AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Klasa macierz z wykorzystaniem Github Copilot

Autor:
Dawid Michura
Dominik Jonik
Sebastian Tatara

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań

Celem niniejszego projektu jest opracowanie autorskiej biblioteki w języku C++ realizującej obsługę klasy `matrix`[1], reprezentującej macierz kwadratową. Projekt zakłada ręczną implementację silnika matematycznego bez udziału wyspecjalizowanych bibliotek zewnętrznych, co wymusza samodzielne zarządzanie pamięcią oraz logiką operacji algebraicznych. github-copilot Kluczowym aspektem zadania, odróżniającym je od standardowych projektów programistycznych, jest obowiązkowe wykorzystanie narzędzia GitHub Copilot[2] jako asystenta programowania, przy jednoczesnym całkowitym zakazie używania czatów opartych na AI (np. ChatGPT). Projekt kładzie nacisk na analizę współpracy człowiek-AI oraz dokumentowanie napotkanych trudności.

Kluczowym aspektem zadania, odróżniającym je od standardowych projektów programistycznych, jest obowiązkowe wykorzystanie narzędzia GitHub Copilot jako asystenta programowania, przy jednoczesnym całkowitym zakazie używania czatów opartych na AI (np. ChatGPT). Projekt kładzie nacisk na analizę współpracy człowiek-AI oraz dokumentowanie napotkanych trudności.

Projekt ma umożliwić:

- praktyczne zastosowanie inteligentnych wskaźników (`unique_ptr`, `shared_ptr` lub `weak_ptr`) w zarządzaniu dynamiczną pamięcią,
- implementację przeciążania operatorów w języku C++,
- zdobycie doświadczenia w pracy z asystentem kodu AI (GitHub Copilot),
- doskonalenie umiejętności pracy zespołowej (projekt dwuosobowy) z wykorzystaniem systemu kontroli wersji Git.

Projekt kończy się stworzeniem w pełni funkcjonalnej klasy, programu testującego oraz dokumentacją zawierającą analizę pracy z AI.

1.1. Cel i zakres pracy

Celem pracy jest napisanie klasy `matrix` operującej na macierzach o wymiarze $n \times n$, która oferuje następujące funkcjonalności:

1. **Zarządzanie pamięcią:** konstruktory (domyślny, kopiujący, alokujący), destruktor oraz metody dynamicznej alokacji (`alokuj`).

[1] Macierze - Wikipedia[1]
[2] Wprowadzenie do Github Copilot[2]

2. **Operacje na danych:** wstawianie i odczyt wartości, generowanie macierzy losowych, transpozycja, operacje na przekątnych (główna, nad i pod przekątną).

3. **Arytmetyka macierzowa:** dodawanie, odejmowanie i mnożenie macierzy ($A + B, A * B$) oraz operacje z liczbami całkowitymi (skalarami).

4. **Operatory inkrementacji i dekrementacji:** postfiksowe operacje zmieniające wartości wszystkich elementów macierzy.

5. **Operatory porównania:** sprawdzanie równości (==), czy macierz jest większa (>) lub mniejsza (<) w sensie logicznym zdefiniowanym w zadaniu.

6. **Obsługa wejścia/wyjścia:** przeciążenie operatora strumieniowego do wypisywania macierzy.

Dodatkowo zakres pracy obejmuje przetestowanie biblioteki na macierzach o rozmiarze $n \geq 30$ oraz zabezpieczenie przed niedozwolonymi operacjami matematycznymi.

1.2. Wymagania techniczne i inicjalizacja projektu

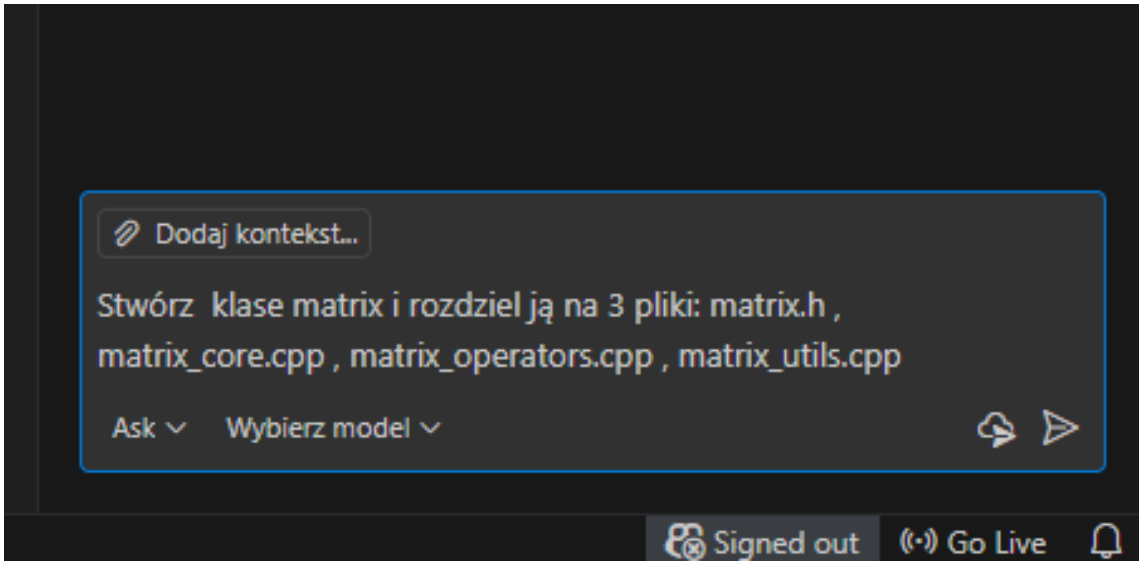
Aplikacja jest tworzona w języku C++ i musi być zgodna ze środowiskiem Visual Studio lub Visual Studio Code. Do prawidłowego przebiegu prac wymagane są:

- **Język:** C++ (zalecany standard C++14 lub nowszy ze względu na `unique_ptr`).
- **Środowisko:** Visual Studio / Visual Studio Code z zainstalowaną wtyczką GitHub Copilot.
- **Konto GitHub:** Zweryfikowane jako konto studenckie w celu uzyskania darmowego dostępu do Copilota.
- **Kontrola wersji:** Git[3] do pracy równoległej w parach.

