Omówienie zagadnienia

Zadanie polegało na zastosowaniu metody potęgowej oraz algorytmu QR na macierzy *M*

$$M = \begin{pmatrix} 8 & 1 & 0 & 0 \\ 1 & 7 & 2 & 0 \\ 0 & 2 & 6 & 3 \\ 0 & 0 & 3 & 5 \end{pmatrix}$$

Podczas stosowania metody potęgowej należało znaleźć największą co do modułu wartość własną macierzy *M* oraz odpowiadający jej wektor własny. Działanie tego programu można było usprawnić poprzez zastosowanie metody Wilkinsona, która miała za zadanie poprawić czas osiągnięcia zadowalającej zbieżności metody potęgowej do dokładnego rozwiązania.

Następnie podczas stosowania algorytmu QR bez przesunięć należało znaleźć wszystkie wartości własne macierzy *M* oraz przedstawić ich ewolucję – zbieżność do dokładnych wartości własnych – podczas kolejnych iteracji.

Dokładne wartości własne oraz odpowiadające im wektory obliczałem z wykorzystaniem biblioteki Eigen.

Metoda potęgowa

Podczas implementacji metody potęgowej za sensowny warunek stopu pętli przyjąłem, że różnica pomiędzy aktualną wartością własną a wartością uzyskaną w poprzedniej iteracji musi być mniejsza niż *1e-16*.

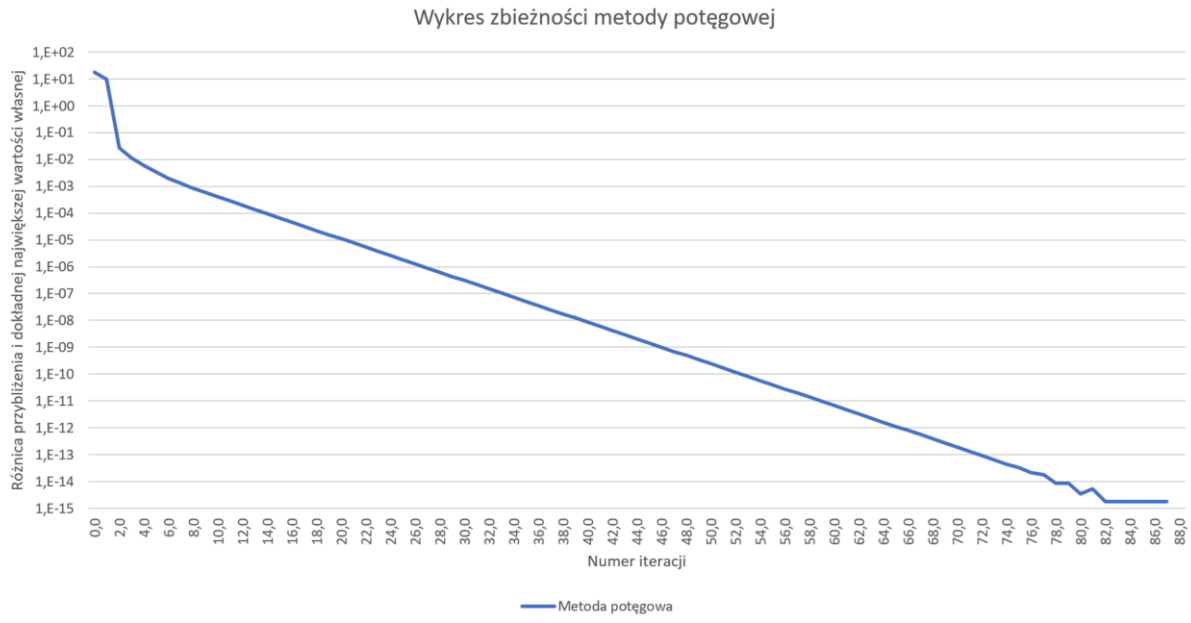
Z takim założeniem otrzymałem następujące wyniki:

