AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Macierz kwadratowa

Autor:

Marcin Dudek
Mateusz Basiaga

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Celem projektu jest stworzenie klasy `matrix` w języku C++, która pozwala na dynamiczne zarządzanie kwadratowymi macierzami o wymiarach $n \times n$. Klasa ta powinna zapewniać szeroki zakres funkcjonalności, takich jak alokacja pamięci, operacje na elementach macierzy, generowanie macierzy specjalnych oraz obsługa operatorów. Ostatecznym rezultatem jest stworzenie wydajnego i funkcjonalnego narzędzia do pracy z macierzami, które spełnia wszystkie wymagania zadania.

1.1. Cel pracy

Podstawowym celem jest stworzenie modułu `matrix`, który:

- umożliwia dynamiczne alokowanie i zwalnianie pamięci na stercie,
- realizuje operacje takie jak dodawanie, mnożenie, transpozycja oraz inne transformacje macierzy,
- zapewnia obsługę wyjątków i mechanizmy kontroli poprawności operacji (np. mnożenie macierzy musi spełniać wymagania matematyczne co do wymiarów),
- jest w pełni przetestowany poprzez zestaw scenariuszy uruchamianych z poziomu funkcji `main`.

1.2. Przewidywane wyniki

Projekt ma dostarczyć:

- Wydajną i poprawną implementację klasy `matrix`, która spełnia wszystkie wymienione wymagania.
- Testy sprawdzające poprawność działania klasy dla macierzy o różnych wymiarach, w tym dużych $n > 30$.
- Dokumentację wygenerowaną przy użyciu Doxygen oraz rozbudowany opis w LaTeX, zawierający analizę funkcjonalności i trudności napotkanych podczas implementacji.
- Praktyczne doświadczenie w wykorzystaniu GitHub Copilot jako narzędzia wspierającego programowanie oraz w pracy zespołowej z użyciem systemu kontroli wersji Git.

1.3. Zakres projektu

Projekt obejmuje:

- Implementację klasy `matrix` w języku C++ w osobnym pliku źródłowym.
- Funkcję `main` testującą wszystkie zaimplementowane funkcjonalności.
- Mechanizmy alokacji dynamicznej, zarządzania pamięcią i zabezpieczenia przed błędnymi operacjami.
- Obsługę przeciążonych operatorów matematycznych dla macierzy.
- Dokumentację w formacie LaTeX oraz automatycznie generowaną dokumentację kodu przy pomocy Doxygen.
- Analizę wykorzystania narzędzi AI w procesie tworzenia projektu oraz wpływ AI na efektywność pracy.

Projekt ten pozwoli nie tylko na zdobycie praktycznych umiejętności programistycznych, ale również na pogłębienie wiedzy w zakresie tworzenia dokumentacji oraz efektywnej pracy zespołowej.

2. Analiza problemu

2.1. Zastosowania macierzy

Macierze są fundamentalnym narzędziem matematycznym używanym w wielu dziedzinach nauki i technologii. Przykładowe zastosowania obejmują:

- Rozwiązywanie układów równań liniowych w algebrze,
- Grafika komputerowa – transformacje, przekształcenia 3D,
- Analiza danych – metody PCA (Principal Component Analysis),
- Modelowanie systemów dynamicznych w fizyce i inżynierii,
- Sztuczna inteligencja – obliczenia w sieciach neuronowych.

W naszym projekcie klasa `matrix` dostarczy zestaw narzędzi pozwalających na implementację wybranych operacji matematycznych i przekształceń. Zostanie także przetestowana jej wydajność i poprawność działania w środowisku C++.

2.2. Opis działania programu

Program realizowany w ramach projektu implementuje klasę `matrix`, która pozwala na dynamiczne zarządzanie macierzami oraz wykonywanie operacji matematycznych. Sposób działania programu można podzielić na następujące kroki:

1. Dynamiczne zaalokowanie pamięci dla macierzy $n \times n$.
2. Wypełnienie macierzy losowymi wartościami, wartościami przekątnymi, wierszami lub kolumnami zgodnie z wybranym algorytmem.
3. Wykonywanie operacji matematycznych, takich jak dodawanie, mnożenie czy transpozycja.
4. Zabezpieczenie programu przed błędnymi operacjami, np. próbą mnożenia macierzy o niezgodnych wymiarach.
5. Testowanie i wizualizacja wyników działania programu w konsoli lub poprzez zapis do pliku.

