t rows, size_t columns)` – метод повертає матрицю із заданою кількістю рядків `rows` та стовпців `columns`, яка складається з цілих чисел у форматі з плаваючою комою, елементи якої заповнюються

псевдовипадковими числами з рівномірного розподілу в діапазоні від `minv` до `maxv`;

Приклад 48:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 4 x 5
    // елементи якої заповнюються псевдовипадковими
    // числами з рівномірного розподілу
    // в діапазоні від 1 до 9
    Matrix A = Matrix::randi(1, 9, 4, 5);
    // виводимо значення елементів матриці A в консоль
    std::cout << "Matrix A:\n" << A << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix A:
2 1 2 1 8
7 8 5 1 3
3 2 3 5 6
1 1 3 5 4
```

- `static std::pair<Matrix, Matrix> lu(const Matrix &m)` – метод здійснює LU-розклад квадратної матриці `m` у вигляді M = LU; метод повертає матриці [L, U] у вигляді пари, де L – нижня трикутна матриця, U – верхня трикутна матриця;

Приклад 49:

```
#include <iostream>
#include "Matrix.h"
```

```
int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 10, -7, 0 },
              { -3,  2, 6 },
              {  5, -1, 5 } };
    // здійснюємо LU-розклад матриці A
    std::pair<Matrix, Matrix> LU = Matrix::lu(A);
    // виводимо значення елементів
    // нижньої трикутної матриці в консоль
    std::cout << "Matrix L:\n" << LU.first << std::endl;
    // виводимо значення елементів
    // верхньої трикутної матриці в консоль
    std::cout << "Matrix U:\n" << LU.second << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix L:
 1.0   0.0   0.0
-0.3   1.0   0.0
 0.5  -25.0  1.0

Matrix U:
10.0  -7.0   0.0
 0.0  -0.1   6.0
 0.0   0.0 155.0
```

- `static Matrix chol(const Matrix &m)` – метод здійснює розклад Холецького квадратної симетричної позитивно визначеної матриці `m` у вигляді $M = LL^T$, де L – нижня трикутна матриця із строго позитивними елементами на діагоналі;

Приклад 50:

```
#include <iostream>
#include "Matrix.h"

int main()
```

```
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1, 0, 1 },
              { 0, 2, 0 },
              { 1, 0, 3 } };
    // здійснюємо розклад Холецького матриці A
    Matrix L = Matrix::chol(A);
    // виводимо значення елементів матриці L в консоль
    std::cout << "Matrix L:\n" << L << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix L:
1.0000 0.0000 0.0000
0.0000 1.4142 0.0000
1.0000 0.0000 1.4142
```

- `static std::pair<Matrix, Matrix> qr(const Matrix &m)` – метод здійснює QR-розклад матриці `m` розміром $s_1 \times s_2$ у вигляді M = QR; метод повертає матриці [Q, R] у вигляді пари, де Q – ортогональна матриця, R – верхньотрикутна матриця;

Приклад 51:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 4
    Matrix A{ { 12, -51,   4, -13 },
              {  6, 167, -68,  18 },
              { -4,  24, -41,   8 } };
    // здійснюємо QR-розклад матриці A
    std::pair<Matrix, Matrix> QR = Matrix::qr(A);
    // виводимо значення елементів
    // ортогональної та верхньотрикутної
    // матриць в консоль
```

```
    std::cout << "Matrix Q:\n" << QR.first << std::endl;
    std::cout << "Matrix R:\n" << QR.second << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix Q:
 0.8571 -0.3943 -0.3314
 0.4286  0.9029  0.0343
-0.2857  0.1714 -0.9429

Matrix R:
14.0000  21.0000 -14.0000 -5.7143
-0.0000 175.0000 -70.0000 22.7486
 0.0000   0.0000  35.0000 -2.6171
```

- `static Matrix mean(const Matrix &m, unsigned int dim = 1)` – метод повертає середнє значення за вказаною розмірністю `dim` (`dim = 1` – повертає вектор-рядок, що містить середнє значення кожного стовпця, `dim = 2` – повертає вектор-стовпець, що містить середнє значення кожного рядка);

Приклад 52:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 4
    Matrix A{ { 12, -51,   4, -13 },
              {  6, 167, -68,  18 },
              { -4,  24, -41,   8 } };
    // виводимо значення елементів
    // вектора-стовпця, що містить
    // середнє значення кожного рядка матриці A
    std::cout << "Mean of each row of matrix A:\n";
    std::cout << Matrix::mean(A, 2) << std::endl;
```

```
    return 0;
}
```

Результат роботи програми:

```
Mean of each row of matrix A:
-12.00
 30.75
 -3.25
```

- `static size_t numel(const Matrix &m)` – метод повертає кількість елементів матриці `m`;
- `static double dotv(const Matrix &v1, const Matrix &v2)` – метод обчислює скалярний добуток двох векторів `v1` та `v2`;

Приклад 53:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо вектор v1
    Matrix v1{ { 5, 2, 7 } };
    // створюємо вектор v2
    Matrix v2{ { 3, 4, 8 } };
    // обчислюємо скалярний добуток векторів
    // v1 і v2, та виводимо результат в консоль
    std::cout << "Dot product of v1 and v2: ";
    std::cout << Matrix::dotv(v1, v2) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Dot product of v1 and v2: 79
```

- `static Matrix dot(const Matrix &m1, const Matrix &m2, unsigned int dim = 1)` – метод обчислює скалярний добуток за