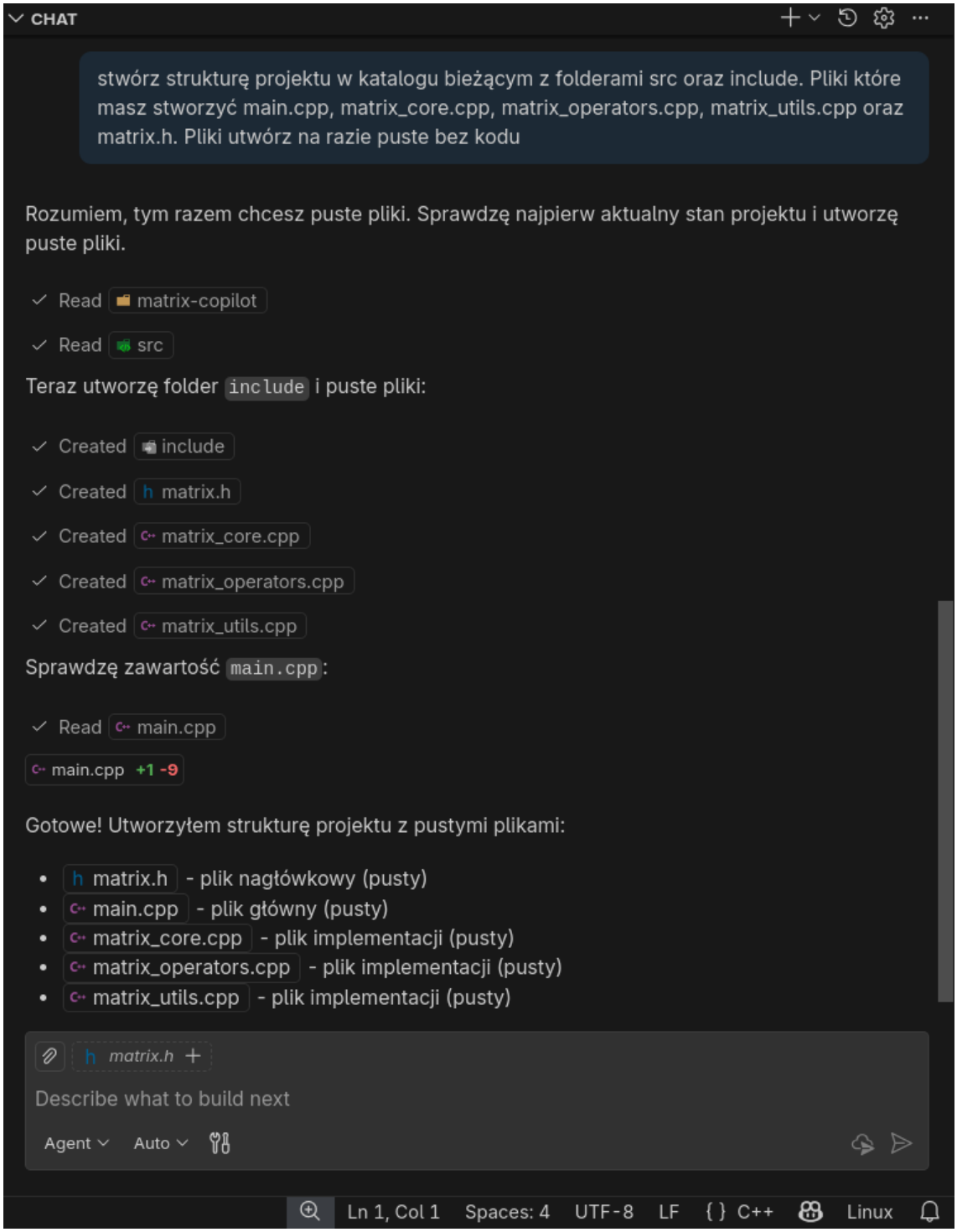
Proces inicjacji projektu rozpoczęto od precyzyjnego sformułowania zadania dla GitHub Copilot. Na Rysunku 1.1 widzimy zapytanie skierowane do AI, mające na celu ustalenie wstępnej struktury projektu z podziałem na pliki nagłówkowe i źródłowe.

Po zaakceptowaniu propozycji AI, pliki zostały automatycznie utworzone, co potwierdza Rysunek 1.2 (s. 7). Następnie w projekcie ustalono strukturę folderów `include` oraz `src`, co przedstawia Rysunek 1.3 (s. 8).