- największa co do modułu wartość własna: 9.74239
- odpowiadający jej wektor własny to: $(0.332723, \quad 0.579734, \quad 0.628568, \quad 0.397627)^T$
- potrzebne iteracje: 88

Powyższe wyniki sprawdziłem za pomocą biblioteki Eigen, były one następujące:

- największa co do modułu wartość własna: 9.74239
- odpowiadający jej wektor własny to: $(0.332723, \quad 0.579734, \quad 0.628568, \quad 0.397627)^T$

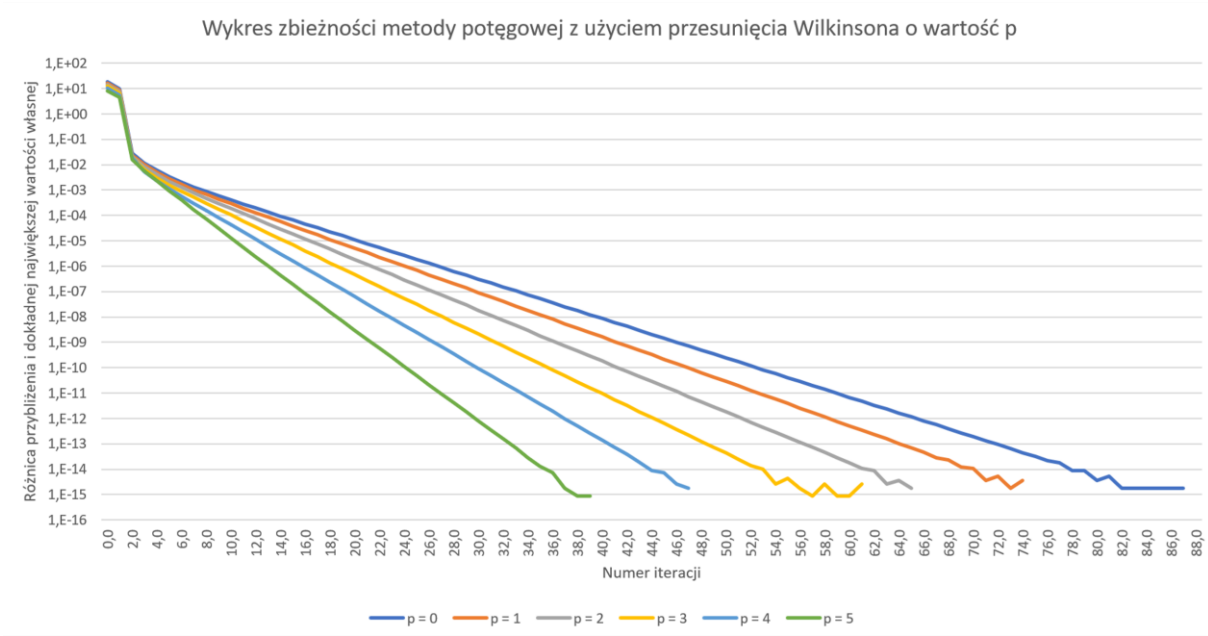
Poniżej przedstawiam wykres wykonany w skali logarytmicznej, ilustrujący zbieżność metody potęgowej podczas kolejnych iteracji.



Jak możemy zauważyć, błąd sukcesywnie malał, a program potrzebował 88 iteracji, aby osiągnąć zbieżność na przyjętym przeze mnie poziomie.