вказаною розмірністю `dim` (`dim = 1` – обробляє стовпці матриць `m1` і `m2` як вектори, та повертає скалярний добуток відповідних стовпців; `dim = 2` – обробляє рядки матриць `m1` і `m2` як вектори, та повертає скалярний добуток відповідних рядків);

Приклад 54:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 7, 6, 1 },
              { 3, 4, 8 },
              { 1, 0, 7 } };
    // створюємо матрицю B розміром 3 x 3
    Matrix B{ { 6, 8, 1 },
              { 3, 2, 0 },
              { 8, 4, 1 } };
    // обчислюємо скалярний добуток відповідних
    // стовпців матриць A та B та виводимо
    // результат в консоль
    std::cout << "Dot product of columns
    matrix A and B:\n";
    std::cout << Matrix::dot(A, B, 1) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Dot product of columns matrix A and B:
59 56 8
```

- `static Matrix proj(const Matrix& a, const Matrix& b)` – метод обчислює проекцію вектора `a` на вектор `b`;
- `static double normv(const Matrix &v, unsigned int p = 2)` – метод обчислює норму вектора `v`; якщо `p = 1`, то норма є

сумою абсолютних значень елементів вектора; якщо p = 2 (за замовчуванням), то обчислюється евклідова норма вектора; якщо p = 3, то норма обчислюється як $\|v\|_{\infty} = \max_i |v_i|$; якщо p = 4, то норма обчислюється як $\|v\|_{-\infty} = \min_i |v_i|$;

Приклад 55:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо вектор v розміром
    Matrix v{ { 6, 2, 4 } };
    // обчислюємо евклідову норму вектора v
    // та виводимо значення норми в консоль
    std::cout << "Norm of v: ";
    std::cout << Matrix::normv(v) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Norm of v: 7.48331
```

- `static double norm(const Matrix &m, unsigned int p = 3)` – метод обчислює норму матриці m; якщо p = 1, то норма є максимальною абсолютною сумою стовпців $\|X\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|$, $m,n \ge 2$; якщо p = 2, то норма є максимальною абсолютною сумою рядків $\|X\|_{\infty} = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}|$, $m,n \ge 2$; якщо p = 3 (за замовчуванням), то обчислюється норма Фробеніуса $\|X\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}$, $m,n \ge 2$;

Приклад 56:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 7, 6, 1 },
              { 3, 4, 8 },
              { 1, 0, 7 } };
    // обчислюємо норму Фробеніуса матриці A
    // та виводимо значення норми в консоль
    std::cout << "Frobenius norm of matrix A: ";
    std::cout << Matrix::norm(A) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Frobenius norm of matrix A: 15
```

- `static Matrix selectv(const Matrix &m, size_t index, unsigned int dim = 1)` – метод повертає вектор, що складається з елементів матриці `m`, які розташовані в рядку або стовпці матриці з індексом `index` за вказаною розмірністю `dim` (`dim = 1` – повертає вектор-рядок з індексом `index`, `dim = 2` – повертає вектор-стовпець з індексом `index`);

Приклад 57:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 7, 6, 1 },
              { 3, 4, 8 },
```

```
                { 1, 0, 7 } };
    // формуємо вектор-рядок, що складається з
    // елементів матриці A, які розташовані в
    // рядку з індексом 1
    Matrix v = Matrix::selectv(A, 1, 1);
    // виводимо значення елементів вектора v в консоль
    std::cout << "v: " << v << std::endl;
    return 0;
}
```

Результат роботи програми:

```
v: 3 4 8
```

- `static void insertv(Matrix &m, const Matrix &v, size_t index)` – метод вставляє вектор `v` в рядок або стовпець матриці `m` з індексом `index`, замінюючи відповідні елементи;

Приклад 58:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 7, 6, 1 },
              { 3, 4, 8 },
              { 1, 0, 7 } };
    // створюємо вектор-рядок v
    Matrix v{ { 9, 5, 1 } };
    // вставляємо вектор v в рядок матриці A з індексом
    // 2, замінюючи відповідні елементи рядка матриці A
    Matrix::insertv(A, v, 2);
    // виводимо значення елементів матриці A в консоль
    std::cout << "Matrix A:\n" << A << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix A:
7 6 1
3 4 8
9 5 1
```

- `static Matrix cummin(const Matrix &m, unsigned int dim = 1)` – метод повертає кумулятивні мінімуми матриці `m` за вказаною розмірністю `dim` (`dim = 1` – повертає кумулятивні мінімуми у кожному стовпці матриці `m`, `dim = 2` – повертає кумулятивні мінімуми у кожному рядку матриці `m`);

Приклад 59:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 3, 5, 2 },
              { 1, 6, 3 },
              { 7, 8, 1 } };
    // обчислюємо кумулятивні мінімуми у кожному
    // стовпці матриці A та виводимо їх в консоль
    std::cout << "Cumulative minimums in each column
    of matrix A:\n";
    std::cout << Matrix::cummin(A, 1) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Cumulative minimums in each column of matrix A:
3 5 2
1 5 2
1 5 1
```

- `static Matrix cummax(const Matrix &m, unsigned int dim = 1)` – метод повертає кумулятивні максимуми матриці `m` за вказаною розмірністю `dim` (`dim = 1` – повертає кумулятивні максимуми у кожному стовпці матриці `m`, `dim = 2` – повертає кумулятивні максимуми у кожному рядку матриці `m`);