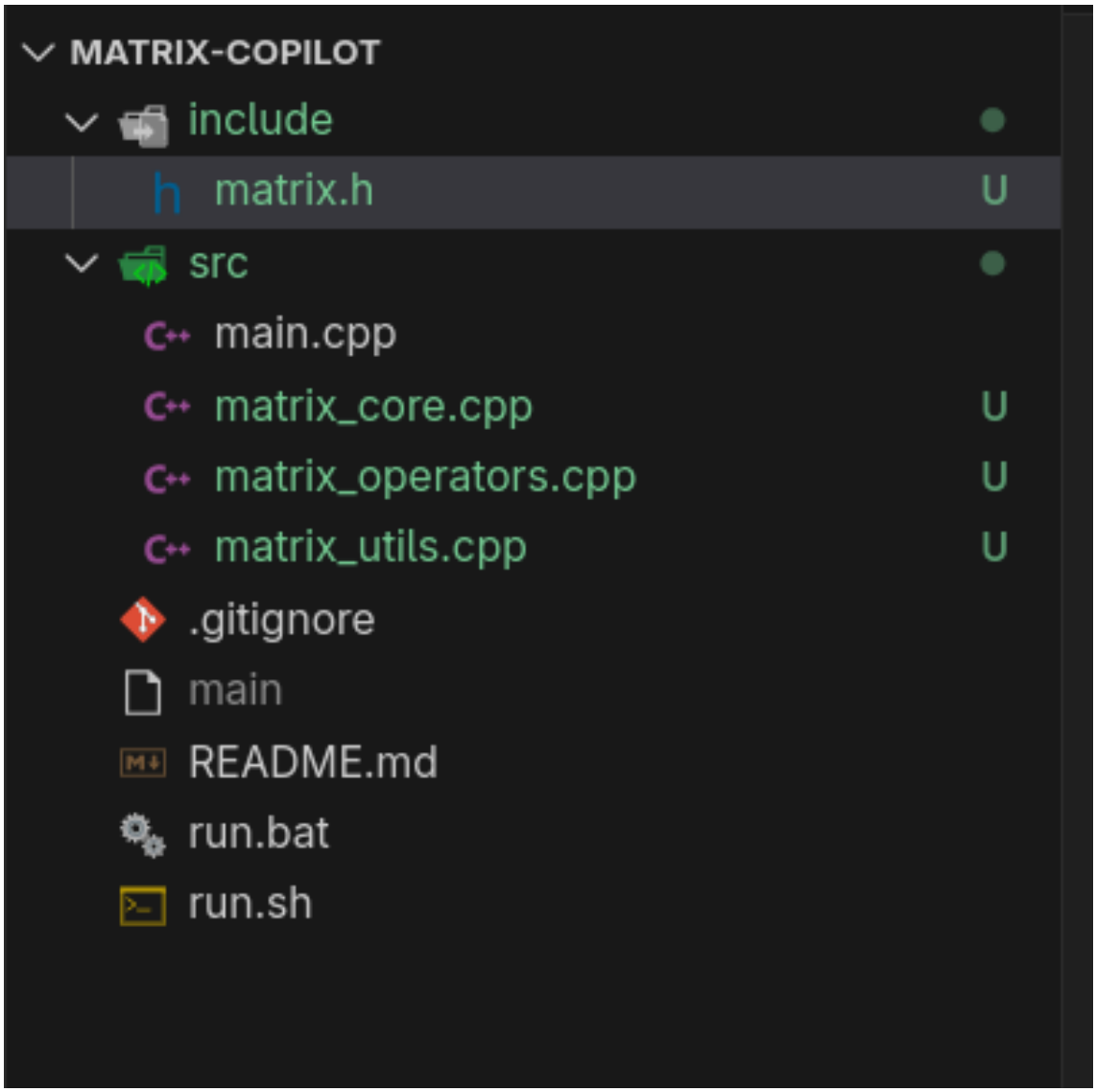
[3]Git - podstawy Gita [3]



Rys. 1.1. Zdefiniowanie głównego zadania dla GitHub Copilot: podział na pliki nagłówkowe i źródłowe



Rys. 1.2. Wykonanie instrukcji przez Copilot – utworzenie pustych plików projektu



Rys. 1.3. Ostateczna struktura projektu w Visual Studio Code, z podziałem na foldery `include` i `src`

2. Analiza problemu

Przed przystąpieniem do implementacji klasy `matrix`, konieczne jest przeanalizowanie sposobu reprezentacji danych w pamięci, logiki operacji matematycznych oraz specyfiki narzędzi AI, które mają wspomagać proces tworzenia kodu.

2.1. Charakterystyka macierzy i zarządzanie pamięcią

Macierz w kontekście tego projektu zdefiniowana jest jako kwadratowa tablica liczb całkowitych o wymiarach $n \times n$. Kluczowym wyzwaniem inżynierskim jest tutaj zarządzanie pamięcią, które nie może opierać się na statycznych tablicach ani kontenerach biblioteki standardowej (jak `std::vector`), lecz musi wykorzystywać dynamiczną alokację oraz inteligentne wskaźniki (`unique_ptr`, `shared_ptr` lub `weak_ptr`).

Zastosowanie inteligentnych wskaźników (ang. *smart pointers*) zmienia podejście do zarządzania cyklem życia obiektu:

- **Alokacja:** Pamięć dla n^2 elementów musi zostać zaalokowana dynamicznie.
- **Własność:** Użycie np. `std::unique_ptr` zapewnia wyłączną własność zasobu i automatyczne zwolnienie pamięci w destruktorze, eliminując ryzyko wycieków pamięci.
- **Reallokacja:** Metoda `alokuj(int n)` musi inteligentnie zarządzać zasobami – zwalniać pamięć i alokować nową tylko wtedy, gdy obecny bufor jest mniejszy niż wymagany rozmiar n. W przypadku nadmiaru pamięci, alokacja pozostaje bez zmian. [cite$_s tart$]

2.2. Ręczna analiza operacji macierzowych

Zrozumienie algorytmiki operacji na macierzach jest niezbędne do weryfikacji poprawności kodu generowanego przez GitHub Copilot. Przeanalizujmy dwa kluczowe przypadki: generowanie wzorca szachownicy oraz mnożenie macierzy.