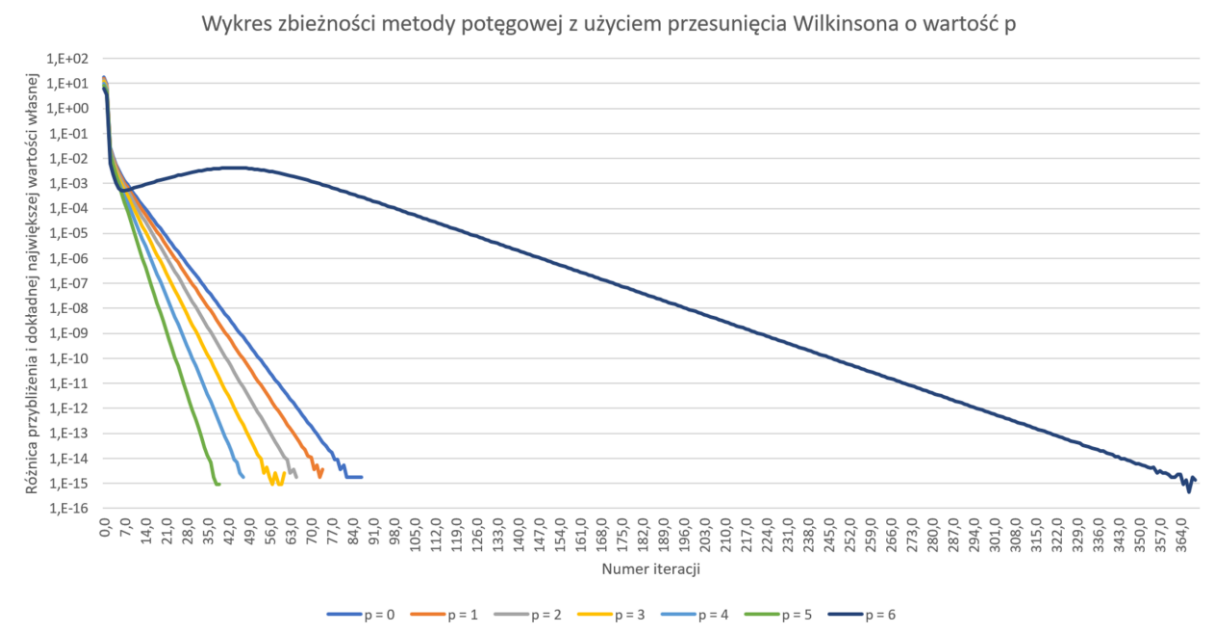
Po napisaniu kodu metody potęgowej i otrzymaniu wyżej przedstawionych wyników, dodałem do funkcjonalności programu możliwość użycia metody Wilkinsona. Umożliwiła ona zmniejszenie potrzebnej ilości iteracji, żeby uzyskać zadaną zbieżność.

Metoda Wilkinsona opiera się o przesunięcie macierzy M o pewną wartość p. Uruchomiłem więc program, wprowadzając przy tym kilka różnych wartości zmiennej p. Poniżej przedstawiam wykres (również w skali logarytmicznej), który ilustruje zbieżność metody potęgowej podczas kolejnych iteracji.



Jak można zauważyć, dla przesunięcia $p = 5$ otrzymaliśmy ponad dwukrotnie mniejszą ilość iteracji.

Po otrzymaniu tego rewelacyjnego wyniku, zacząłem się zastanawiać jaka jest najlepsza możliwa wartość p, aby otrzymać jak największą optymalizację programu. Wykonałem więc kolejny test – wyniki przedstawiam na bliźniaczym wykresie poniżej.



Jak się okazało, dla $p = 6$ pojawiło się ogromne zaburzenie wyniku. Na tej podstawie stwierdziłem, że $p = 5$ jest najlepszym możliwym przesunięciem, które pozwoli przyśpieszyć działanie programu.

Wnioski

Zastosowanie przesunięcia Wilkinsona do programu korzystającego z metody potęgowej, pozwala na znaczące skrócenie czasu działania programu – w tym przypadku ilość iteracji dla $p = 5$ zapewniła ponad dwukrotnie mniejszą ilość iteracji. Po dokonaniu przesunięcia dla $p = 6$ i otrzymania dużego zaburzenia wyniku, stwierdzam, że przesunięcie o wartość 5 zapewnia najlepszy rezultat w tym zadaniu.