Приклад 60:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 3, 5, 2 },
              { 1, 6, 3 },
              { 7, 8, 1 } };
    // обчислюємо кумулятивні максимуми у кожному
    // рядку матриці A та виводимо їх в консоль
    std::cout << "Cumulative maximums in each row
    of matrix A:\n";
    std::cout << Matrix::cummax(A, 2) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Cumulative maximums in each row of matrix A:
3 5 5
1 6 6
7 8 8
```

- `static Matrix addrow(const Matrix &m, size_t index)` – метод формує нову матрицю із елементів початкової матриці `m`, додаючи рядок в позиції `index`;

Приклад 61:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 3, 5, 2 },
              { 1, 6, 3 },
              { 7, 8, 1 } };
    // створюємо матрицю B розміром 4 x 3,
    // яка складається з елементів початкової матриці A
    // з додатковим рядком в позиції 1
    Matrix B = Matrix::addrow(A, 1);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
3 5 2
0 0 0
1 6 3
7 8 1
```

- `static Matrix addcol(const Matrix &m, size_t index)` – метод формує нову матрицю із елементів початкової матриці `m`, додаючи стовпець в позиції `index`;

Приклад 62:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
```

```
    Matrix A{ { 3, 5, 2 },
              { 1, 6, 3 },
              { 7, 8, 1 } };
    // створюємо матрицю B розміром 3 x 4,
    // яка складається з елементів початкової матриці A
    // з додатковим стовпцем в позиції 2
    Matrix B = Matrix::addcol(A, 2);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
3 5 0 2
1 6 0 3
7 8 0 1
```

- `static Matrix removerow(const Matrix &m, size_t index)` – метод формує нову матрицю із елементів початкової матриці `m`, видаляючи рядок в позиції `index`;

Приклад 63:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 3, 5, 2 },
              { 1, 6, 3 },
              { 7, 8, 1 } };
    // створюємо матрицю B розміром 2 x 3,
    // яка складається з елементів початкової
    // матриці A, але без рядка в позиції 1
    Matrix B = Matrix::removerow(A, 1);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
```

```
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
3 5 2
7 8 1
```

- `static Matrix removecol(const Matrix &m, size_t index)` – метод формує нову матрицю із елементів початкової матриці `m`, видаляючи стовпець в позиції `index`;

Приклад 64:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 3, 5, 2 },
              { 1, 6, 3 },
              { 7, 8, 1 } };
    // створюємо матрицю B розміром 3 x 2,
    // яка складається з елементів початкової
    // матриці A, але без стовпця в позиції 2
    Matrix B = Matrix::removecol(A, 2);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
3 5
1 6
7 8
```

- `static Matrix rayleigh(const Matrix &m, double eigenValue)` – метод ітерацій Релея; метод ітеративно обчислює власний вектор за деяким початковим наближенням до власного значення `eigenValue` матриці `m`;

Приклад 65:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 3, 5, 2 },
              { 1, 6, 3 },
              { 7, 8, 1 } };
    // обчислюємо власний вектор матриці A за деяким
    // початковим наближенням до власного значення
    Matrix v = Matrix::rayleigh(A, 2);
    // виводимо значення елементів вектора v в консоль
    std::cout << "v:\n" << v << std::endl;
    return 0;
}
```

Результат роботи програми:

```
v:
-0.561592
 0.499046
-0.659975
```

- `static std::pair<Matrix, Matrix> eig(const Matrix &m)` – метод обчислює власні значення та власні вектори квадратної матриці `m`; метод повертає матриці [V, D] у вигляді пари, де V – квадратна матриця, стовпці якої є відповідними правими власними векторами матриці `m`, D – діагональна матриця із власними значеннями матриці `m` на головній діагоналі;

Приклад 66:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 3, 5, 2 }, { 1, 6, 3 }, { 7, 8, 1 } };
    // обчислюємо власні значення та
    // власні вектори матриці A
    std::pair<Matrix, Matrix> VD = Matrix::eig(A);
    // присвоюємо значення матриці з
    // власними векторами в змінну V
    Matrix V = VD.first;
    // присвоюємо значення матриці з
    // власними значеннями в змінну D
    Matrix D = VD.second;
    // виводимо значення елементів матриці V в консоль
    std::cout << "V:\n" << V << std::endl;
    // виводимо значення елементів матриці D в консоль
    std::cout << "D:\n" << D << std::endl;
    // обчислюємо норму вектора
    // mult((A - lambda * I), B),
    // де lambda - власне значення матриці A,
    // B - власний вектор, що відповідає
    // власному значенню lambda,
    // I - одинична матриця
    for (int i = 0; i < 3; ++i)
    {
        std::cout << "[V" << i + 1 << ',' << 'D' <<
        i + 1 << "]: " << Matrix::normv(Matrix::mult(A -
        D[i][i] * Matrix::eye(3), Matrix::selectv(V, i,
        2))) << std::endl;
    }
    return 0;
}
```

Результат роботи програми:

```
V:
-0.47914  0.03325 -0.56159
-0.50260  0.34144  0.49905
-0.71959 -0.93932 -0.65998

D:
11.24854  0.00000 0.00000
 0.00000 -2.15578 0.00000
 0.00000  0.00000 0.90724

[V1,D1]: 0.00000
[V2,D2]: 0.00000
[V3,D3]: 0.00000
```

- `static Matrix pinv(const Matrix& m)` – метод обчислює псевдообернену матрицю Мура-Пенроуза для заданої матриці `m`;

Приклад 67:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 5
    Matrix A{ { 7, 3, 9, 1, 6 },
              { 2, 5, 0, 4, 3 },
              { 8, 7, 6, 5, 4 } };
    // обчислюємо псевдообернену матрицю
    Matrix B = Matrix::pinv(A);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Pseudo-inverse matrix B:\n";
    std::cout << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Pseudo-inverse matrix B:
-0.06285 -0.15118  0.16107
-0.03909  0.06783  0.04013
 0.05468 -0.08950  0.02463
-0.06039  0.03803  0.06056
 0.18758  0.27037 -0.25502
```