2.2.1. Generowanie wzorca „Szachownica”

Metoda `szachownica()` ma wypełnić macierz zerami i jedynkami w naprzemiennym układzie. [cite$_s tart$]$Rozwamyprzykaddlan$=4, $podanywspecyfikacji$:

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

Analizując indeksy $(wiersz, kolumna)$ dla wartości 1:

- $(0, 1) \to 0 + 1 = 1$ (nieparzysta)
- $(1, 0) \to 1 + 0 = 1$ (nieparzysta)
- $(2, 3) \to 2 + 3 = 5$ (nieparzysta)

Dla wartości 0:

- $(0, 0) \to 0 + 0 = 0$ (parzysta)
- $(1, 1) \to 1 + 1 = 2$ (parzysta)

Wniosek implementacyjny: Wartość komórki to reszta z dzielenia sumy indeksów przez 2: $value = (x + y)\%2$.

2.2.2. Mnożenie macierzy ($A * B$)

Operacja mnożenia macierzy $C = A \times B$ jest zdefiniowana tylko wtedy, gdy liczba kolumn macierzy A jest równa liczbie wierszy macierzy B (w przypadku macierzy kwadratowych $n \times n$ warunek ten jest zawsze spełniony, o ile n jest takie samo). Element c_{ij} obliczamy jako iloczyn skalarny i-tego wiersza macierzy A i j-tej kolumny macierzy B:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

Złożoność obliczeniowa tej operacji wynosi $O(n^3)$. Należy zabezpieczyć program przed próbą mnożenia macierzy o różnych wymiarach.

2.3. Analiza narzędzi: GitHub Copilot i Git

Projekt wymaga specyficznego środowiska pracy, łączącego tradycyjną kontrolę wersji z nowoczesnym asystentem AI.

2.3.1. Rola GitHub Copilot

GitHub Copilot to narzędzie oparte na modelu językowym OpenAI Codex, które sugeruje fragmenty kodu w czasie rzeczywistym. W przeciwieństwie do czatów typu ChatGPT, Copilot integruje się bezpośrednio z IDE (Visual Studio). W projekcie Copilot ma za zadanie:

- Podpowiadać implementację metod na podstawie ich sygnatur i komentarzy w kodzie.
- Pomagać w pisaniu testów jednostkowych.

Istotnym elementem analizy jest dokumentowanie błędów popełnianych przez AI oraz sytuacji, w których Copilot nie radził sobie z logiką zadania. Zabronione jest korzystanie z zewnętrznych czatów AI.

2.3.2. Współpraca w systemie Git

Ze względu na dwuosobowy charakter projektu, system Git służy do synchronizacji prac realizowanych równolegle. Kluczowe aspekty współpracy obejmują:

- **Branching:** Tworzenie osobnych gałęzi dla implementacji poszczególnych operatorów.
- **Dokumentacja w LaTeX:** Wersjonowanie pliku dokumentacji wraz z kodem źródłowym, co pozwala na śledzenie postępów w opisie projektu.

3. Projektowanie

Na podstawie przeprowadzonej analizy wymagań, przystąpiono do fazy projektowej. Obejmuje ona wybór technologii, zaprojektowanie architektury systemu oraz zdefiniowanie strategii pracy zespołowej.

3.1. Wykorzystane narzędzia i technologie

- **Język programowania**: C++ (standard C++14 lub nowszy). Wybór podyktowany był wymaganiami projektu dotyczącymi zarządzania pamięcią za pomocą inteligentnych wskaźników (`std::unique_ptr`).
- **Kompilator**: MSVC (Visual Studio) lub g++ (MinGW/Linux).
- **Środowisko (IDE)**: Visual Studio lub Visual Studio Code z zainstalowaną wtyczką GitHub Copilot.
- **Kontrola wersji**: Git, do lokalnego zarządzania zmianami i śledzenia postępów.
- **Platforma hostingowa**: GitHub, do zdalnej współpracy i weryfikacji kont studenckich (dostęp do Copilot).
- **Dokumentacja**: LaTeX do przygotowania niniejszego dokumentu oraz Doxygen do generowania dokumentacji technicznej.

Zgodnie z założeniami, nie wykorzystano żadnych zewnętrznych bibliotek matematycznych. Cały silnik macierzowy został zaimplementowany od podstaw.

3.2. Architektura systemu i diagram klas

System został zaprojektowany w oparciu o zasady programowania obiektowego. Implementacja została podzielona na pliki nagłówkowe i źródłowe (`.h` / `.cpp`).

System składa się z następujących komponentów:

- **matrix** (Klasa główna): Odpowiada za całą logikę matematyczną i zarządzanie pamięcią.
  - Przechowuje dane w dynamicznej strukturze zarządzanej przez `unique_ptr` (lub inny inteligentny wskaźnik).

  - Implementuje konstruktory, destruktor oraz metodę `alokuj(int n)` do dynamicznej rezerwacji pamięci.
  - Zawiera metody operacyjne: `wstaw`, `losuj`, `diagonalna`, `szachownica` itd.
  - Udostępnia przeciążone operatory arytmetyczne $(+, -, *)$ oraz operatory porównania $(==, >, <)$.

- **main.cpp**: Plik sterujący. Zawiera funkcję `main()`, która przeprowadza testy jednostkowe zaimplementowanych funkcjonalności, sprawdzając poprawność obliczeń dla różnych rozmiarów macierzy (w tym $n \geq 30$).

3.3. Strategia pracy w systemie Git

Praca nad projektem w zespole dwuosobowym wymagała zdefiniowania przepływu pracy (workflow). Zdecydowano się na model *feature branching*.

1. Główna gałąź `main` (lub `master`) jest gałęzią stabilną.
2. Każda nowa funkcjonalność (np. „dodawanie macierzy", „obsługa pamięci") jest implementowana w dedykowanej gałęzi `feature/...`.
3. Po zakończeniu implementacji i przetestowaniu, gałąź `feature` jest scalana (merge) z gałęzią `main`.

Rysunek 3.1 (s. 13) przedstawia fragment grafu commitów, ilustrujący równoległą pracę nad różnymi gałęziami oraz ich późniejsze scalanie.