2.3. Przykład działania – mnożenie macierzy

Mnożenie macierzy odbywa się według następującego schematu: Dane są macierze:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

Iloczyn macierzy $C = A \cdot B$ obliczamy według wzoru:

$$C(i,j) = \sum_{k=1}^{n} A(i,k) \cdot B(k,j).$$

Dla podanych macierzy obliczenia wyglądają następująco:

$$C(1,1) = 1 \cdot 5 + 2 \cdot 7 = 19, \quad C(1,2) = 1 \cdot 6 + 2 \cdot 8 = 22,$$

$$C(2,1) = 3 \cdot 5 + 4 \cdot 7 = 43, \quad C(2,2) = 3 \cdot 6 + 4 \cdot 8 = 50.$$

Wynikowa macierz C to:

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}.$$

2.4. Wykorzystanie narzędzi AI i systemu kontroli wersji Git

W projekcie stosowane są następujące narzędzia:

- **GitHub Copilot**: Narzędzie wspomagające pisanie kodu poprzez sugestie generowane przez sztuczną inteligencję. Copilot przyspiesza proces implementacji, ale wymaga uważnej weryfikacji wygenerowanego kodu.
- **Git**: System kontroli wersji umożliwiający efektywną współpracę w zespole. Git pozwala na śledzenie zmian w projekcie, pracę nad różnymi funkcjonalnościami w oddzielnych gałęziach oraz łatwe rozwiązywanie konfliktów kodu.

Oba narzędzia są kluczowe w projekcie, wspierając zarówno implementację kodu, jak i jego organizację w repozytorium.

3. Projektowanie

W projekcie implementacji klasy `matrix` wykorzystano następujące narzędzia:

3.1. Narzędzia programistyczne

- **Język programowania i kompilator:** Projekt został napisany w języku **C++**, a do kompilacji kodu wykorzystano narzędzie **CMake**, które umożliwia zarządzanie konfiguracją procesu budowania oraz integrację z bibliotekami zewnętrznymi.

- **Google Test:** W celu przetestowania poprawności implementacji metod klasy `matrix` użyto frameworka **Google Test**. Testy obejmują weryfikację operacji matematycznych (np. dodawanie, mnożenie) oraz transformacji macierzy (np. transpozycja).

- **Doxygen:** Do generowania dokumentacji kodu źródłowego wykorzystano narzędzie **Doxygen**, które pozwala na automatyczne tworzenie szczegółowego opisu funkcji, zmiennych i struktur projektu.

3.2. Kontrola wersji

Kod źródłowy projektu jest zarządzany za pomocą systemu kontroli wersji **Git**, a repozytorium znajduje się na platformie **GitHub** pod adresem: https://github.com/Me-Phew/square-matrix. Zastosowano standardowe praktyki zarządzania kodem, takie jak regularne commitowanie zmian i utrzymywanie głównej gałęzi w stanie gotowym do użycia.

3.3. Specyfikacja kompilatora

W projekcie wykorzystano następujące ustawienia kompilatora:

- Standard języka: **C++17**.

- Plik konfiguracyjny `CMakeLists.txt` definiuje właściwości kompilacji oraz linkowanie z biblioteką Google Test.

3.4. Sposób użycia narzędzi AI

W trakcie implementacji projektu wspierano się narzędziem **GitHub Copilot**, które oferowało sugestie dotyczące składni i implementacji funkcji. Narzędzie to było szczególnie pomocne w przyspieszeniu procesu tworzenia kodu oraz w poprawie czytelności. Szczegóły wykorzystania AI oraz analiza jego efektywności zostaną omówione w kolejnym rozdziale.

4. Implementacja

W tej sekcji przedstawiono szczegóły implementacji klasy `matrix`, omówiono najważniejsze fragmenty kodu oraz wyniki uzyskane w trakcie działania programu.

4.1. Opis implementacji

Klasa `matrix` została zaprojektowana z myślą o dynamicznym zarządzaniu pamięcią dla kwadratowych macierzy o wymiarach $n \times n$. Kluczowe elementy implementacji obejmują:

- **Konstruktor i destruktor:** Klasa wykorzystuje konstruktor do alokacji pamięci oraz destruktor do jej zwalniania, co zapobiega wyciekom pamięci.
- **Operacje matematyczne:** Klasa obsługuje dodawanie, mnożenie i transpozycję macierzy. Metody te są zaimplementowane w sposób uwzględniający zgodność wymiarów macierzy.
- **Walidacja danych:** Operacje na macierzach są zabezpieczone przed nieprawidłowymi operacjami, takimi jak próba mnożenia macierzy o różnych wymiarach.