- `static void swap(Matrix& m, size_t index1, size_t index2, unsigned int dim = 1)` – метод змінює місцями вміст двох рядків або двох стовпців матриці `m` з індексами `index1` та `index2`; якщо `dim = 1`, відбувається обмін рядками, якщо `dim = 2` – обмін стовпцями;

Приклад 68:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 5
    Matrix A{ { 7, 3, 9, 1, 6 },
              { 2, 5, 0, 4, 3 },
              { 8, 7, 6, 5, 4 } };
    // змінюємо місцями вміст 1-го і 3-го
    // стовпців матриці A
    Matrix::swap(A, 1, 3, 2);
    // виводимо значення елементів матриці A в консоль
    std::cout << "Matrix A after swapping columns with
    indexes 1 and 3:\n" << A << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix A after swapping columns with indexes 1 and 3:
7 1 9 3 6
2 4 0 5 3
8 5 6 7 4
```

- `static Matrix kron(const Matrix& m1, const Matrix& m2)` – метод обчислює тензорний добуток матриць `m1` та `m2`; якщо матриця `m1` має розмір $s_1 \times s_2$, а `m2` – розмір $s_3 \times s_4$, то результуюча матриця матиме розмір $(s_1 * s_3) \times (s_2 * s_4)$;

Приклад 69:

```cpp
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 2 x 3
    Matrix A{ { 1, 2, 3 },
              { 4, 5, 6 } };
    // створюємо матрицю B розміром 4 x 2
    Matrix B{ { 1, 2 },
              { 3, 4 },
              { 5, 6 },
              { 7, 8 } };
    // обчислюємо тензорний добуток матриць A та B
    Matrix C = Matrix::kron(A, B);
    // виводимо значення елементів матриці C в консоль
    std::cout << "Matrix C:\n" << C << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix C:
 1  2  2  4  3  6
 3  4  6  8  9 12
 5  6 10 12 15 18
 7  8 14 16 21 24
 4  8  5 10  6 12
12 16 15 20 18 24
20 24 25 30 30 36
28 32 35 40 42 48
```

- `static double trace(const Matrix& m)` – метод обчислює слід матриці `m`;

Приклад 70:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю розміром 3 x 3
    Matrix A{ { 5, 3, 8 },
              { 2, 1, 4 },
              { 7, 6, 9 } };
    // обчислюємо слід матриці A
    double tr = Matrix::trace(A);
    // виводимо результат в консоль
    std::cout << "Trace of matrix A: " << tr;
    return 0;
}
```

Результат роботи програми:

```
Trace of matrix A: 15
```

- `static std::tuple<Matrix, Matrix, Matrix> svd(const Matrix& m)` – метод виконує сингулярний розклад (SVD – Singular

Value Decomposition) матриці m; розклад представляється у вигляді добутку трьох матриць: $m = USV^T$, де U – ортогональна матриця, стовпці якої є лівими сингулярними векторами матриці m, S – діагональна матриця, що містить сингулярні значення матриці m, упорядковані за спаданням вздовж головної діагоналі, V – ортогональна матриця, стовпці якої є правими сингулярними векторами матриці m;

Приклад 71:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю розміром 3 x 3
    Matrix A{ {  1,  0,  1 },
              { -1, -2,  0 },
              {  0,  1, -1 } };
    // виконуємо SVD-розклад матриці A
    // на три матриці U, S, V, де
    // U – ортогональна матриця, що містить ліві
    // сингулярні вектори,
    // S – діагональна матриця, що містить
    // сингулярні значення,
    // V – ортогональна матриця, що містить
    // праві сингулярні вектори
    auto [U, S, V] = Matrix::svd(A);
    // виводимо значення елементів матриці U в консоль
    std::cout << "Matrix U:\n" << U << std::endl;
    // виводимо значення елементів матриці S в консоль
    std::cout << "Matrix S:\n" << S << std::endl;
    // виводимо значення елементів матриці V в консоль
    std::cout << "Matrix V:\n" << V << std::endl;
    // перевіряємо правильність розкладу: A = U * S * V^T
    std::cout << "Matrix A = U * S * V^T:\n" <<
    Matrix::mult(U, Matrix::mult(S,
    Matrix::transpose(V))) << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix U:
 0.120000  0.809712 0.574427
-0.901753 -0.153123 0.404222
 0.415261 -0.566498 0.711785

Matrix S:
2.46050 0.00000 0.00000
0.00000 1.69963 0.00000
0.00000 0.00000 0.23912

Matrix V:
 0.415261  0.566498  0.711785
 0.901753 -0.153123 -0.404222
-0.120000  0.809712 -0.574427

Matrix A = U * S * V^T:
 1  0  1
-1 -2  0
 0  1 -1
```

- `static Matrix pascal(size_t n)` – метод створює матрицю Паскаля розміром $n \times n$;

Приклад 72:

```cpp
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю Паскаля розміром 7 x 7
    Matrix P = Matrix::pascal(7);
    // виводимо значення елементів матриці P в консоль
    std::cout << "Pascal matrix:\n" << P << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Pascal matrix:
1   1   1   1   1   1   1
1   2   3   4   5   6   7
1   3   6  10  15  21  28
1   4  10  20  35  56  84
1   5  15  35  70 126 210
1   6  21  56 126 252 462
1   7  28  84 210 462 924
```

- `static Matrix interp1(const Matrix& x, const Matrix& y, const Matrix& xi)` – метод виконує одновимірну лінійну інтерполяцію, дозволяючи знаходити значення функції у проміжних точках на основі наявного набору дискретних значень; `x` – вектор-рядок або вектор-стовпець, що містить значення аргументів функції (вузли інтерполяції), `y` – вектор-рядок або вектор-стовпець, що містить значення функції у відповідних точках `x`, `xi` – вектор-рядок або вектор-стовпець, що містить значення аргументів, у яких потрібно обчислити значення функції;

Приклад 73:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо вектор-рядок,
    // що містить значення аргументів функції
    Matrix x{ { 0, 0.5, 1, 2, 3.5, 4, 6 } };
    // створюємо вектор-стовпець,
    // що містить значення функції в точках x
    Matrix y{ { 82.914, 102.491, 172.301, 281.908,
    202.791, 159.072, 187.951 } };
    // створюємо вектор-рядок,
    // що містить значення аргументів,
```

```
    // в яких потрібно знайти значення функції
    Matrix xi{ { 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5,
    5, 5.5, 6 } };
    // знаходимо значення функції в точках xi
    Matrix yi = Matrix::interp1(x, y, xi);
    // виводимо значення елементів вектора yi в консоль
    std::cout << "Interpolated values:\n" <<
    Matrix::transpose(yi);
    return 0;
}
```

Результат роботи програми:

```
Interpolated values:
82.914
102.491
172.301
227.105
281.908
255.536
229.163
202.791
159.072
166.292
173.512
180.731
187.951
```

- `static Matrix interp2(const Matrix& x, const Matrix& y, const Matrix& v, const Matrix& xi, const Matrix& yi)` – метод виконує білінійну інтерполяцію, дозволяючи знаходити значення функції у проміжних точках на основі наявного набору дискретних значень; `x` та `y` – матриці однакового розміру, що містять координати вузлів, у яких визначені значення функції; `v` – матриця такого ж розміру, що містить значення функції у відповідних точках `x` та `y`; `xi` та `yi` – матриці однакового розміру, що містять координати точок, у яких потрібно обчислити значення функції; метод повертає матрицю того ж

розміру, що й `xi` та `yi`, яка містить інтерпольовані значення функції у вказаних точках;

Приклад 74:

```
#include <iostream>
#include <iomanip>
#include "Matrix.h"

int main()
{
    // формуємо вектор-рядок x-координат, що складається
    // з послідовності чисел від -1 до 1 з кроком 0.5
    Matrix coordx = Matrix::range(-1, 1, 0.5);
    // формуємо вектор-рядок y-координат, що складається
    // з послідовності чисел від -2 до 2 з кроком 0.5
    Matrix coordy = Matrix::range(-2, 2, 0.5);
    // формуємо координати двовимірної сітки,
    // в яких визначені значення функції
    auto [x, y] = Matrix::meshgrid(coordx, coordy);
    // обчислюємо значення Гаусової функції
    // в точках з координатами (x, y)
    Matrix r = Matrix::hypot(x, y);
    Matrix v = Matrix::exp(-Matrix::pow(r, 2));
    // виводимо значення Гаусової функції в точках
    // з координатами (x, y) в консоль
    std::cout << std::fixed << std::setprecision(4) <<
    "v:\n" << v << std::endl;
    // формуємо вектор-рядок x-координат, що складається
    // з послідовності чисел від -1 до 1 з кроком 0.25
    coordx = Matrix::range(-1, 1, 0.25);
    // формуємо вектор-рядок y-координат, що складається
    // з послідовності чисел від -2 до 2 з кроком 0.25
    coordy = Matrix::range(-2, 2, 0.25);
    // формуємо координати двовимірної сітки,
    // в яких необхідно знайти значення функції
    auto [xi, yi] = Matrix::meshgrid(coordx, coordy);
    // виконуємо білінійну інтерполяцію
    // для знаходження значень функції
    // в точках з координатами (xi, yi)
    // на основі відомих значень функції
```

```
    // в точках з координатами (x, y)
    Matrix vi = Matrix::interp2(x, y, v, xi, yi);
    // виводимо значення Гаусової функції в точках
    // з координатами (xi, yi) в консоль
    std::cout << "vi:\n" << vi << std::endl;
    return 0;
}
```