Rys. 3.1. Graf commitów z repozytorium GitHub ilustrujący pracę na gałęziach

4. Implementacja

W tym rozdziale przedstawiono kluczowe fragmenty kodu źródłowego klasy `matrix` oraz opisano proces implementacji. Szczególną uwagę poświęcono analizie współpracy z asystentem GitHub Copilot, dokumentując napotkane trudności i błędy generowane przez AI, a także proces rozwiązywania konfliktów w systemie kontroli wersji Git.

4.1. Implementacja kluczowych metod klasy matrix

Rdzeniem projektu jest dynamiczne zarządzanie pamięcią oraz operacje arytmetyczne. Poniżej przedstawiono wybrane fragmenty kodu ilustrujące przyjęte rozwiązania.

4.1.1. Zarządzanie pamięcią i alokacja

Kluczowym wyzwaniem projektu było ręczne zarządzanie pamięcią przy jednoczesnym wykorzystaniu nowoczesnych mechanizmów C++. W klasie `matrix` zastosowano wskaźnik `std::unique_ptr` zarządzający tablicą wskaźników, co pozwala na automatyczne zwolnienie pamięci w momencie niszczenia obiektu.

Kod 1 (s. 14) prezentuje metodę `alokuj`, która jest wywoływana przez konstruktory. Warto zwrócić uwagę na dwuetapową inicjalizację: najpierw alokowana jest tablica wskaźników na wiersze, a następnie w pętli alokowane są poszczególne wiersze macierzy. Zapewnia to poprawne odwzorowanie struktury 2D w pamięci.

```
1  void matrix::alokuj(std::size_t /*n*/) {
2      // Alokacja tablicy wskaznikow na wiersze
3      data = std::make_unique<double*[]>(rows);
4
5      // Alokacja poszczegolnych wierszy
6      for (int i = 0; i < rows; ++i) {
7          data[i] = new double[cols];
8      }
9
10     // Zerowanie pamieci
11     for (int i = 0; i < rows; ++i) {
12         for (int j = 0; j < cols; ++j) {
13             data[i][j] = 0.0;
14         }
15     }
```

```
}
```

Listing 1. Alokacja pamięci dla macierzy 2D przy użyciu unique_ptr

4.1.2. Algorytmika metod narzędziowych

Implementacja metod użytkowych, takich jak wypełnianie wzorcami (`przekatna`, `szachownica`), była realizowana przy wsparciu Copilota.

Ciekawym przypadkiem jest metoda generująca wzór szachownicy. Kod 2 (s. 15) pokazuje eleganckie rozwiązanie matematyczne zaproponowane w trakcie pracy. Zamiast skomplikowanych instrukcji warunkowych, wykorzystano właściwość parzystości sumy indeksów $(i + j)$. Jeśli suma jest parzysta, wstawiane jest 0, w przeciwnym razie 1. Pozwoliło to na uzyskanie pożądanego wzoru w jednej zagnieżdżonej pętli.

```
matrix& matrix::szachownica() {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            // Decyzja o wartosci na podstawie sumy indeksow
            if ((i + j) % 2 == 0) {
                data[i][j] = 0;
            } else {
                data[i][j] = 1;
            }
        }
    }
    return *this;
}
```

Listing 2. Algorytm generowania wzoru szachownicy

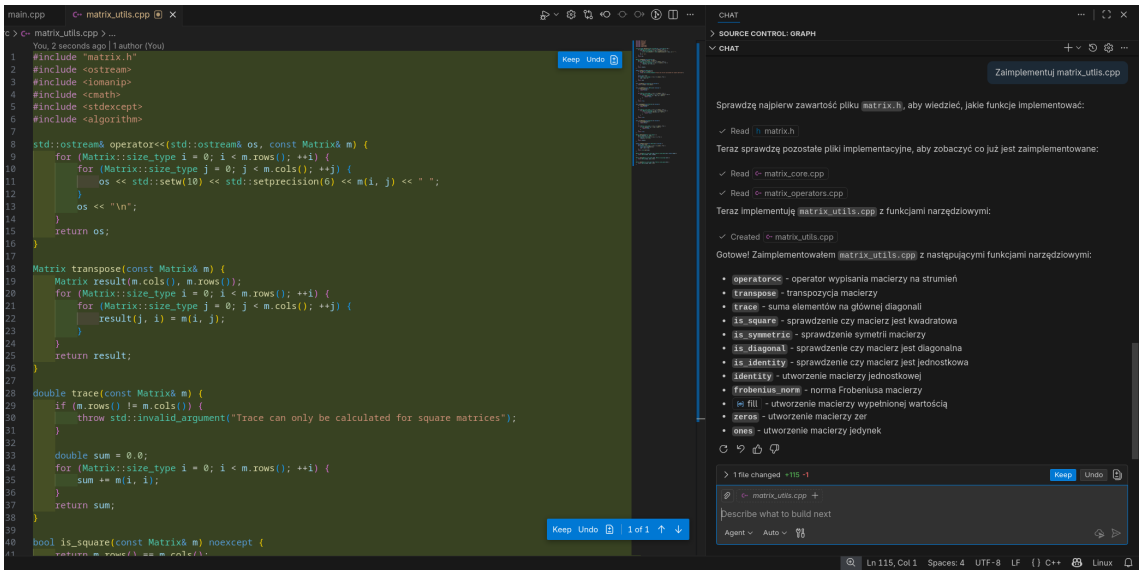
Na Rysunku 4.1 widzimy, jak Copilot generuje pozostałe metody narzędziowe na podstawie listy wymagań zawartej w komentarzach.