Algorytm QR

Podczas implementacji algorytmu QR za sensowny warunek stopu pętli przyjąłem, że przekształcana macierz musi jak najbardziej upodobnić się do macierzy trójkątnej górnej. Stwierdziłem więc, że suma elementów znajdujących się pod diagonalą musi być mniejsza niż *1e-15*.

Po osiągnięciu takiego warunku odczytałem następujące wartości własne z diagonali:

- 9.74239
- 8.14772
- 6.03172
- 2.07816

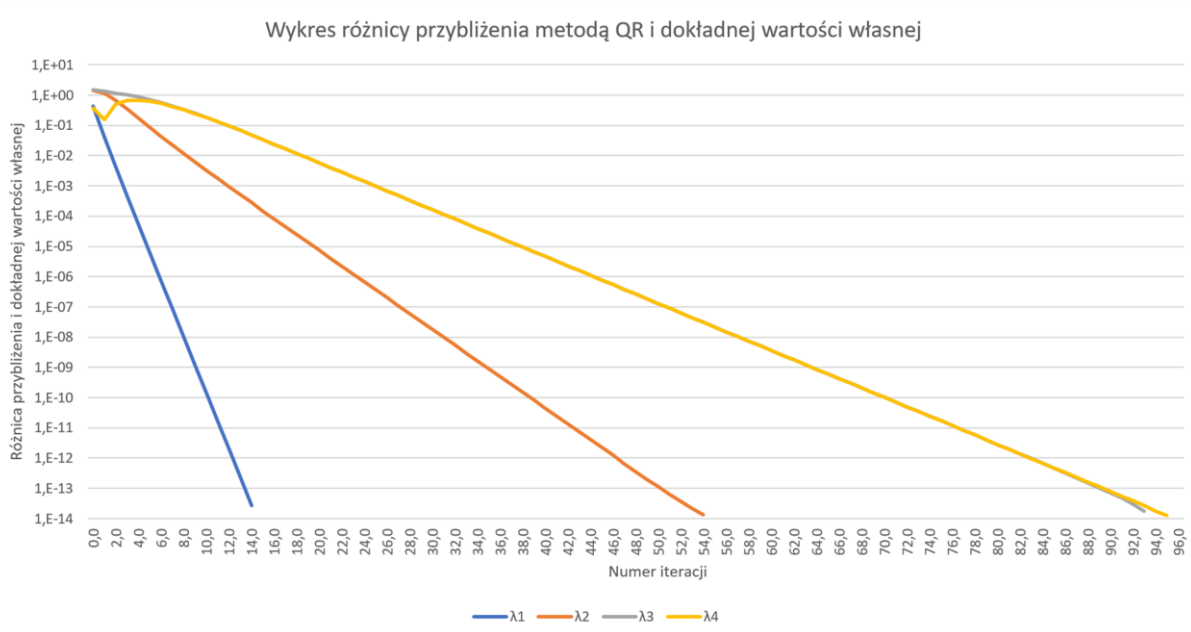
Iteracje, które były potrzebne: 96.

Powyższe wyniki sprawdziłem za pomocą biblioteki Eigen, były one następujące:

- 2.07816
- 6.03172
- 8.14772
- 9.74239

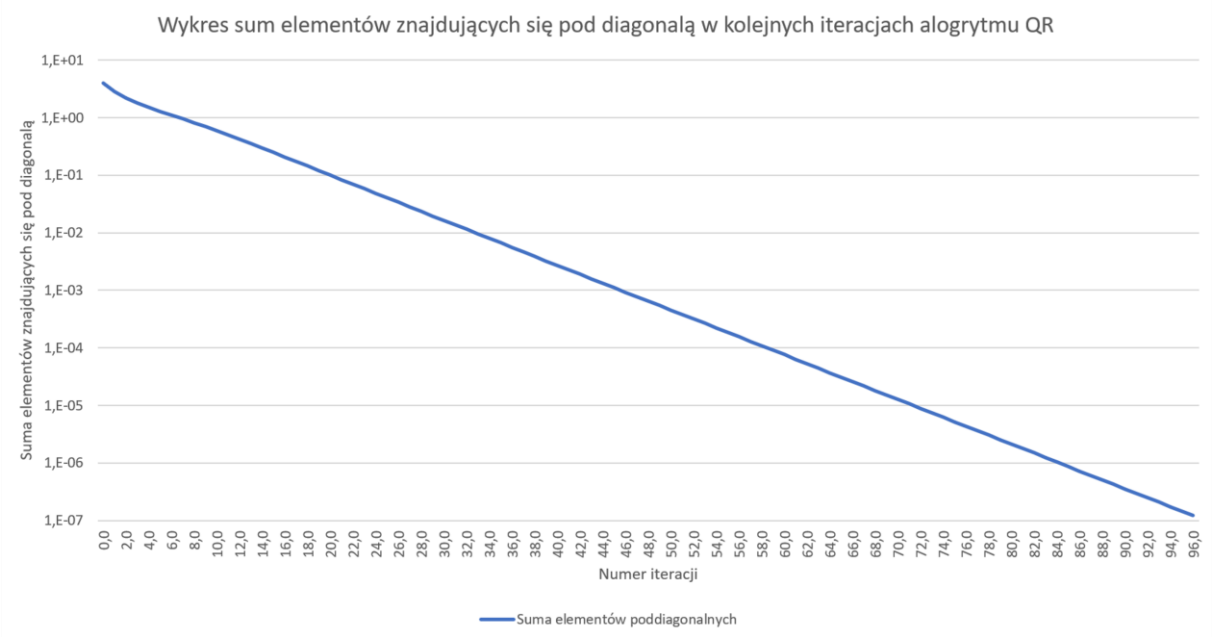
Jak widać, są one identyczne – różnią się jedynie kolejnością.

Poniżej przedstawiam wykres wykonany w skali logarytmicznej, pokazujący jak elementy diagonalne ewoluowały w czasie (czyli jaka była różnica pomiędzy przybliżeniami a odpowiadającymi im dokładnymi wynikami) podczas kolejnych iteracji.



Jak możemy zauważyć, błąd dla każdej wartości własnej sukcesywnie malał, a program potrzebował 96 iteracji, aby osiągnąć zbieżność na przyjętym poziomie.

Podczas gdy elementy diagonalne zbiegały do wartości własnych, elementy znajdujące się pod diagonalą zdążały do 0 – dobrze ilustruje to wykres sum elementów poddiagonalnych, który zamieszczam poniżej (jest on również wykonany w skali logarytmicznej).



Widać na nim wyraźnie, iż z każdą kolejną iteracją elementy te były coraz mniejsze – osiągały wartości coraz bliższe zeru. Zdaje się to dobitnie demonstrować, że macierze przekształcane w kolejnych iteracjach upodobniają się do macierzy górnotrójkątnych.

Wnioski

Podczas działania algorytmu QR, macierze, które powstają podczas wykonywania się kolejnych iteracji w istocie upodabniają się do macierzy trójkątnej górnej. Ponadto, jako rezultat działania programu własności własne, które odczytujemy pod koniec działania programu z diagonali są równe dokładnym wartościom własnym, obliczonymi za pomocą biblioteki Eigen.

Podsumowanie

Zarówno stosując metodę potęgową, jak i algorytm QR możemy dokładnie wyliczyć potrzebne wartości własne. Przesunięcie Wilkinsona zapewnia dużo szybsze osiągnięcie zbieżności, które usprawnia działanie programu.

Wnioski

Zastosowanie metody Wilkinsona znacząco poprawia efektywność programu. Zastosowanie przesunięcia do metody potęgowej skróciło potrzebą ilość iteracji o ponad dwukrotność w porównaniu do metody potęgowej bez przesunięcia, czy też algorytmu QR.