Результат роботи програми:

```
v:
0.0067 0.0143 0.0183 0.0143 0.0067
0.0388 0.0821 0.1054 0.0821 0.0388
0.1353 0.2865 0.3679 0.2865 0.1353
0.2865 0.6065 0.7788 0.6065 0.2865
0.3679 0.7788 1.0000 0.7788 0.3679
0.2865 0.6065 0.7788 0.6065 0.2865
0.1353 0.2865 0.3679 0.2865 0.1353
0.0388 0.0821 0.1054 0.0821 0.0388
0.0067 0.0143 0.0183 0.0143 0.0067

vi:
0.0067 0.0105 0.0143 0.0163 0.0183 0.0163 0.0143 0.0105 0.0067
0.0228 0.0355 0.0482 0.0550 0.0619 0.0550 0.0482 0.0355 0.0228
0.0388 0.0604 0.0821 0.0937 0.1054 0.0937 0.0821 0.0604 0.0388
0.0871 0.1357 0.1843 0.2105 0.2366 0.2105 0.1843 0.1357 0.0871
0.1353 0.2109 0.2865 0.3272 0.3679 0.3272 0.2865 0.2109 0.1353
0.2109 0.3287 0.4465 0.5099 0.5733 0.5099 0.4465 0.3287 0.2109
0.2865 0.4465 0.6065 0.6927 0.7788 0.6927 0.6065 0.4465 0.2865
0.3272 0.5099 0.6927 0.7910 0.8894 0.7910 0.6927 0.5099 0.3272
0.3679 0.5733 0.7788 0.8894 1.0000 0.8894 0.7788 0.5733 0.3679
0.3272 0.5099 0.6927 0.7910 0.8894 0.7910 0.6927 0.5099 0.3272
0.2865 0.4465 0.6065 0.6927 0.7788 0.6927 0.6065 0.4465 0.2865
0.2109 0.3287 0.4465 0.5099 0.5733 0.5099 0.4465 0.3287 0.2109
0.1353 0.2109 0.2865 0.3272 0.3679 0.3272 0.2865 0.2109 0.1353
0.0871 0.1357 0.1843 0.2105 0.2366 0.2105 0.1843 0.1357 0.0871
0.0388 0.0604 0.0821 0.0937 0.1054 0.0937 0.0821 0.0604 0.0388
0.0228 0.0355 0.0482 0.0550 0.0619 0.0550 0.0482 0.0355 0.0228
0.0067 0.0105 0.0143 0.0163 0.0183 0.0163 0.0143 0.0105 0.0067
```

- `static Matrix gauss(const Matrix& A, const Matrix& B)` – метод розв’язує систему лінійних алгебраїчних рівнянь $Ax = B$ відносно x із використанням методу Гаусса; `A` – квадратна матриця коефіцієнтів системи розміром $n \times n$; `B` – вектор-стовпець вільних членів системи

розміром $n \times 1$; метод повертає вектор-стовпець розміром $n \times 1$, що є розв'язком системи;

Приклад 75:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1,  2, -1 },
              { 2, -3,  2 },
              { 3,  1,  1 } };
    // створюємо вектор-стовпець B розміром 3 x 1
    Matrix b = { { 2 },
                 { 2 },
                 { 8 } };
    // розв'язуємо систему лінійних
    // алгебраїчних рівнянь метод Гауса
    Matrix x = Matrix::gauss(A, b);
    // виводимо значення елементів вектора x в консоль
    std::cout << "Solution of the system of linear
    algebraic equations:" << std::endl;
    std::cout << x << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Solution of the system of linear algebraic equations:
1
2
3
```

- `static size_t rank(const Matrix& m)` – метод обчислює ранг матриці `m`;

Приклад 76:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 4 x 5
    Matrix A{ {  3, 1, -1, -2,  8 },
              {  7, 1, -2, -1, 12 },
              { 11, 1, -3,  0, 16 },
              {  2, 2, -1, -5, 12 } };
    // обчислюємо ранг матриці A
    double r = Matrix::rank(A);
    // виводимо ранг матриці A в консоль
    std::cout << "Rank of matrix A: " << r << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Rank of matrix A: 2
```

- `static double cond(const Matrix& m)` – метод обчислює число обумовленості матриці `m` за нормою Фробеніуса;

Приклад 77:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1,  2, -1 },
              { 2, -3,  2 },
              { 3,  1,  1 } };
    // обчислюємо число обумовленості матриці A
    double cond_A = Matrix::cond(A);
    // виводимо число обумовленості матриці A в консоль
    std::cout << "cond(A) = " << cond_A << std::endl;
```

```
    return 0;
}
```

Результат роботи програми:

```
cond(A) = 12.1527
```

- `static Matrix cross(const Matrix& m1, const Matrix& m2, unsigned int dim = 1)` – метод обчислює векторний добуток матриць `m1` і `m2` уздовж вибраного виміру `dim`; якщо `dim = 1`, метод розглядає стовпці `m1` і `m2` як тривимірні вектори та повертає матрицю, що містить векторний добуток відповідних стовпців; якщо `dim = 2`, метод розглядає рядки `m1` і `m2` як тривимірні вектори та повертає матрицю, що містить векторний добуток відповідних рядків; `m1` і `m2` повинні мати однакові розміри, причому `size(m1, dim)` та `size(m2, dim)` мають дорівнювати 3, що відповідає визначенню векторного добутку у тривимірному просторі;

Приклад 78:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 5
    Matrix A = { { 13, 14,  5, 15, 15 },
                 { 14, 10,  9,  3,  8 },
                 {  2,  2, 15, 15, 13 } };
    // створюємо матрицю B розміром 3 x 5
    Matrix B = { {  4, 20,  1, 17, 10 },
                 { 11, 24, 22, 19, 17 },
                 { 23, 17, 24, 19,  5 } };
    // обчислюємо векторний добуток відповідних
    // рядків матриці A та матриці B
    Matrix C = Matrix::cross(A, B, 2);
```

```
    // виводимо значення елементів матриці C в консоль
    std::cout << "Matrix C:\n" << C << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix C:
 300  122 -114 -228 -181
-291 -198 -105  -30   55
  87  136  101  234  175
```