4.1.3. Implementacja operatorów

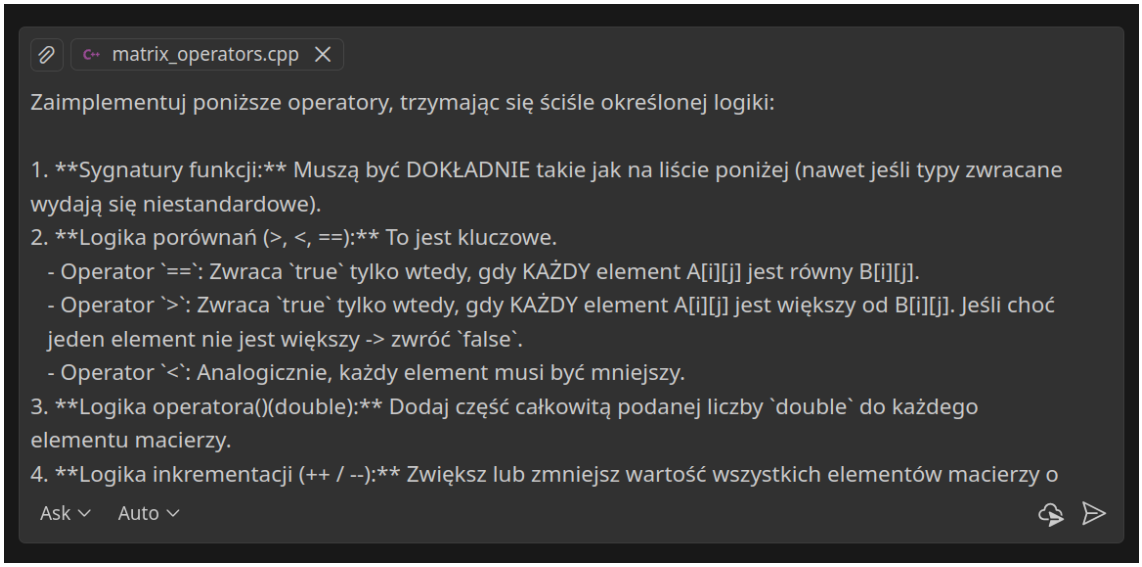
Implementacja przeciążonych operatorów arytmetycznych i porównania wymagała szczegółowego opisu logiki dla Copilota. Na Rysunku 4.2 widzimy zapytanie skierowane do AI w celu wygenerowania operatorów logicznych i postfiksowych.

4.2. Analiza współpracy z GitHub Copilot

Jak wskazano w wymaganiach, praca z Copilotem wymagała nie tylko wykorzystania jego możliwości, ale także dogłębnej analizy jego działania.



Rys. 4.1. Generowanie implementacji metod użytkowych w matrix_utils.cpp



Rys. 4.2. Prompt dla GitHub Copilot w celu wygenerowania operatorów porównania oraz postfiksowych

4.2.1. W czym AI pomogło?

Asystent AI był nieoceniony w generowaniu:

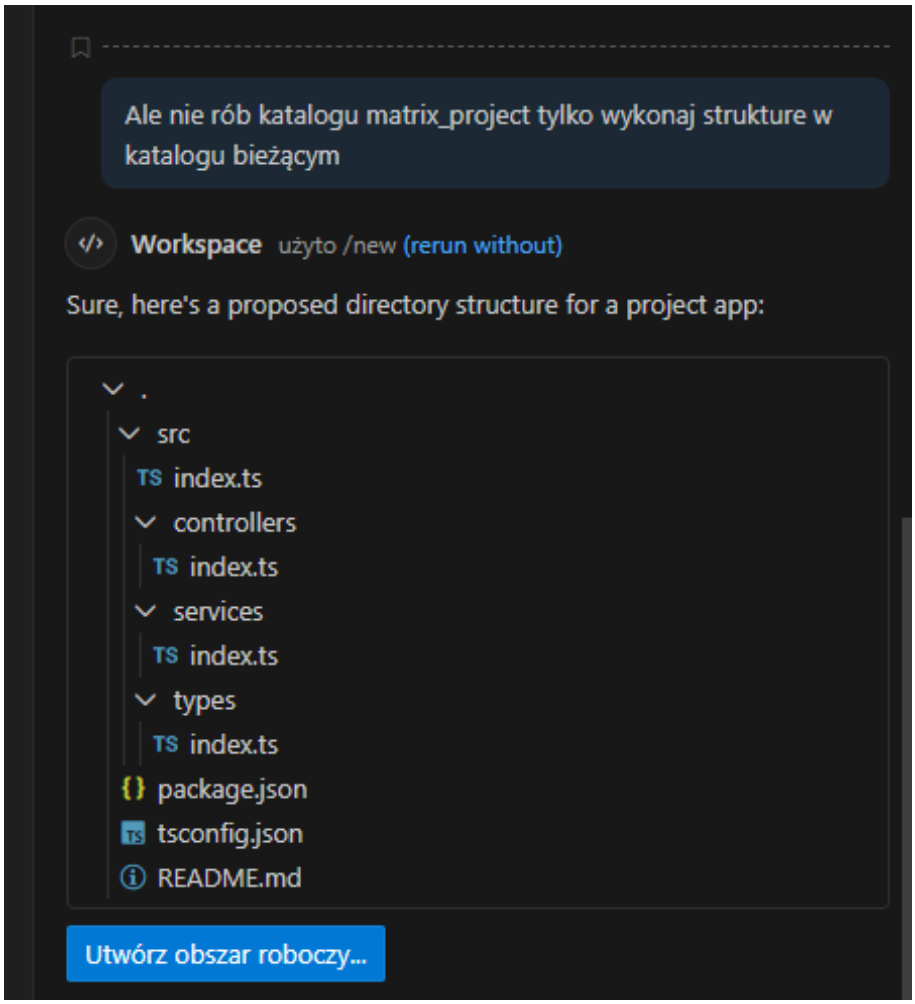
- **Struktury projektu** (Rys. 1.1, s. 6): Szybka inicjalizacja plików.
- **Powtarzalnego kodu (boilerplate):** Tworzenie pętli i warunków w metodach takich jak `wstaw`, `pokaz` czy `losuj`.
- **Dokumentacji:** Copilot automatycznie generował komentarze Doxygen do

większości sygnatur metod.