4.2. Fragmenty kodu

Poniżej zaprezentowano najważniejsze fragmenty kodu, które ilustrują implementację wybranych funkcji:

4.2.1. Konstruktor klasy `matrix`

Konstruktor klasy odpowiada za dynamiczną alokację pamięci **(Listing 1)**:

```
Matrix::Matrix(size_t n) : size(n) {
    data = new double*[size];
    for (size_t i = 0; i < size; ++i) {
        data[i] = new double[size]();
    }
}
```

Listing 1. Konstuktor klasy

4.2.2. Operacja mnożenia macierzy

Poniżej przedstawiono implementację funkcji realizującej mnożenie dwóch macierzy **(Listing 2)**:

```
Matrix Matrix::operator*(const Matrix& other) const {
    if (size != other.size) {
        throw std::invalid_argument("Incompatible matrix dimensions
    .");
    }
    Matrix result(size);
    for (size_t i = 0; i < size; ++i) {
        for (size_t j = 0; j < size; ++j) {
            for (size_t k = 0; k < size; ++k) {
                result.data[i][j] += data[i][k] * other.data[k][j];
            }
        }
    }
    return result;
}
```

Listing 2. Implementacja funkcji `operator`

4.2.3. Transpozycja macierzy

Funkcja transponuje macierz, zamieniając wiersze z kolumnami **(Listing 3)**:

```
Matrix Matrix::transpose() const {
    Matrix result(size);
    for (size_t i = 0; i < size; ++i) {
        for (size_t j = 0; j < size; ++j) {
            result.data[j][i] = data[i][j];
        }
    }
    return result;
}
```

Listing 3. Implementacja fukncji `Transpose`

4.3. GitHub Copilot

4.4. Trudności i ograniczenia w pracy z AI

W trakcie implementacji z użyciem Copilot napotkaliśmy zarówno zalety, jak i wyzwania związane z wykorzystaniem sztucznej inteligencji. Poniżej podsumowano kluczowe aspekty współpracy z Copilotem.

4.4.1. W czym AI sobie nie radziła?

Podczas pracy z Copilotem zauważyliśmy kilka problemów:

1. **Nieprecyzyjne sugestie w bardziej złożonych algorytmach.**
   AI często proponowała błędne implementacje bardziej skomplikowanych funkcji, takich jak obliczanie wyznacznika macierzy czy mnożenie macierzy. Na przykład pierwotnie wygenerowany kod dla mnożenia macierzy nie uwzględniał sumowania elementów w odpowiednich iteracjach **(Listing 4)**:

```
    SquareMatrix multiply(const SquareMatrix &other) const {
        if (size != other.size) {
            throw std::invalid_argument("Matrix dimensions must
    match");
        }
        SquareMatrix result(size);
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                result.matrix[i][j] = matrix[i][j] * other.
   matrix[i][j]; // BLAD
            }
        }
        return result;
    }

```

Listing 4. Błędna implementacja mnożenia macierzy

   W tym przypadku AI błędnie uznała, że mnożenie elementów w tej samej pozycji w wierszach i kolumnach wystarczy, zamiast zastosować klasyczny algorytm mnożenia macierzy.

   Poprawna implementacja wymagała ręcznej ingerencji **(Listing 5)**:

```
    SquareMatrix multiply(const SquareMatrix &other) const {
        if (size != other.size) {
```

```
3             throw std::invalid_argument("Matrix dimensions must
    match");
4         }
5         SquareMatrix result(size);
6         for (int i = 0; i < size; ++i) {
7             for (int j = 0; j < size; ++j) {
8                 result.matrix[i][j] = 0;
9                 for (int k = 0; k < size; ++k) {
10                    result.matrix[i][j] += matrix[i][k] * other
    .matrix[k][j];
11                }
12            }
13        }
14        return result;
15    }
16
```

Listing 5. Poprawna implementacja mnożenia macierzy

2. **Nieoptymalne zarządzanie pamięcią.**
   Copilot czasami sugerował wykorzystanie statycznych tablic zamiast dynamicznej alokacji pamięci, co ograniczało skalowalność programu. Ręczna refaktoryzacja była konieczna, aby zastosować elastyczną strukturę, taką jak `std::vector`.

4.4.2. Jakie błędy popełniała AI?

- **Nieprawidłowa obsługa wyjątków:** W wygenerowanym kodzie brakowało odpowiednich rzutów wyjątków w funkcjach, co prowadziło do nieprzewidzianego zachowania w sytuacjach brzegowych.
- **Niejasne komentarze:** AI generowała komentarze, które były zbyt ogólne i mało pomocne, np. `// Add matrices`, co wymagało ręcznej edycji.

4.4.3. W czym AI pomogła?

Mimo trudności, Copilot znacząco przyspieszył proces tworzenia kodu poprzez:

- **Automatyzację prostych funkcji:** AI dobrze radziła sobie z generowaniem podstawowych metod, takich jak akcesory (`getElement` i `setElement`) oraz funkcji wyświetlających macierz.
- **Sugestie składniowe:** W wielu przypadkach Copilot znacząco ułatwił tworzenie pętli i warunków dzięki precyzyjnym sugestiom składniowym.