- `static Matrix funm(const Matrix& m, const std::function<double(double)>& func)` – метод обчислює значення функції `func` від кожного елемента матриці `m`;

Приклад 79:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1, 2, 3 },
              { 4, 5, 6 },
              { 7, 8, 9 } };
    // створюємо функцію f(x) = x^2
    auto f = [](double x) { return x * x; };
    // обчислюємо значення функції f
    // від кожного елемента матриці A
    Matrix B = Matrix::funm(A, f);
    // виводимо значення елементів матриці B в консоль
    std::cout << "Matrix B:\n" << B << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix B:
 1  4  9
16 25 36
49 64 81
```

- `static Matrix vec2mat(const std::vector<std::vector<double>>& vec)` – метод конвертує вектор векторів `vec` у матрицю; якщо `vec` містить n підвекторів однакової довжини m, метод створює матрицю розміром $n \times m$, зберігаючи структуру вихідних даних; якщо `vec` порожній або підвектори мають різну довжину, то генерується виняток;

Приклад 80:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо вектор векторів розміром 3 x 3
    std::vector<std::vector<double>> vec =
    {
      { 1, 2, 3 },
      { 4, 5, 6 },
      { 7, 8, 9 }
    };
    // конвертуємо вектор векторів у матрицю
    Matrix A = Matrix::vec2mat(vec);
    // виводимо значення елементів матриці A в консоль
    std::cout << "Matrix A:\n" << A << std::endl;
    return 0;
}
```

Результат роботи програми:

```
Matrix A:
1 2 3
4 5 6
7 8 9
```

- `static std::vector<std::vector<double>> mat2vec(const Matrix& m)` – метод конвертує матрицю `m` у вектор векторів; метод повертає вектор векторів, який містить *n* підвекторів, де *n* – кількість рядків у матриці `m`, а довжина кожного підвектора дорівнює кількості стовпців матриці `m`;

Приклад 81:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // створюємо матрицю A розміром 3 x 3
    Matrix A{ { 1, 2, 3 },
              { 4, 5, 6 },
              { 7, 8, 9 } };
    // конвертуємо матрицю в вектор векторів
    std::vector<std::vector<double>> vec =
    Matrix::mat2vec(A);
    // виводимо значення елементів вектора в консоль
    std::cout << "Result:" << std::endl;
    for (auto v : vec)
    {
        for (auto x : v)
        {
            std::cout << x << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Результат роботи програми:

```
Result:
1 2 3
4 5 6
7 8 9
```

- `static double trapz(const Matrix& x, double dx, const std::function<double(double)>& func = nullptr)` – метод виконує числове інтегрування функції однієї змінної методом трапецій; `x` повинен бути вектором, який містить дискретний набір точок інтегрування; `dx` – додатне число, що визначає крок інтегрування; якщо параметр `func` задано, метод спочатку застосовує функцію `func` до кожного елемента `x`, визначаючи значення функції в цих точках, а потім обчислює наближене значення інтегралу; якщо параметр `func` не задано, то `x` повинен містити значення функції в точках інтегрування;

Приклад 82:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // задаємо крок інтегрування
    double dx = 0.01;
    // створюємо дискретний набір точок інтегрування
    // від -4 до 4 з кроком dx
    Matrix x = Matrix::range(-4, 4, dx);
    // створюємо функцію f(x) = cos(x) * sin(x) *
    // (x^2 + x + 1) для інтегрування
    auto f = [](double x) { return std::cos(x) *
    std::sin(x) * (x * x + x + 1); };
    // обчислюємо інтеграл від функції f(x)
    // методом трапецій
    double res = Matrix::trapz(x, dx, f);
    // виводимо наближене значення інтегралу в консоль
```

```
    std::cout << "Result:" << std::endl;
    std::cout << std::setprecision(12) << res;
    return 0;
}
```

Результат роботи програми:

```
Result:
0.538338174022
```

- `static double simpson(const Matrix& x, double dx, const std::function<double(double)>& func = nullptr)` – метод виконує числове інтегрування функції однієї змінної методом Сімпсона; `x` повинен бути вектором, який містить дискретний набір точок інтегрування; `dx` – додатне число, що визначає крок інтегрування; якщо параметр `func` задано, метод спочатку застосовує функцію `func` до кожного елемента `x`, визначаючи значення функції в цих точках, а потім обчислює наближене значення інтегралу; якщо параметр `func` не задано, то `x` повинен містити значення функції в точках інтегрування;

Приклад 83:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // задаємо крок інтегрування
    double dx = 0.01;
    // створюємо дискретний набір точок інтегрування
    // від -4 до 4 з кроком dx
    Matrix x = Matrix::range(-4, 4, dx);
    // створюємо функцію f(x) = cos(x) * sin(x) *
    // (x^2 + x + 1) для інтегрування
    auto f = [](double x) { return std::cos(x) *
    std::sin(x) * (x * x + x + 1); };
    // обчислюємо інтеграл від функції f(x)
```

```
    // методом Сімпсона
    double res = Matrix::simpson(x, dx, f);
    // виводимо наближене значення інтегралу в консоль
    std::cout << "Result:" << std::endl;
    std::cout << std::setprecision(12) << res;
    return 0;
}
```

Результат роботи програми:

```
Result:
0.538339628871
```

- `static double boole(const Matrix& x, double dx, const std::function<double(double)>& func = nullptr)` – метод виконує числове інтегрування функції однієї змінної методом Буля; `x` повинен бути вектором, який містить дискретний набір точок інтегрування; `dx` – додатне число, що визначає крок інтегрування; якщо параметр `func` задано, метод спочатку застосовує функцію `func` до кожного елемента `x`, визначаючи значення функції в цих точках, а потім обчислює наближене значення інтегралу; якщо параметр `func` не задано, то `x` повинен містити значення функції в точках інтегрування.