4.2.2. W czym AI sobie nie radziła i popełniało błędy?

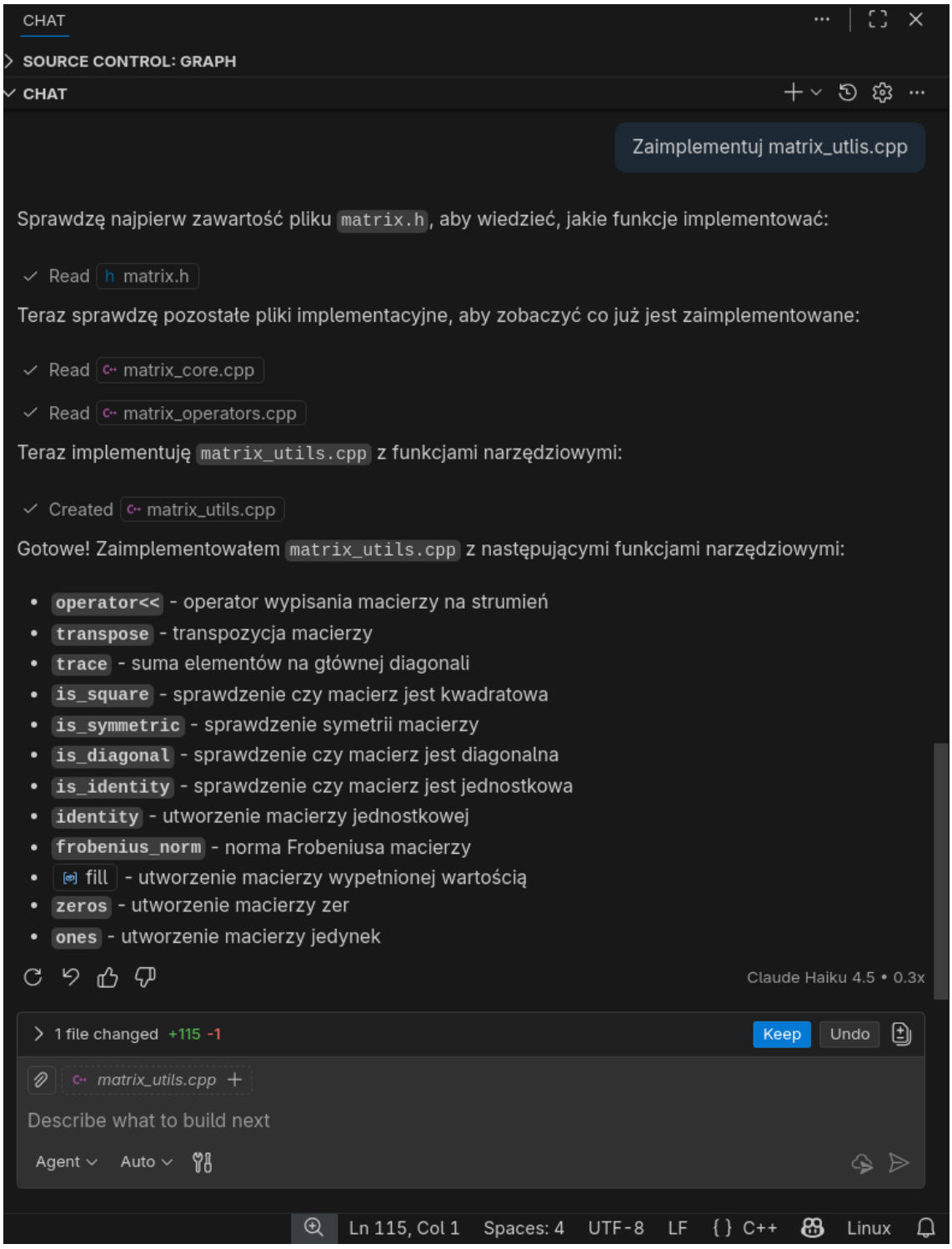
Największe trudności i błędy AI wystąpiły w logice wysokiego poziomu i braku kontekstu całego projektu:

- **Błąd Kontekstu Technologicznego (Maniana AI):** Na początku projektu, mimo pracy w plikach `.cpp`, Copilot zaproponował wygenerowanie kodu w JavaScript/TypeScript.



Rys. 4.3. Błąd Copilota: Propozycja struktury w języku TypeScript (Maniana AI)

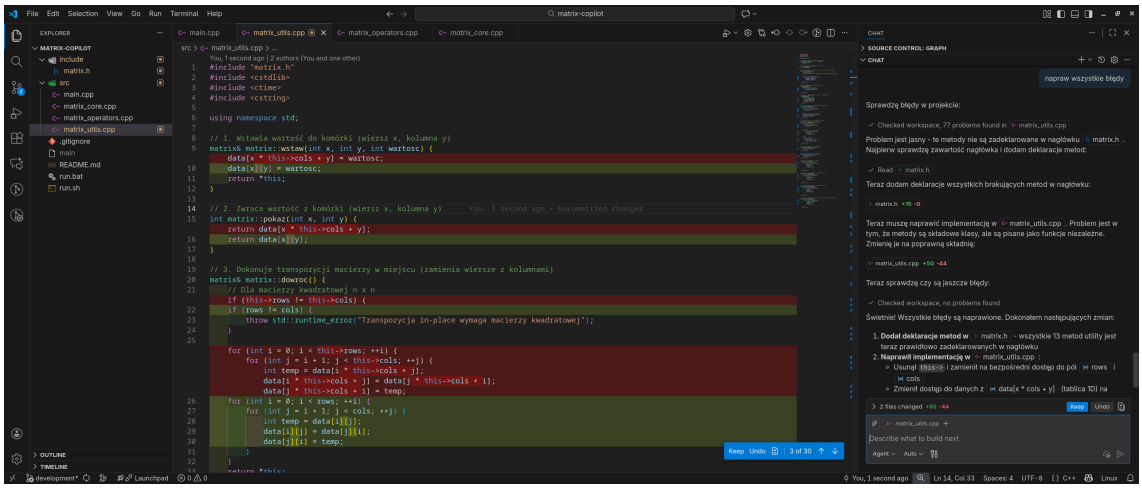
- **Nadmierna funkcjonalność:** AI często proponowało funkcje, które nie były wymagane w zadaniu (np. `trace`, `frobenius_norm`). [cite$_s$*tart*]*NaRysunku*4.4*widapropozycj* 88, 113].