Przykład wygenerowanej poprawnej funkcji wyświetlania macierzy **(Listing 6)**:

```
void display() const {
    for (const auto &row : matrix) {
        for (const auto &element : row) {
            std::cout << element << " ";
        }
        std::cout << std::endl;
    }
}
```

Listing 6. Funkcja wyświetlająca macierz

4.5. Wnioski z pracy z Copilotem

Praca z AI, takim jak Copilot, była cennym doświadczeniem, które pokazało zarówno mocne, jak i słabe strony tego narzędzia. Sztuczna inteligencja okazała się bardzo przydatna przy generowaniu kodu dla prostych operacji oraz w sytuacjach, gdy kluczowa była znajomość składni języka C++. Jednak w przypadku bardziej złożonych algorytmów konieczne było ręczne wprowadzanie poprawek, a także samodzielne projektowanie struktury programu.

W kolejnych projektach warto korzystać z Copilota jako narzędzia wspomagającego, ale z zachowaniem krytycznego podejścia do wygenerowanego kodu.

4.6. Wyniki działania programu

Przeprowadzone testy jednostkowe w środowisku Google Test wykazały, że:

- Operacje matematyczne, takie jak dodawanie, mnożenie i transpozycja, zostały zaimplementowane poprawnie.
- Program prawidłowo obsługuje sytuacje wyjątkowe, np. próby mnożenia macierzy o różnych wymiarach.
- Algorytm efektywnie działa dla macierzy o wymiarach do 1000×1000.

4.7. Podsumowanie

Implementacja klasy `matrix` została zrealizowana zgodnie z założeniami projektowymi. Dzięki zastosowaniu dynamicznej alokacji pamięci oraz obsłudze wyjątków, program jest elastyczny i odporny na nieprzewidziane błędy. Wyniki testów potwierdzają poprawność oraz wydajność zaimplementowanego algorytmu.

5. Wnioski

Realizacja projektu implementacji klasy `matrix` umożliwiła zgłębienie praktycznych aspektów programowania w języku C++. W ramach pracy osiągnięto następujące cele:

- Zaprojektowano i zaimplementowano klasę obsługującą operacje matematyczne na kwadratowych macierzach o wymiarach $n \times n$, z zachowaniem zasad dynamicznego zarządzania pamięcią.
- Opracowano efektywne metody realizujące operacje dodawania, mnożenia oraz transpozycji macierzy, z uwzględnieniem walidacji poprawności danych wejściowych.
- Przeprowadzono testy jednostkowe, które potwierdziły poprawność implementacji oraz wydajność algorytmów w różnych scenariuszach, w tym dla dużych macierzy.

5.1. Wnioski techniczne

- **Znaczenie zarządzania pamięcią:** Projekt ukazał kluczową rolę poprawnej alokacji i dealokacji pamięci w zapobieganiu wyciekom i problemom z wydajnością w programach o dużej złożoności.
- **Obsługa wyjątków:** Implementacja walidacji wejścia oraz mechanizmów obsługi wyjątków pozwoliła zwiększyć niezawodność programu.
- **Optymalizacja algorytmów:** Dla dużych macierzy (np. 1000×1000) istotne było zapewnienie efektywności obliczeniowej, co osiągnięto dzięki odpowiedniemu zagnieżdżeniu pętli i minimalizacji operacji pamięciowych.

5.2. Wnioski ogólne

Przeprowadzone prace nad projektem pozwoliły:

- Udoskonalić umiejętności programistyczne w zakresie języka C++ oraz posługiwania się narzędziami kontroli wersji (`git`).
- Zrozumieć praktyczne zastosowanie algorytmów matematycznych w rozwiązywaniu problemów inżynierskich.

- Zwiększyć świadomość na temat potencjału oraz ograniczeń narzędzi wspierających, takich jak GitHub Copilot, które przyspieszają proces tworzenia kodu, ale wymagają krytycznego podejścia i weryfikacji wygenerowanych wyników.

5.3. Potencjalne kierunki rozwoju

W przyszłości możliwe jest rozszerzenie projektu o:

- Implementację bardziej zaawansowanych operacji matematycznych, takich jak wyznacznik, rząd macierzy czy rozkład LU.
- Optymalizację algorytmów poprzez zastosowanie wielowątkowości, co mogłoby znacznie zwiększyć wydajność przy pracy z dużymi macierzami.
- Integrację z innymi projektami, np. wizualizacją danych macierzowych w interfejsie graficznym.

Podsumowując, projekt ten był wartościowym doświadczeniem edukacyjnym, łączącym teorię z praktyką, a uzyskane wyniki potwierdzają zarówno poprawność implementacji, jak i jej potencjał do dalszego rozwoju.

Bibliografia

[1] *Strona internetowa GitHub Copilot*. URL: https://github.com/features/copilot (term. wiz. 04. 12. 2024).

Spis rysunków

Spis tabel

Spis listingów