Приклад 84:

```
#include <iostream>
#include "Matrix.h"

int main()
{
    // задаємо крок інтегрування
    double dx = 0.01;
    // створюємо дискретний набір точок інтегрування
    // від -4 до 4 з кроком dx
    Matrix x = Matrix::range(-4, 4, dx);
    // створюємо функцію f(x) = cos(x) * sin(x) *
    // (x^2 + x + 1) для інтегрування
```

```
    auto f = [](double x) { return std::cos(x) *
    std::sin(x) * (x * x + x + 1); };
    // обчислюємо інтеграл від функції f(x)
    // методом Буля
    double res = Matrix::boole(x, dx, f);
    // виводимо наближене значення інтегралу в консоль
    std::cout << "Result:" << std::endl;
    std::cout << std::setprecision(12) << res;
    return 0;
}
```

Результат роботи програми:

```
Result:
0.538339629272
```

У цьому розділі було детально розглянуто основи роботи з бібліотекою «MATRIX» на мові програмування C++. Викладено принципи конструювання та ініціалізації об’єктів класу `Matrix`, особливості використання перевантажених операторів та широкий спектр методів, що дозволяють виконувати чисельні обчислення.

Наведені приклади ілюструють можливості бібліотеки «MATRIX» у вирішенні різних обчислювальних задач, таких як розв’язання систем лінійних алгебраїчних рівнянь, чисельне диференціювання, інтерполяція, спектральний аналіз та статистична обробка даних. Робота з класом `Matrix` забезпечує ефективну реалізацію алгоритмів та автоматизацію складних обчислень, що є необхідним для аналізу та моделювання різних фізичних процесів.

Засвоєння матеріалу цього розділу дозволить студентам впевнено застосовувати бібліотеку «MATRIX» у практичній діяльності при розв’язанні прикладних задач в галузі обчислювальної фізики.

3. ПЕРЕЛІК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Deitel H., Deitel P. C++ How to Program : An Objects-Natural Approach. 11th ed. Pearson, 2024. 1364 p.
2. Deitel H., Deitel P. C++ How to Program. 10th ed. Pearson, 2017. 1075 p.
3. Deitel H., Deitel P. C++20 for Programmers. 3rd ed. Pearson, 2020. 1008 p.
4. Kirch-Prinz U., Prinz P. A Complete Guide to Programming in C++. 1st ed. Jones & Bartlett Learning, 2001. 848 p.
5. Lippman S. B., Lajoie J., Moo B. C++ Primer. 5th ed. Addison-Wesley Professional, 2012. 1342 p.
6. Meyers S. Effective C++: 55 Specific Ways to Improve Your Programs and Designs. 3rd ed. Addison-Wesley Professional, 2005. 320 p.
7. Meyers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. 1st ed. O'Reilly Media, Inc., 2014. 332 p.
8. Prata S. C++ Primer Plus. 6th ed. Addison-Wesley Professional, 2011. 1440 p.
9. Stroustrup B. A Tour of C++. 3rd ed. Addison-Wesley Professional, 2022. 320 p.
10. Stroustrup B. Programming: Principles and Practice Using C++. 3rd ed. Addison-Wesley Professional, 2024. 656 p.

4. ІНФОРМАЦІЙНІ РЕСУРСИ

1. Microsoft Learn [Електронний ресурс] : [Веб-сайт]. Електронні дані. Microsoft, 2025. URL: https://learn.microsoft.com/uk-ua/cpp/?view=msvc-170 (дата звернення: 21.02.2025).
2. cppreference.com [Електронний ресурс] : [Веб-сайт]. Електронні дані. cppreference.com, 2021. URL: https://en.cppreference.com/w/ (дата звернення: 21.02.2025).

5. ДОДАТКИ

Додаток А

УКРАЇНА

СВІДОЦТВО

про реєстрацію авторського права на твір

№ 116758

Комп'ютерна програма «Бібліотека для роботи з матрицями на С++ «MATRIX»

(вид, назва твору)

Автор(и) **Гарячевська Ірина Василівна, Протектор Денис Олегович**

(повне ім'я, псевдонім (за наявності))

Дата реєстрації 6 березня 2023 р.

Директор
Державної організації
«Український національний
офіс інтелектуальної власності
та інновацій» **Олена ОРЛЮК**

УКРНОІВІ

Електронне навчальне видання комбінованого використання
Можна використовувати у локальному та мережному режимі

Протектор Денис Олегович
Гарячевська Ірина Василівна

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ
Бібліотека «MATRIX» для роботи з матрицями на C++

Навчально-методичний посібник
для здобувачів вищої освіти за спеціальністю
Е6 «Прикладна фізика та наноматеріали»

В авторській редакції

Підписано до розміщення 21.05.2025. Гарнітура Times New Roman.
Ум. друк. арк. 6,01. Обсяг 3,438 Мб. Зам. № 412/25.

Харківський національний університет імені В. Н. Каразіна,
61022, м. Харків, майдан Свободи, 4.
Свідоцтво суб'єкта видавничої справи ДК № 3367 від 13.01.2009
Видавництво ХНУ імені В. Н. Каразіна