Rys. 4.4. Propozycja niepotrzebnych metod przez Copilot w matrix_utils.cpp

- **Błędy Kompilacji i Braki Deklaracji:** Po zaimplementowaniu metod w plikach .cpp, Copilot nie zaktualizował pliku nagłówkowego. Na Rysunku 4.5 Copilot sam diagnozuje brak 77 deklaracji.

- **Indeksowanie 1D vs 2D:** Copilot wielokrotnie wymagał korygowania spo-



Rys. 4.5. Copilot diagnozuje błąd: brak deklaracji metod w matrix.h

sobu dostępu do danych. Ponieważ użyliśmy tablicy wskaźników (`double*[]`), poprawnym dostępem jest `data[i][j]`, podczas gdy AI często próbowało stosować spłaszczoną arytmetykę wskaźników `data[i * cols + j]`, co prowadziło do błędów kompilacji.

5. Wnioski

Realizacja projektu "Klasa Matrix"była dla naszego trzyosobowego zespołu nie tylko sprawdzianem umiejętności programistycznych w języku C++, ale przede wszystkim nowym doświadczeniem w zakresie inżynierii oprogramowania wspomaganej przez sztuczną inteligencję. Wszystkie założone cele, w tym implementacja własnego silnika matematycznego bez użycia bibliotek zewnętrznych, zostały zrealizowane pomyślnie.

Z perspektywy technicznej, największym wyzwaniem i jednocześnie sukcesem było poprawne zaimplementowanie dynamicznego zarządzania pamięcią przy użyciu inteligentnych wskaźników (`std::unique_ptr`). Decyzja ta pozwoliła nam uniknąć typowych błędów związanych z wyciekami pamięci, zrzucając odpowiedzialność za jej zwalnianie na kompilator. Własna implementacja metody `alokuj`, która inteligentnie decyduje o realokacji zasobów, dała nam głębszy wgląd w to, jak kosztowne operacyjnie jest zarządzanie stertą (heap) i jak ważna jest optymalizacja w tym obszarze.

Kluczowym aspektem projektu była praca z asystentem GitHub Copilot. Nasze odczucia co do tego narzędzia są mieszane, choć z przewagą pozytywnych.

- **Co Copilot ułatwił:** AI okazało się bezkonkurencyjne w generowaniu tzw. "boilerplate code"– powtarzalnych fragmentów, takich jak konstruktory, gettery, czy proste pętle wypełniające macierz. Ogromną pomocą było również automatyczne generowanie dokumentacji Doxygen, co zaoszczędziło nam wiele godzin żmudnej pracy.
- **Co Copilot utrudniał:** Narzędzie wielokrotnie traciło kontekst projektu. Zdarzały się sytuacje kuriozalne, jak propozycja struktury projektu w języku TypeScript zamiast C++, czy generowanie metod, które nie istniały w pliku nagłówkowym, co prowadziło do licznych błędów kompilacji. Zauważyliśmy również, że przy skomplikowanej logice (np. indeksowanie tablicy 1D jako 2D: $x \cdot n + y$), Copilot potrafił "halucynować", myląc indeksy, co wymuszało na nas szczegółową weryfikację każdej linii kodu.

Wniosek jest jasny: Copilot to świetny "inteligentny asystent", ale nie zastąpi on programisty rozumiejącego architekturę kodu.

Praca w grupie trzyosobowej wymusiła na nas rygorystyczne podejście do systemu kontroli wersji Git. Podział projektu na logiczne moduły (`core`, `operators`, `utils`) ułatwił równoległą pracę.

Podsumowując, projekt ten pozwolił nam stworzyć solidną bibliotekę macierzową. Jako kierunek dalszego rozwoju widzimy przede wszystkim przekształcenie klasy `matrix` w szablon (template), co pozwoliłoby na operacje na liczbach zmiennoprzecinkowych (`double`) czy zespolonych, czyniąc bibliotekę użyteczną w rzeczywistych zastosowaniach inżynierskich.

Bibliografia

[1] *Macierze - Wikipedia*. URL: https://pl.wikipedia.org/wiki/Macierz (term. wiz. 11. 12. 2025).

[2] *Wprowadzenie do Github Copilot*. URL: https://sii.pl/blog/wprowadzenie-do-github-copilot/?gad_source=1&gad_campaignid=23274849327&gclid=CjOKCQiAuvTJBhCwARIsAL6DemipDXqiQUWgE-wK3esM-yXGZOWGvbMbhMPEhQ_GiMaN9uwbVtigYBkaAl5iEALw_wcB (term. wiz. 11. 12. 2025).

[3] *Git - podstawy Gita*. URL: https://git-scm.com/book/pl/v2/Pierwsze-kroki-Podstawy-Git (term. wiz. 29. 11. 2025).

Spis rysunków

Spis tabel

Spis listingów